TECNOLOGIE INFORMATICHE PER IL WEB

Federico Mainetti Gambera

23 giugno 2020

Indice

Calendario delle lezioni

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

LEZIONE 1 11/03/20
link clicca qui

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 architetture sono cambiate molto negli anni. Se ne individuano tre grandi famiglie: Mainframe, Client-Server, Multi-tier.
Le applicazioni web sfruttano l'architettura Three-tiers che prevede tre livelli: Client, Middle-tier, Data-tier.
Lo scopo di questa prima metà di corso è quello di riuscire a programmare nel **Middle-tier**. Il Middle-tier si occupa di centralizzare la connesione ai database servers, maschera il modello dei dati al cliente e in genere può avere funzionalità varie di complessità anche elevata.

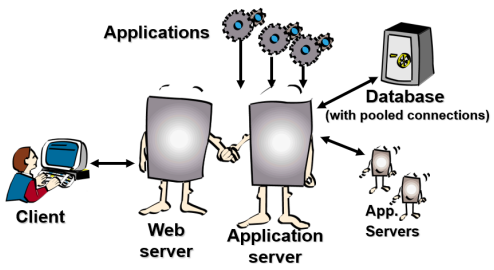
In un'applicazione web il Client interagisce con il Middle-tier con un preciso protocollo, che non è il classico TCP-IP, ma, nel nostro caso, è il protocollo HTTP.

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 meccanismo principale del protocollo HTTP consiste in requests mandate dal Client al Server, che a sua volta 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** (HTTP request) che vengono servite con delle **risposte** (HTTP response) dal Middle-tier, in particolare da un **web server**, detto anche **HTTP server**.
L'archiettura di cui ci occupiamo è, però, più complicata di così, il Middle-tier prevede alle spalle del web server un **application server** (TomCat), che ha lo diverse funzionalità, fra le quali, per esempio, quella di calcolare una risposta "personalizzata" secondo parametri e criteri precisi e di produrre un pagina web appropriata o di connettersi ai database SQL e di interagirci.
L'application server a sua volta risiede in un **origin server**, che è il server nel quale le risorse risiedono (o vengono create).

Altri elementi dell'architettura sono poi 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** è colui che agisce da intermediario per altri server ed è tipicamente utilizzato per far interagire un applicazione web con applicazioni che non usano il protocollo HTTP: il gateway traduce richieste HTTP per applicazioni che non lo comprendono, come i sistemi SQL.



2 HTTP

HTTP sta per **HyperText Transfer Protocol**.

HTTP è **stateless**, cioè ogni ciclo di richiesta-risposta è indipendente, non si preserva alcun tipo di dato fra connessioni diverse. Essendo stateless è anche **sessionless**.
Al contrario l'application server può essere **statefull** e di conseguenza essere **sessionfull**.

Come si identifica una risorsa? con l'**URL** (Uniform Resourse Locator) che è una striga formatta (**structured string**) in maniera molto semplice:

- prefisso (protocollo): **"http"**;
- **"//"**;
- indicazione della macchina fisica in cui è presente la risorsa con una eventuale porta in cui la macchina ascolta le richieste: **host[":"port]**;
- un path (opzionale): **[abs_path]**;

2.1 HTTP request

L'HTTP request una banale stringa che contiene una **request-line**, degli **header** e un allegato opzionale, detto **body**.
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.

2.2 HTTP response

Anche l'HTTP response è una banale stringa, che, ancora più semplicemente contiene solo: un **codice di stato** (es. 404 file not found), degli **header** facoltativi, e un allegato, anche qui chiamato **body**.
I codici di stato si classificano in base alla loro prima cifra: 1 informativi, 2 successo, 3 reindirizzamento, 4 errore nel client, 5 errore nel server.

Gli header sono informazioni opzionali a cura del browser, trasmesse come fossero parametri che aggiungono informazioni alla request o alla response.

La conseguenza di un protocollo così semplice è che con un solo identico client si può interagire con tanti back end differenti. La complessità si sposta però nell'application server, a carico del programmatore.

3 CGI

CGI sta per **Common Gateway Interface** ed è una tecnologia ormai deprecata e superata.

CGI nasce a fronte del problema di creare applicazioni web dinamiche e personalizzate in base alle richieste inviate dal client. Il problema che si pone di risolvere era l'impossibilità di comunicare i parametri HTTP presenti nella request al programma (che "gira" su un altro thread) che si occupa della creazione della response (del file HTML personalizzato).

CGI è, quindi, un'interfaccia che permette al web server di eseguire applicazioni esterne che creino pagine dinamiche. Questa interfaccia è implementata tramite un certo numero di **variabili d'ambiente** (condivisibili fra più processi) standardizzate, nelle quali il web server può salvare le informazioni di un richiesta HTTP in modo che siano in una zona di memoria alla quale un processo di un applicazione esterna può accedere.

Vediamo il tipico work-flow di un applicazione che sfrutta CGI: innanzitutto l'URL della richiesta HTTP presenta un path che porta a un applicazione **eseguibile** (.exe), il web server intercetta la richiesta e smonta (fa il parsing) la richeista, ne preleva le informazioni (i parametri) e le salva nelle variabili d'ambiente, successivamente invoca l'eseguibile. L'application server può ora prelevare le informazioni necessarie dalle variabili d'ambiente e costruire un file HTML appropriato, che 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 mantenga una **sessione** attiva. Inoltre le prestazioni di CGI sono molto scarse, 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

4 Servlet

Nasce l'esigenza di generare **contenuti dinamici** per le applicazioni web, inizialmente, infatti, HTTP era concepito come protocollo per scambio di documenti statici.

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 un modo semplice e che usa le variabili d'ambiente per adempiere al suo compito. Ma CGI ha diversi problemi, perciò è necessario trovare una nuova soluzione.

Vediamo ora il meccanismo più popolare fino a qualche anno fa per la produzione dinamica di contenuti.

HTTP è nato per essere semplice, ha solo due metodi, GET e POST e 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 parametrici, cioè si possono aggiungere informazioni sotto forma di parametri stringa: per il metodo GET i parametri sono inseriti nella query-string, per il metodo POST si mettono nel body.
HTML ha un costrutto **form**, che viene spesso associato al metodo POST di HTTP. Una form serve al browser per costruire un richeista dove i parametri sono complicati, per esempio richieste dove un parametro è un file.

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 java**, egli dovrà lavorare all'interno di un **Framework** (ooo), cioè una soluzione parziale a una serie di problemi con caratteristiche comuni. Il framework è uno schema, una soluzione parziale che omette la parte variabile di una serie di applicazioni con qualcosa in comune. Un famework può anche essere definito come una serie di utils, o come una libreria, che facilita il lavoro al programmatore.

Java Servlet è appunto un framework, tutte le applicazioni web hanno in comune il protocollo HTTP. Dal framework ci aspettiamo quindi di non dover, per esempio, 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 automaticamente, senza che ce ne dobbiamo preoccupare.

(ooo) La gerarchia dunque è che a monte di tutto c'è la JVM, nella quale gira il web server, all'interno del quale risiede il servlet container che gestisce i Servlet che il programmatore scrive.

Un primo beneficio 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 è possibile eseguire più di una richiesta senza dover essere terminato.
Inoltre 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 . 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.