TECNOLOGIE INFORMATICHE PER IL WEB

Federico Mainetti Gambera

3 aprile 2020

Indice

LEZIONE 1 11/03/20
link clicca qui

Calendario delle lezioni

`https://docs.google.com/spreadsheets/d/14ShTdkFCZ63MlyKP0LCSLPsPAeiu8D2sYVXSrkw9B6k/edit#gid=0`

Prerequisiti

Introduzione al corso - Evoluzione architetture
Protocollo HTTP
HTML 3 e 4
CGI

Introduzione al corso

Non ci saranno prove in itinere per questo corso.
E' opportuno guardare i video relativi agli argomenti indicati prima di partecipare alla lezione.
Queste lezioni hanno lo scopo di aggiungere informazioni, chiarire dubbi e aiutarci a capire quali siano gli argomenti più importanti e i punti chiave.
I materiali video coprono interamente gli argomenti del corso.

1 Architetture

Il web è una piattaforma per sviluppo di applicazioni con un architettura molto particolare. Per architettura si intende l'insieme delle risorse (hardware, connettività, software di base, software applicativo). Le applicazioni web hanno una particolare conformazione della loro architettura che prevede tre livelli: Client, Middle-tier, Data-tier.
Le architetture sono cambiate molto negli anni. Se ne individuano tre grandi famiglie: Mainframe, Client-Server, Multi-tier (per esempio le applicazioni web).
Il compito di questo corso è quello di riuscire a programmare nel Middle-tier con qualche accenno al Client. In un'applicazione web il Client interagisce con il Middle-tier con un preciso protocollo, che non è il classico tcp-ip, ma il protocollo HTTP.
[Una lettura molto importante per il corso è il documento RequestforComment 1945, Tim Berners Lee (https://www.w3.org/Protocols/rfc1945/rfc1945), che rappresenta l'atto di fondazione del web.]
Il Client manda delle request al Server che invia delle responses. Le request del Client sono gestite tramite un applicazione detta User agent (browser).
Http è rivoluzionario per la sua semplicità: le richieste del Client e le risposte del server sono delle semplici stringhe.
Architettura delle applicazioni web: c'è un client (pc) con un user agent (browser) che emette richieste che vengono servite con delle risposte da un web server, detto anche HTTP server. L'archiettura di cui però ci occupiamo è più complicato di così, perchè vede alle spalle del web server un application server (TomCat), che ha lo scopo di calcolare una risposta "personalizzata" secondo parametri e criteri precisi e di produrre un pagina web appropriata.
[Installare un'archiettura completa sul pc che prevede tutti i livelli appena visti (guida nella cartella strumenti) (si può usare anche IntellyJ invece di Eclipse)]
Altri elementi dell'architettura sono il proxy e il gateway. Il proxy è un intermediario fra un client e un origin server, per definizione può comportarsi sia come server sia come client. Fa copie di risorse e le inoltra ai client e può essere anche utile per limitare gli accessi e gestire autorizzazioni. Il Gateway serve per far interagire un applicazione web con applicazioni che non usano il protocollo HTTP, quindi il gateway traduce richieste HTTP per applicazioni che non lo comprendono, tipicamente usato per SQL.

2 HTTP

Come si identifica una risorsa? con l'URL (Uniform Resourse Locator) che è una striga formatta in maniera molto semplice: prefisso (protocollo), indicazione della macchina fisica in cui è presente la risorsa, una eventuale porta in cui la macchina ascolta le richieste e un Path. Gli URL sono estremamenti semplici, e utilizzano concetti già noti in precedenza.
HTTP request: è una banale stringa che contiene una request-line, degli header e un allegato (per gli upload) facoltativi. la request-line contiene tre informazioni: il metodo (funzione) della richiesta, l'URL, la versione del protocollo usata. I metodi di richiesta sono principalmente due, GET e POST; questi metodo possono essere visti come chiamate di funzione.
HTTP response: anche questa è una banale stringa, che, ancora più semplicemente contiene un codice di stato (es. 404 file not found), degli header facoltativi, e un allegato.
La conseguenza di un protocollo così semplice è che con un solo client si può interagire con tutti i back end. La complessità si sposta però nell'application server.
Gli header sono informazioni opzionali a cura del browser, trasmesse come fossero parametri che aggiungono informazioni alla request o alla response.

3 CGI, Common Gateway Interface

Tecnologia ormai morta e superata.
CGI è una convenzione che standardiza un certo numero di variabili d'ambiente (condivisibili fra più processi) grazie alle quali il web server può smontare una richiesta HTTP e salvarla in una zona di memoria alla quale un processo di un applicazione esterna può accedere.
Per invocare un applicazione tramite CGI, l'URL della richiesta HTTP deve presentare un path che porta a un applicazione eseguibile. Una volta ricevuta una richiesta HTTP che rimanda a un eseguibile, il web server smonta la richeista, ne preleva le informazioni (parametri) e le salva nelle variabili d'ambiente, successivamente fa eseguire all'application server il programma indicato nel path dell'URL. L'application server preleva le informazioni dalle variabili d'ambiente e costruisce un file HTML che poi verrà mandato come risposta al client.
CGI ha due grandi difetti, il primo riguarda la sicurezza, in quanto i file eseguibili erano direttamente accessibili, il secondo è che siccome i processi dell'application server muoiono dopo ogni esecuzione, non c'è modo di creare un sistema che mantiene una sessione attiva. Inoltre le prestazioni di CGI sono molto basse, per esempio, siccome i processi muoiono continuamente, c'è un continuo bisogno di stabilire connessioni coi database, che è un operazione onerosa.

LEZIONE 2 12/03/20
link clicca qui

Prerequisiti

Fondamenti Servlet - base

4 Servlet - base

Nasce l'esigenza di generare contenuti dinamici per le applicazioni web, inizialmente, infatti, HTTP era concepito come protocollo per scambio di documenti.
Il gateway è un elemento imprescindibile che aumenta le capacità di un'applicazione web. Abbiamo visto che la versione arcaica di gateway era il CGI, che rappresenta il modo più semplice e che usa le variabili d'ambiente per adempiere al suo compito. Abbiamo però visto che CGI ha diversi problemi, perciò è necessario trovare una nuova soluzione.
Vediamo ora il meccanismo più popolare (almeno fino a qualche anno fa) per realizzare i requisiti di produzione dinamica di contenuti. HTTP è nato per essere semplice, ha solo due metodi, GET e POST, non prevede l'identificazione di un client, tutte le richieste sono uguali. Per un Web Server la nozione di sessione non esiste. I metodi GET e POST sono richieste parametriche, cioè alle quali si possono aggiungere informazioni sotto forma di parametri stringa. Il metodo GET i parametri li mette nella query-string, il metodo POST li mette nel body. HTML ha un costrutto "form" che serve al browser per costruire un richeista dove i parametri sono complicati, per esempio richieste dove un parametro è un file. Il metodo POST è quindi spesso usato con il costrutto HTML form.
Java Servlet è un modo nuovo, rispetto a CGI, di strutturare l'applicazione che riceve la request e formula la response.
Al programmatore di un Servlet si chiede di programmare una classe, egli dovrà lavorare all'interno di un Framework, cioè una soluzione parziale a una serie di problemi con caratteristiche comuni. Il framework è uno schema, uan soluzione parziale che omette la parte variabile di una serie di applicazioni con qualcosa in comune. Java Servlet è appunto un framework, tutte le applicazioni web hanno in comune tutto il protocollo HTTP. Dal framework ci aspettiamo quindi di non dover programmare la gestione delle request e delle response.
Il Servlet container è un ambiente che esegue il tuo programma, nel nostro caso è Tomcat. Il container materializza l'API del framework che stiamo usando, per esempio il metodo doGet() di cui noi facciamo l'override viene chiamato dal Servlet container.
Alle spalle del web server dunque sta la JVM, all'interno del quale risiede il servlet container che gestisce i Servlet che il programmatore scrive.
Un primo beneficio fra che si nota nel programmare in un ambiente così fortemente strutturato è che, diversamente da CGI, i Servlet vengono eseguiti all'interno di un ambiente persistente e quindi può eseguire più di una richiesta senza essere terminato. Non c'è bisogno di preoccuparsi della concorrenza, semplicemente si programma un Servlet pensando a come deve rispondere a una certa request. Siamo quindi in presenza di un ambiente stateful. C'è un prezzo da pagare per questo beneficio, ovvero che non siamo noi a gestire la concorrenza e quindi i thread, il modello di concorrenza lo ereditiamo dal contenitore. Il modello da seguire è chiamato "one thread per request" e significa che una volta sviluppato un servlet non siamo noi a invocare la "new" su quell'oggetto, perchè è il contenitore a gestire questo aspetto e lui ne avrà sempre e soltanto uno istanziato. Tutte le request interagiranno sempre con lo stesso oggetto istanza della Servlet programmata da noi. Quindi, si programma una servlet, ce ne ssarà una soltanta istanza, un solo oggetto, tutte le request che riceviamo (tante nello stesso istante) avrà un thread allcoato, non da noi, ma dal container e questi thread agiranno tutti sulla stessa istanza del Servlet. Questo processo ha ovviamente un grande problema: le variabili della classe Servlet sono condivise per tutte le request.
Un framework ti da servizi, ma ti impone dei vincoli.
Dal punto di vista materiale un framework è una libreria e noi in particolare useremo javax.servlet e javax.servlet.http, che sono interfacce implementate dal contenitore.
Il contenitore mappa le request sui metodi doGet() e doPost().
Il metodo destroy() viene chiamato quando il processo dell'applicazione viene stoppato, fatto che è deciso dall'amministratore del server.
Il file web.xml serve per mappare le richieste http al corretto servlet.
La programmazione per Servlet è una programmazione per componenti, nel senso che il programmatore scrive solo i componenti variabili per l'applicazione, tutto il resto è gestito dal framework.

Il servlet Context è una scatola che contiene diversi componenti che presi nell'insieme rappresentano un'applicazione. L'oggetto java che rappresenta un'applicazione è proprio il servlet context, infatti quando devo fare la deploy di una applicazione si distribuisce il servlet context. Il servlet context è un insieme di risorse che contiene i sorgenti delle applicazioni java scritte dal programmatore, i file di configurazione e le risorse stesse (per esempio le immagini).
Il file web.xml contiene la configurazione dell'applicazione, fra cui la mappatura delle varie servlet. Per maggiori informazioni guardare i video. In breve: c'è Tomcat che è rappresentato dal server stesso, il contenitore, nei nostri esempi sarà localhost:8080, a Tomcat arriva una richiesta. Una richeista appende all'indicazione del server (localhost:8080) un url, che è quello che consente di scatenare la servlet corretta. Tomcat viene istruito tramite un processo di Servlet mapping. Per maggiori informazioni guardare i video o i lucidi.

LEZIONE 3 18/03/20
link clicca qui

Prerequisiti

Fondamenti Servlet - esempi

Informazioni

Su Beep nella sezione Documents and media > Esercizi > Servlet - jsp - jdbc > esercizio server side messaggi, si trova un esempio (progetto eclipse completo .zip) commentato con video e power point. E' importante, dice il prof, perchè presenta tutto ciò che può esserci in esame.

Sempre su Beep, ma nella sezione Documents and media > Esercizi > CSS c'è un esercizio, che però è integralmente spiegato nella cartella delle video lezioni riguardanti Css.

Useremo i giorni di sospensione (8-9 aprile) pe le prove in itinere per fare degli esercizi autonomi.

Verranno pubblicati anche altri progetti, man mano più complessi.

Sono molto importanti la documentazione sul package javax.servlet.http e il package javax.servlet.

5 Servlet - esempi

Una programmazione in servlet è una programmazione che ci chiede di rispettare alcune regole per trarne dei benefici. I benefici principali sono la scalabilità, la gestione automatica dei cicli di vita, la mascheratura degli aspetti tecnici di HTTP. Java servlet ci permette di interagire com HTTP attraverso degli oggetti. Uno dei primo esempio di oggetto che si incontra è l'oggetto request.

5.1 Esempio: contatore condiviso fra i client

Il file web.xml ci permette di controllare le impostazioni con cui vogliamo lavorare, inoltre ci permette di definire parametri a livello di applicazione (globali) o parametri a livello di servlet.
I parametri possono poi essere reperiti con chiamate di metodi di oggetti di sistema in maniera molto semplice.
Con questo esempio, oltre a mostrare l'utilizzo di questi parametri, viene mostrata l'utilità dei dati membro all'interno della servlet (gli attributi dell'oggetto servlet). Esistendo una sola istanza di servlet per tutte le request, ci aspettiamo che i dati membro vengano usati in maniera concorrente per tutti i client.
Nell'esempio particolare del contatore possiamo vedere che tutti i client possono incrementare il valore del contatore, perchè condividono il dato membro della servlet.
Quindi con questo esempio si imparano due cose importanti: come si configura una servlet per farla partire e il particolare modello di gestione della concorrenza e delle variabili.

5.2 Esempio: contatore per ogni client

Con questo esempio andiamo ad analizzare il meccanismo di una sessione. Per sessione si intende un gruppo di richieste che possono essere fatte risalire a un unico cliente. HTTP ha solo il metodo GET e POST, e quindi sembra che non sia lui ad occuparsi della gestione della sessione, infatti la gestione della sessione non era nativamente intesa all'interno del protocollo HTTP, ma si è sviluppata successivamente grazie all'utilizzo degli header, in particolare esistono due tecnica: o l'uso degli header cookie, oppure grazie all'url rewriting.
Da un punto di vista programmativo queste due tecniche sono trasparenti per noi programmatori, noi otterremo un oggetto che maschererà i meccanismi che stanno dietro alla gestione delle sessioni.

5.3 Forme di identificazione

Il protocollo HTTP nella sua versione più base (senza i metadati aggiuntivi) serve richieste anonime.
[tema molto importante per il professore, spesso usato all'orale:] Analiziamo la terminologia e i concetti

base che stanno dietro alla sicurezza della gestione delle sessioni (con forme di identificazione del client via via più stringenti):

- Pseudo identificazione: l'atto con cui il server associa un'etichetta arbitraria alle richeiste del client allo scopo di identificare quelle che provengono dallo stesso client. Come esempio si per vedere in atto la pseudo identificazione si può usare il "SessionCounter", aprire due browser (uno in incognito e uno no) e vedere come i due contatori non sono condivisi. La pseudo identificazione non ha bisogno che l'utente si logghi.
- Identificazione: è il processo con cui il cliente dichiara un'identità. Non ci stiamo più riferendo a un client, ma a un user, un account, una persona.
- Autenticazione: è il processo in cui il server verifica l'identificazione del client. Il client e il server condividono un "segreto" (password). La differenza fra pseudo identificazione e identificazione-autenticazione è che nel primo il server inventa un label da dare a un client, che comunque rimane anonimo in quanto l'identità è fornita dal server, e nel sencondo il client si dichiara un user che poi viene verificato dal server, quindi in questo caso l'identità proviene dal client.
- Autorizzazione: è il processo in cui il server garantisce l'accesso a risorse e processi a un utente identificato e autenticato. L'autorizzazione è una proprietà dell'identità verificata. E' in poche parole il permesso di vedere o fare cose particolari a fronte di un'identità verificata.
- RBAC (Role based access control): è uno schema di autorizzazione che garantisce diritti non solo sulla base della tua identità, ma anche grazie al tuo ruolo.

5.4 Tracciemento della sessione

Si dice cookie un piccolo contenuto di informazioni che il server installa sul browser del client, allo scopo che venga restituito al server a ogni successiva richiesta per poter tracciare un client e la sua sessione.
Vediamo ora in maniera "tecnica" come funziona la pseudo-identificazione tramite gli header cookie. Il server, quando riceve la prima richiesta da parte di un client (quindi una richiesta anonima), vuole che la seconda richiesta sia pseudo-identificata (non più anonima). Se andiamo ad analizzare la seconda richiesta infatti ci accorgiamo che fra i request header è presente, nella voce cookie, il label che il server ha generato per me. Infatti se andiamo a vedere la response alla prima request, notiamo che il server propone al client un label tramite l'header set-cookie. Dunque alla seconda richiesta il client restituisce al server questo cookie e quindi riesce a pseudo-identificarsi.

Fra i vari esercizi caricati su beep ce n'è uno che si chiama cookie inspector, con cui possiamo andare a fondo con la questione dei cookie. Come esercizio provare anche a disabilitare i cookie e a vedere cosa succede.

I cookie non sono un canale sicuro, possono essere sniffati e si possono fare attacchi man in the middle.

L'identificazione autorizzata non sostituisce la pseudoidentificazione. Immaginiamo che avvenga sia una psaeudoidentificazione sia una identificazione autenticazione con per esempio un login, quindi il mio client è stato autenticato e la sessione è tracciata dai cookie. Da ora in poi è ovvio che la sola pseudoidentificazione non implica l'autenticazione precedentemente fatta (altrimenti dei semplici attacchi informatici potrebbero causare grandi problemi di falsa appropriazione di identità), ovvero il tracciamento della sessione e l'identificazione autenticazione sono su due piani diversi.
Il session tracking implica la psaudoidentificazione del client, o con cookie, dove il server inietta una stringa e il browser la restituisce a ogni futura richiesta, o con url rewriting, ormai non più usato.
Invece allo scopo di associare allo pseudo-cliente una qualunque informazione durevole (per esempio l'autenticazione di un client, oppure l'inserimento di un prodotto in un carrello), il server abbina a ogni pseudoidentità uun oggetto Java chiamato Session che è controllabile dal programmatore della servlet. In questo oggetto può essere presente un informazione che conferma che questa pseudoidentità è autenticata e corrisponde a un determinato user. In questo oggetto Java, presente nel server container e totalmente all'oscuro del browser, si possono depositare informazioni legate a una pseudoidentità (come per esempio il fatto che il client si sia identificato e autenticato).
Un esempio di utilizzo dell'oggetto Session si può vedere nel servlet fornito dal professore chiamato "SessionCounter".

Possibile domanda da orale: c'è bisogno del login per personalizzare una pagina? no, basta il concetto di session tracking. A cosa serve il login? Il login serve ad associare ad una pseudoidentità un'identità autenticata allo scopo di fare autorizzazioni.

In lezioni > 8-Sevlet-base > progetto eclipse-esempi-base-servlet e servlet con JDBC, ci sono progetti con cui possiam olavorare per scavare più a fondo in questi argomenti.

LEZIONE 4 19/03/20
link clicca qui

Prerequisiti

Connessione a database con JDBC

Informazioni

Informazioni sul progetto.
Nuovo progetto caricato su beep: "Esercizio gestione spese di trasferta", è commentato (slide e readme.md) e contiene una base di dati. Contiene concetti molto improtanti su JDBC.

6 JDBC

Gli argomenti di questa lezione verranno corredati alla visualizzazione di un progetto nella cartella su Beep delle lezioni > Servlet-base > Progetto eclipse servlet con jDBC.

JDBC è un architettura molto elegante per il mondo Java e serve per mettere in contatto applicazioni Java con DB.
JDBC eredita la sua logica da ODBC, un architettura sviluppata originariamente da Microsoft.

Il vantaggi di JDBC sono legati al mascheramento delle differenze che sono presenti nelle modalità con cui un'applicazione interagisce con una base di dati. Il principale aspetto di JDBC è quello di migliorare la portabilità dei sistemi che fanno uso di DB. Per ottenere questo sfrutta due tecniche: rimuove la necessità che il programmatore conosca le tecnicalità con cui avviene la connessione coi DB, maschera alcune differenze "dialettali" nell'uso di primitive SQL che alcune basi di dati offrono e altre magari offrono in maniera diversa.

L'architettura JDBC struttura la connessione con la base di dati in maniera intelligente, che consente all'applicazione di ignorare l ospecifico modello della base di dati. Il modello utilizzato è **stratificato**: l'**applicazione Java** vede solo un'**API** (application programming interface) al di sotto del quale c'è un ambiente di runtime che, a fronte di un particolare modello di base di dati, carica un determinato Driver. Per un sistema operatico, un Driver è un programma speciale che contiene le istruzioni di gestione di una determinata periferica, questo stesso meccanismo è stato replicato per sviluppare JDBC. Più in dettaglio, l'API utilizza un servizio chiamato **Driver Menager** che ha lo scopo di caricare un determinato **Driver** a seconda di con quale base di dati dovrà interagire.

L'effettiva comodità di JDBC risiede nei vantaggi che il programmatore ne trae, ovvero che dovrà interagire con una serie di oggetti di utilità: Connection, Statement, ResultSet, SQLException (Driver e DriverManager li vedremo una sola volta). L'intera architettura JDBC è fatta da queste classi.

JDBC è utilizzato anche al di fuori dell'ambiente web.

Per l'uso di JDBC all'interno di una Servlet si usa il seguente workflow (pattern):

- connsessione;
- preparazione ed esecuzione di query;
- precessamento dei risultati;
- disconessione (molto importante, siccome la base di dati è una risorsa comune a tutte le servlet, lasciarla "libera" è essenziale, talvolta la disconnessione precede anche il processamento dei risultati; un metodo più avanzato per distribuire in maniera efficiente le connessioni è il cosiddetto "connection pooling", più informazioni sul video di questo argomento).

Nel momento il programmatore non ha controllo sul ciclo di vita della propria applicazione (come nel caso dei Servlet) si prevedono dei pattern specifici riguardanti le quattro operazioni appena elencate.
Connessione: avviene nel momento di creazione della Servlet, cioè nella funzione init() della Servlet

(nelle slide della lezioni ci sono più informazioni sull'esatto procedimento), i parametri (url, utente, pass, modello DB) necessari alla connessione al DB sono tipicamente inseriti all'interno del file xml di configurazione dell'applicazione web;
oss. tutti i blocchi riguardanti l'interazione con DB saranno contenuti in blocchi critici (try... catch... finally), nell'esercizio di gestione spese di trasferta su Beep c'è una variante molto carina, che abbrieva molto la lunghezza del codice.
Disconnessione: simmetricamente la disconnessione è eseguita nel metodo destroy(), per cui la servlet ha una connessione disponibile per tutta la sua durata.

La connessione e disconnessione sono le operazioni più costose all'interno di un DB, dunque ci sono tecniche avanzate che permettono di sfruttare al meglio le connessioni (vedi connection pooling).

Questo schema lo vedremo in tutti i progetti caricati.

In questo schema c'è un grande problema. Un processore di query è un componente che riceve una stringa query SQL, lo analizza (parsing), costruisce l'albero sintattico della query, lo ottimizza, lo compila in un piano di accesso disco e infine lo esegue. Il grande problema è che non c'è bisogno di ricompilare il piano di accesso disco per tutte quelle query che sono "simili" fra di loro (cioè che differiscono per alcune variabili), quindi mandare query a ricompilare continuamente spreca un sacco di tempo. Come soluzione a questo problema intervengono le **parametric prepare statement**, che utilizzano degli scheletri di istruzioni con dei "?" al posto dei valori che potrebbero cambiare, che sono dette variabili di binding, e vengono compilati in piani di accesso una sola volta ed eseguiti quante volte si vuole con le variabili che si vuole. Se una query è molto complessa e la sua preparazione è molto onerosa, sfruttare questi prepare può rappresentare un vantaggio molto ingente.
"Mai mai mai fare query al volo, è sempre buona pratica prepararle prima".

Le **SQL injection** sono un problema che riguardando l'esecuzione di query che sono frutto della concatenazione di una parte sicura con una parte insicura.
Per parte insicura si intendono stringhe senza controlli di formattazione da parte del programmatore. Questi problemi insorgono tipicamente quando la query è formata da una parte programmata e una parte che è il risultato di un parametro o un campo inserito direttamente (senza controllo da parte del programmatore) dall'utente. Per esempio se usiamo una query "select * where ID = " + "[input dell'utente]", un utente male intenzionato potrebbe scrivere "1 OR 1=1", ricevendo quindi come risposta l'intera tabella dal database, in quanto 1=1 sempre vero. Il concetto è che dobbiamo evitare di dare all'utente la possibilità di poter modificare l'albero sintattico della queri (nel caso appena mostrato è successo quando l'utente ha inserito "OR") Questo meccanismo può creare grandi falle di sicurezza.
Il rimedio a questo problema è l'utilizzo dei prepare statement, infatti preparando le query, l'albero sintattico viene precreato e quindi l'utente può solo e soltanto inserire parametri, che se sono mal formati, la base di dati produce un errore, invece di rilevare informazioni che non dovrebbe.

LEZIONE 5 25/03/20
link clicca qui

Prerequisiti

Templating con JSP

7 JSP

JSP (Java Server Pages) serve per la costruzione di interfaccie utente tramite template.

In tutti gli esempi che abbiamo visto le interfaccie utente sono estremamente basichem mentre nelle applicazioni reali sono uno dei fattori più imortanti.

L'approccio che abbiamo usato fino ad ora mescola diverse funzionalità: l'aspetto di richiesta dei dati (Data Base), la formattazione dell'interfaccia utente (HTML), la logica applicativa. Facendo così i programmi diventavano complessi e inleggibili. C'è bisogno di separare i diversi aspetti di un'applicazione.
Il flusso di lavoro al quale si vuole giungere è:

- Il progettista grafico crea un esempio di come le pagine dovranno apparire usando contenuti fittizzi;
- Il programmatore sostituisce i contenuti statici con contenuti dinamici, computato dinamicamente a run-time;
- Nel ciclo di manutenzione il grafico può lavorare sugli aspetti estetici in maniera indipendente dal programma che ci sta sotto, può lavorare in un ambiente "grafic-friendly".

L'approccio che si usa è un cambio di prospettiva. Prima le Servlet stampavano il contenuto grafico, la nuova prospettiva prevede che il contenuto grafico contiene del codice.

Prima fase (Servlet): Servlet stampano contenuto HTML.

Seconda fase (JSP): Cambio di prospettiva, file HTML contengono tag speciali che permettono l'utilizzo di codice Java. In questa fase si ha un cambio di prospettiva, ma layout e calcolo sono ancora molto legati.

Terva fase (JSTL): file HTML non contiene più codice Java, ma contiene dei tag in formato HTML che permette di ottenere lo stesso risultato della seconda fase, ma mantenendo il file in formato più grafico. Questi speciali tag utilizzati non sono tag nativi di HTML e quindi il server li processa per trasformarli in puro HTML. Queste marche si chiamano active tag o run-at-server tag, che un browser non sa interpretare. Questo metodo ci da l'illusione di lavorare con HTML.

Quarta fase (Thymeleaf): Contiene solo tag HTML, che però hanno attributi speciali. Questi tag sono in ogni caso interpretabili da un browser. Con questo approccio, il grafico e il programmatore possono lavorare sullo stesso template. Il prezzo da pagare è che Thymeleaf viene eseguito da un proprio ambiente (che non fa parte di Java EE), quindi serve una configurazione.

Oggi parliamo di JSP.

Quando una richiesta da parte di un client è indirizzata a un file .jsp, il server lo riocnosce, richiama la pagina .jsp e la converte in una Servlet (se non è già stato fatto)- La Servlet viene compilata (anche qui se non è già stato fatto) ed eseguita, da qui la risposta viene generata e inoltrata.
JSP è quindi modo per scrivere Servlet in maniera più comoda, tutto ciò che sappiam sulle Servlet vale anche per JSP.
I template JSP non si mappano, perchè funzionano come i file HTML, più avanti nella lezione vedremo un modo con il quale possiamo però evitare l'accesso diretto a un file .jps obbligando il passaggio per una Servlet.

Domanda trabocchetto: chi esegue i template? nessuno. I template non vengono eseguiti, ma convertiti in Servlet che verranno compilate ed eseguite.

Nasce una domanda spontanea: Il grafico crea un template, il programmatore sostituisce il contenuto statico con contenuto dinamico, ma da dove viene questo contenuto dinamico? Se devo usare delle istruzioni programmative per andare a recuperare questi contenuti, allora sto vanificando il senso stesso di usare i template. La soluzione che JSP propone è molto rozza ed è rappresentata dall'utilizzo di oggetti predefiniti (request, response, out, application, config, ...).
Con JSP si risolve il da dove prendere i contenuti, ma rimane ancora il problema di avere pezzi di codice Java (es. iterare sui contenuti degli oggetti predefiniti) all'interno del template.

Un altro problema a cui JSP cerca di dare una soluzione è sulla gestione di oggetti di utilità.
Vediamo un esempio: i dati di una form inseriti da un utente devono essere messi in una richiesta, che viene inviata al server. Questi dati devono essere immagazzinati in un oggetto o una struttura dati che deve essere mantenuta per esempio tutta la durata di una sessione.
Il problema di gestire degli oggetti di utilità all'interno dei template senza programmare è risolto tramite l'utilizzo di active tag (useBean, setProperty, getProperty,...), cioè tag che vengono eseguiti assieme al template che provocano la creazione di oggetti (di struttura standard) che vengono depositati in opportuni contenitori.

L'azione useBean crea un oggetto, l'azione setProperty imposta una proprietà di un oggetto, l'azione getProperty preleva una proprietà da un oggetto. Gli oggetti "bean" creati hanno una struttura standard e vengono messi in dei contenitori (scope), gli oggetti bean hanno due attributi: ID che li caratterizza e lo scope, che regola la visibiltà dell'oggetto, per esempio quale web component può accederci (page, request, session, application).

Adesso abbiamo due strumenti nelle nostre mani: le Servlet e i template. Ma chi fa che cosa? Le template potrebbero fare direttamente le richieste ai database per esempio, ma è meglio dividere i compiti: l'interfaccia utente consuma i contenuti e li presenta, la parte programmativa procura i contenuti. Ciò che collega la produzione e il consumo dei contenuti sono degli oggetti (bean) formattati in maniera standard: JavaBean.
I JavaBean sono classi standard con attributi e getter e setter. Grazie all'utilizzo di questi standard, le istruzioni Java possono essere tradotte e semplificate all'interno dell'html, in modo tale da non dover programmare lì.

Farwarding: Una servlet può chiedere a una request un dispatcher, dare al dispatcher una destinazione (Servlet o template) e fare farward, cioè di passare la request a chiunque sia stato detto al dispatcher. Forward è il metodo parallelo rispetto al redirect per la separazione dei compiti.
Domanda classica da esame: Che differenza c'è fra redirect e farward? Redirect spezza il compito in due request, farward spezza il compito in una sola request, cioè i due componenti condivideranno la stessa request.
Un tipico esempio di suddivisioni dei compiti: una request viene gestita inizialmente da una Servlet che fa la query alla base di dati, poi trasforma il risultato in una serie di Bean, mette i bean nell'oggetto di scope requeste e poi fa il farwarding al template.

Come si organizzano (usanto anche i template) i vari componenti di una applicazione web?
L'applicazione web viene gestita su due livelli: uno strato che dipende dal web (web tier) e uno strato che non dipende dal web (Logica applicativa e accesso ai dati o Business and data tier).
Questa struttura è molto comoda, perchè per esempio lo strato non legato al web può essere riciclato con un nuovo strato legato al web (per esempio se si vuole rinnovare l'interfaccia utente si può ancora lavorare con la stessa logica applicativa).
I due strati sono legati dagli oggetti di modello (JavaBean), che sono prodotti dallo strato logico e consumato dallo strato web.
Analiziamo il business and data tien: la proposta del prof è quella di mediare l'accesso al database con una serie di oggetti che nascondono l'SQL, che chiameremo DAO (Data Access Object). Così lo strato web è completamente indipendente dalla base di dati. Nel momento in cui si vuole cambiare sistema DB, è sufficiente modificare il DAO. Gli oggetti DAO ritornano dei Bean. Abbiamo disaccoppiato l'accesso ai dati da chi li consuma.
Il Web Tier separa gli aspetti di controllo (gestione di eventi come l'arrivo di request) da chi stampa l'effettiva interfaccia utente (HTML). Gli aspetti di controllo sono gestiti dalle Servlet (controller) e

l'interfaccia utente è gestita dai template (view) e i meccanismi di redirect e farward servono alle sevlet per passare il controllo alle view. Le view ottengono i dati dai JavaBean forniti dai DAO o dal controller se i dati vengono dal'utente stesso.
Questo aproccio prevede un thin client, anche detto pure-html, in quanto il client è al di fuori dello strato del web tier, non si occupa di nulla se non di ricevere i file html e di stile. Ovviamente questa non è la situazione delle macchina moderne.
Nell'esempio della bacheca messaggi su beep viene utilizzata questa struttura e d'ora in poi useremo sempre questo approccio.

LEZIONE 6 26/03/20
link clicca qui

Prerequisiti

JSTL

8 JSTL

JSP ha una serie di tag attivi (useBean, setProperty, getProperty) che hanno lo scopo di arricchire il template di funzionalità nascondendone il codice dietro a degli elementi pseudo html. In JSP per programmare un custom tag bisogna costruire una classe (tag handler) che implementa un'interfaccia con una serie di metodi (per esempio doStartTag() e doEndTag()) che operano sul contenuto del tag. Così la programmazione Java diventa completamente trasparente al template. Questa tecnica prende il nome di Code-behind.

Un primo svantaggio di questo metodo è che nel momento in cui un grafico va ad aprire un file con questi custom tag, non potrà visualizzarlo correttamente in quanto queste marche non possono essere interpretate da un comune editor di html.
Un altro aspetto svantaggioso di permettere di creare tag custom al programmatore è che molti di questi tag finiscono per eseguire la stessa cosa. Per questo motivo si è passati alla standardizzazione di una libreria di tag "universale": JSTL (JSP Standard Tag Library).

JSTL è un set di standard tag che possono essere usati in modo da evitare la proliferazione di codice duplicato.

JSTL contiene due componenti:

- un linguaggio di espressione EL;
- una libreria di tag.

Il linguaggio di espressione è EL (expression language) ed è stato inizialmente concepito come sussidio di JSP, ma ora è stato generalizzato per molti linugaggi di programmazione. Il linguaggio è abbastanza articolato, ma noi lo useremo a livello basico (quello che impariamo guardando esercizi e progetti è più che sufficiente).
EL è un linguaggio per la scrittura di espressioni, che sono costrutti la cui valutazione da luogo a un valore (mentre un istruzione è un costrutto che produce un effetto collaterale ed eventualmente produce un valore).
E' particolarmente importante che un linguaggio di espressione sia conciso e facilmente leggibile. EL permette l'accesso diretto agli attributi di uno scope JSP che ha il ruolo di mappa. EL permette anche la navigazione all'interno di una complessa struttura di un oggetto con la dot-notation.
Il contenitore a cui si fa riferimento non è necessariamente da esprimere, EL utilizza un sistema default in cui cerca l'attributo in tutti gli scope, dal più specifico al più generale (page, request, session, application).
EL ha anche una serie di oggetti impliciti che rappresentano degli shortcut per raggiungere dati spesso utilizzati (es.param, header, cookie, etc).

JSTLContructs ed ELexamples sono progetti in cui possiamo vedere tutti questi concetti applicati, con varie varianti sintattiche.

Il secondo componente è la libreria di tag, che è una famiglia di tag attivi che assolvono a determinati compiti. JSTL prevede 4 famigli, noi ne vedremo solo due, perchè le altre due sono poso utilizzare (XML Processing, ormai si usa Json) o addirittura quasi deprecate (SQL, secondo il nostro schema il template non si occupa di accedere al database). Le due librerie che useremo sono Core e Internationalization.
La libreria Core si occuppa dell'output, dell'accesso alle variabili e agli scope, di logica condizionale, di loop, di URLs e di error handling.
La famiglia Internationalization ha lo scopo di risolvere tutti i problemi di formattamento dei dati a seconda dei vari paesi del mondo di provenienza della richiesta (es. distribuire contenuto in lingue

diverse, convenzioni sui numeri con la virgola, il formato delle date, etc.) costruendo uno e un solo template. L'internationalization non può però automatizzare il contenuto dinamico, perchè dipende dal database che non è gestito nel web tier.

Ci sono numerosi progetti che mostrano l'utilizzo di tutti tag.

LEZIONE 7 01/04/20
link clicca qui

Prerequisiti

Templating con Thymeleaf

9 Thymeleaf

Perchè è nato Thymeleaf? Fin'ora abbiamo visto metodi di templating che o utilizzano codice Java o utilizzano tag attivi, in entrambi i casi il file html non è interpretabile da un web browser o da un editor di html, e perciò un grafico non può fare una preview di uno di questi file. Thymeleaf cerca di risolvere questo problema eliminando completamente tutto ciò che non è html e che quindi complica il coordinamento fra un grafico e un programmatore.
Vengono eliminati quindi i tag custom e introdotto il concetto di presentazione duale, cioè che il grafico possa aprire il file (offline) e abbia una resa visiva reale dell'interfaccia utente con contenuti statici e, allo stesso tempo, il programmatore possa aprire il file nell'application server e abbia anche lui una resa reale ma con contenuto dinamico.

Thymeleaf è una soluzione pratica e intelligente che permette al grafico e al programmatore di lavorare separatamente ma in maniera coordianta.

Thymeleaf ha solo un difetto, cioè che non è perfettamente integrato all'interno del ciclo di vita delle Servlet, come vedremo thymeleaf è un sistema esterno (come i DB) che rappresenta un processore di tempalte.

Thymeleaf può raggiungere elevate prestazione tramite il parsed tempalte caching (da notare è che il caching è molto forto e può capitare che, a seguito di una modifica di un template, l'interfaccia non venga aggiornata, in questo caso c'è da fare una pulizia manuale della cache su Eclipse e un riavvio di Tomcat).

Thymeleaf si basa su un principio: non inventare custom tag, ma aggiungiamo custom attribute ai tag html che già esistono. Facendo così i browser e gli editor html ingorano questi attributi (gli interpreti di html sono programmati per ignorare tutti quegli attributi che non sono standard) e lavorano solo sul contenuto html.
Nel template possiamo inserire i tag html, definire tutti gli aspetti della presentazione e poi aggiugnere degli attributi thymeleaf che non hanno alcuna influenza sulla presentazione. Così grafico e programmatore possono lavorare sullo stesso template senza intralciarsi. Il contenuto dinamico dei template è confinato negli attributi dei tag html.

Un template thymeleaf è una pura pagina html, con la diciarazione di un insieme di attributi specifici "th".

Il concetto più importante di thymeleaf è la sua dualità (nello stesso template c'è sia contenuto statico sia contenuto dinamico) che permette il coordinamento fra grafico e programmatore.

Uno svantaggio di thymeleaf è che, essendo un sistema esterno, deve essere avviato e poi essere utilizzato.
Per avviarlo usiamo una metodologia molto simile a quella che usiamo per l'altro sistema esterno, i DataBase. Notiamo che il metodo che presentiamo non è il più ottimale, ma è semplice (non affrontiamo il metodo migliore per una questione di crediti del corso). Nella funzione init() della servlet ci sono istruzioni che creano un oggetto tecnico "new TemplateEngine()", che è l'ingresso che la nostra applicazione ha con il processore di template thymeleaf. Una volta creato e salvato nella servlet (come facciamo pure per le connessioni ai data base), si customiza il TemplateEngine chiamando certi metodi con certi parametri.
Ora che abbiamo avviato il TemplateEngine, per usarlo non possiamo usare il metodo forward (perchè è un metodo che fa parte del ciclo di vita delle Servlet, mentre thymeleaf è un processo esterno), come per JSP. L'istruzione forward è sostituita da tre righe di codice (vedi slide) che rappresentano la creazione di un contesto per thymeleaf, ci mettiamo i contenuti dei modelli, e invece di forward usiamo

il metodo process.

Thymeleaf ha tre famiglie di espressioni:

- message expression: stampa le etichette, cioè le stringhe "fisse", che però in thymeleaf non sono mai fisse, perchè sono internazionalizzate per default (anche se si ha una lingua sola). Quindi tutte le stringhe base non sono mai scritte sul file html, vanno etichettate e sarà thymeleaf a inserircele.
- variable expression: per accedere agli oggetti del model o agli oggetti predefiniti
- link url expression: per costruire gli url

Le espressioni usano gli operatori standard.

Thymeleaf fornisce prefabbricati, ugetti di utilità di ogni tipo e strumenti molto comodi detti convertitori (ci lascia questo argomento da esplorare autonomamente).

Gli attributi più usati su thymeleaf (vedi slide): th:text=; th:href=; th:class=; th:remove.

Si possono mescolare attributi thymeleaf che cambiano il contenuto di un tag (es. th:text) o attributi thymeleaf che cambiano il valore di un altro attributo (es. th:href o th:class) con espressioni condizionali, looping e altro ancora. Si ha un potere di espressione massimo in una soluzione estremamente concisa.

LEZIONE 8 02/04/20
link clicca qui

10 Esercitazione

10.1 Esercizio bacheca messaggi

[Su beep troviamo le slide .pptx e i video che commentano la soluzione di questo esercizio e pure il progetto eclipse. In questi appunti non prenderò note in dettaglio sulla soluzione dell'esercizio in quanto su Beep è possibile reperire tutto il materiale necessario. Mi concentrerò di più sugli aspetti chiave e sui concetti fondamentali da usare per risolvere un generico esercizio].

Per tutti gli esercizi si parte dalla lettura di un testo che rappresenta i requisiti.

La prima operazione da fare è domandarsi quali sono i dati necessari per il supporto delle funzioni applicative. Per modellare i dati usiamo i concetti che dovrebbero già essere stati appresi durante il corso di basi di dati.
Per l'analisi dei dati si legge il testo e si cercano quelli che possono essere oggetti (entity), attributi degli oggetti (attributes) e proprietà che legano più di un oggetto (relationship).
L'analisi dei dati è un punto critico che spesso gli studenti tralasciano o fanno male e che può portare a sanzioni pesanti. E' bene esercitarsi su questa tecnica.

Dall'analisi dei dati segue la stesura dello schema concettuale, ovvero il Database design con lo schema entità relazione (da ripassare). Riguardo allo schema entità relazione ricordiamo: le entità sono dei quadrati, hanno degli attributi scritti a fianco e sono legate fra di loro con delle relationship, che sono dei rombi. E' importante segnare sempre le cardinalità (minima:massima)! Usiamo sempre la notazione di Peter Chan (?).

Ora si usano le regole di Stefano Ceri per la traduzione di uno schema concettuale in uno schema logico in linguaggio ddl (data definition language) SQL. Dobbiamo sempre mostrare le primary keys e le foreign keys. Per le chiavi si cerca sempre di usare un ID numerico con autoincremento (auto fornito dalla base di dati per default), ormai questo è lo standard dei database moderni, non cerchiamo di usare titoli o stringhe articolate come chiavi.
Un errore che spesso accade in esame è che gli studenti si scordano di mostrare lo schema logico (quello in codice).

Application requirements analysis: l'obbiettivo dell'analisi funzionale delle specifiche per un applicazione web è il cercare i componenti che poi andranno a costituire la nostra applicazione.
Grazie allo schema strutturale diviso in web tier e business and data tier, sappiamo già quali sono i componenti che dobbiamo andare a ricercare.
Utiliziamo un percorso graduale per cercare questi componenti.
Cominciamo a vedere quali sono le interfacce utente (le pagine che abbiamo) e gli eventi che l'utente può scatenare e le azioni che ne corrispondono.
Si cercano le pagine, poi i vari componenti di ciascuna pagina (form, tabelle, etc).
Come terza cosa si cercano gli eventi. L'evento più tipico sono le azioni che si possono fare per interagire con le pagine, esistono anche altri tipi di eventi, come le notifiche, ma solitamente in questi esercizi non ce ne sono. C'è sempre un evento nascosto implicito (non espresso nella specifica) in tutti gli esercizi che è l'accedere all'applicazione.
Come quarta cosa si cercano le azioni, cioè le risposte che l'applicazione ha allo scaturirsi di un evento. Per brevità ignoriamo le azioni di estrazioni di dati dal DB e che sono implicitamente previste dai contenuti dinamici. Focaliziamoci sulle azioni più esplicite come il compilare un form.

Una volta eseguita l'application requirement analysis, si fa uno schema, un disegno che lo rappresenti. Questo disegno possiamo farlo come ci pare, se ci piace usiamo IFML. L'importante è che si vedano le seguenti cose:

- le viste (in IFML sono dei rettangoli);
- i componenti delle viste (in IFML sono rettangoli smussati all'interno delle viste);
- gli eventi (in IFML sono dei pallini bianchi e con una freccia che mostra cosa succede);

- azioni (in IFML sono sono dei rettangoli allungati in cui il lato corto è tipo curvo... non so come spiegarlo, vedi le slide)

IFML non è obbligatorio, si può usare qualunque cosa, se volete potete usare IFMLEdit.org per creare dei grafici IFML.

Fino ad ora abbiamo analizzato il cosa tratterà la nostra applicazione, ora ci occuperemo del come avverrà.

Per tradurre il nostro schema IFML in qualcosa che mi esprima il come le varie cose verranno fatte, passiamo per l'architettura web-tier e Business and data tier. Il processo è molto facile e dobbiamo solo occuparci di definire gli oggetti di modello (beans), i DAO (data access object, avremo tipicamente un DAO per ogni entità, in più dovremmo analizzare i metodi che ogni DAO deve avere), i controllori (servlet, sono gli eventi) e le viste (template, sono le pagine dell'applicazione).

Per quanto riguarda il codice effettivo da scrivere su carta all'esame, il professore richiede solo tre cose: per primo tutti i metodi dei DAO, cioè il professore vuole vedere tutto ciò che è legato all'SQL nella nostra applicazione, in più viene solitamente richiesta la stesura di un controllore e di una vista.

Gli eventi vengono spesso facilmente analizzati con diagrammi di sequenza. Ogni evento a il suo diagramma di sequenza.
E' buona abitudine chiamare eventi che riguardano la restituzione di una vista con il nome "/GoToNomeDellaPagina" (non è un obbligo...).
Questi diagrammi non trattano la gestione degli errori. Solitamente si fanno più diagrammi, quelli per scenari regolari e quelli per i casi di errore.

10.2 Esercizio post management

Lo scopo di questo secondo esercizio è quello di mostrare un'esercizio i cui requirements sono meno chiari e la cui implementazione è scritta "meno bene" di quella prima.

Anche per questo esercizio è possibile reperire tutto il materiale necessario su Beep.