TECNOLOGIE INFORMATICHE PER IL WEB

Federico Mainetti Gambera

20 settembre 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.

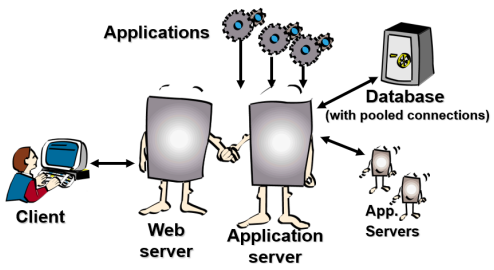
In un'applicazione web il Client interagisce con il Middle-tier con un preciso protocollo, che non è il classico TCP-IP, ma, nel nostro caso, è il protocollo HTTP.

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

4.1 Concetti base

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 componente dell'application server ed è un ambiente che esegue il tuo programma, nel nostro caso è **Tomcat**. Il container materializza l'API del framework che stiamo usando e interagisce con le Java servlet, per esempio il metodo doGet() di cui noi facciamo l'override viene chiamato dal Servlet container automaticamente, senza che ce ne dobbiamo preoccupare. Il servlet container è quindi un mediatore fra HTTP server e le Servlet, è responsabile del ciclo di vita delle Servlet e si occupa della mappatura delle richieste HTTP alla specifica servlet.

(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 **stateful**.

C'è un prezzo da pagare per questo beneficio, ovvero che non siamo noi a gestire, per esempio, la concorrenza e quindi i thread, il modello di concorrenza lo ereditiamo dal contenitore.

Il modello da seguire è chiamato "**one thread per request**" (ooo) e significa che una volta sviluppata una 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 sarà una sola istanza, un solo oggetto, tutte le request che riceviamo (anche tante nello stesso istante) avranno un thread allocato, 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 useremo tendenzialmente javax.servlet e javax.servlet.http, che sono interfacce implementate dal contenitore.

Analiziamo il ciclo di vita delle servlet.
Il contenitore mappa automaticamente le request sui metodi doGet() e doPost().
Il metodo init() viene chiamato quando la servlet inizia. 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 alla corretta 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 bisogna 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 (per esempio delle immagini).

Il file web.xml contiene la configurazione dell'applicazione, fra cui la mappatura delle varie servlet.

LEZIONE 3 18/03/20
link clicca qui

4.2 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, il mascheramento 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.

4.2.1 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.

4.2.2 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.
Queste due tecniche sono trasparenti per noi programmatori, noi otterremo un oggetto che maschererà i meccanismi che stanno dietro alla gestione delle sessioni.

4.3 Forme di identificazione

Il protocollo HTTP nella sua versione più base (senza i metadati aggiuntivi) serve richieste anonime.

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 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 effettui il login.
- **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 come user e 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 dell'identità, ma anche sulla base del ruolo.

4.4 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), e chiede al client di impostare un label specifico tramite i cookie. Nel momento in cui avviene una richiesta successiva ci accorgiamo che fra i request header è presente, nella voce cookie, il label che il server ha generato e assegnato al client durante la prima interazione.

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.

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 psaeudo-identificazione, sia una identificazione-autenticazione tramite un login: il 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 un 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. L'oggetto Session è nel server container e totalmente all'oscuro del browser, all'intenro si possono depositare informazioni legate a una pseudoidentità (come per esempio il fatto che il client si sia identificato e autenticato).

Tipiche domande: 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.

4.5 Filtri

La verifica evento epr evento dei diritti di accesso è una funzionalità obligatoria dei siti multiruolo.

La verifica viene fatta in ogni controllore e per evitare di duplicare codice si utilizzano i filtri.

Un filtro è una classe Java che il servler container interpone fra la richiesta (l'HTTP request) e il servizio (la servlet). I filtri possono essere anche concatenati uno dopo l'altro.

La classe filtro ha sostanzialmente la stessa struttura di una servlet, ha init(), destroy() e invece di doGet e doPost, ha doFilter(). Ci sono tre interfaccie che vengono usate per i filtri e sono: Filter, FilterChain e FIlterConfig.

La catena di filtri si può specificare nel web.xml, inoltre la catena di filtri è unica per ogni app, non si possono specificare catene particolari per ogni servlet. Il comando chain.doFilter() in poche parole passa il comando al prossimo filtro della catena, se ce ne è uno, altrimenti prosegue passando la richiesta alla servlet che se ne deve occupare.

LEZIONE 4 19/03/20
link clicca qui

4.6 JDBC

Gli argomenti di questa lezione sono strettamente legati 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.
La principale caratteristica di JDBC è 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 in un modo e altre in un altro.

L'architettura JDBC struttura la connessione con la base di dati in maniera intelligente e consente all'applicazione di ignorare lo specifico 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 quale sia la base di dati con cui 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 segue il seguente work-flow/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.

Nel momento in cui il programmatore non ha controllo sul ciclo di vita della propria applicazione (come nel nostro caso delle 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, 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).

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).

C'è un grande problema che richiede una certa attenzione. 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 risorse. 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 statement può rappresentare un vantaggio molto ingente.

"Mai mai mai fare query al volo, è sempre buona pratica prepararle prima".

Le **SQL injection** sono un'altro problema critico delle interazioni con le basi di dati e 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 è sempre vero. Il succo del discorso è che dobbiamo evitare di dare all'utente la possibilità di poter modificare l'albero sintattico della query (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, nel caso in cui fossero mal formati, producono semplicemente un errore, invece di rilevare informazioni che non dovrebbe.

LEZIONE 5 25/03/20
link clicca qui

5 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.
`https://javarevisited.blogspot.com/2011/09/sendredirect-forward-jsp-servlet.html`

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

6 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

7 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

8 Esercitazione

8.1 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.

8.2 Esercizio post management

Lo scopo di questo secondo esercizio è quello di mostrare un'esercizio in cui la specifica è meno chiara (esposta in maniera meno facile) e la cui implementazione è scritta "meno bene" di quella prima.

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

LEZIONE 9 15/04/20 **link** clicca qui

8.3 esercizio da un tema d'esame (pt.1)

LEZIONE 10 16/04/20 **link** clicca qui

8.4 esercizio da un tema d'esame (pt.2)

LEZIONE 11 22/04/20 **link** clicca qui

9 CSS

CSS serve per separare la presentazione dalla struttura dell'interfaccia.

Uno style sheet è un set di regole che specificano la presentazione, o, meglio, lo stile, da applicare a vari elementi. Le regole CSS sono composte da un selettore e da una lista di dichiarazioni.

Le regole CSS si possono scrivere in vari "posti":

- in un file esterno (utilizzo prevalente). Questo metodo separa fisicamente lo stile dal contenuto, ci sono due approcci: o si usa o il tag link di HTML (consigliato), o la clausola @import di CSS (sconsigliato) all'interno di un tag style di HTML. In caso di conflitto di regole, per il browser "vince" l'ultima regola "letta", dunque bisogna sempre tenere in mente l'ordine di scrittura.
- nell'attributo style di un elemento HTML (inline styling, sconsigliato). L'inline styling ha prevalenza sullo styling da file esterno, nel senso che in caso di conflitto vince sempre. Inoltre, con javascript, quando modifichiamo lo stile di un elemento, stiamo accedendo e modificando l'attributo style di quest'ultimo, come fosse inline styling. Ci sono modi per evitare di usare inline styling e agire in maniera più indiretta sulla presentazione con javascript, per esempio modificando la classe dell'elemento (consigliato), invece dell'attributo style.

9.1 Selettori

I selettori sono delle query, delle interrogazioni, un modo per filtrare un documento ed estrarne delle porzioni.

Ci sono quattro grandi categorie di selettori:

- Selettori basati sugli elemento;
- Selettori basati sugli attributi;
- Selettori basati sulle pseudo-classi/pseudo-elementi, cioè selettori che ci permettono di utilizzare informazioni che vanno oltre ciò che è esprimibile con HTML, per esempio la ricerca del primo o del secondo figlio di un elemento (in html non c'è nozione di primo o secondo), oppure la selezione degli elementi che l'utente ha già visitato (qui addirittura viene fatta una selezione tramite informazioni dinamiche);
- Selettori basati sui combinatori, cioè la combinazione delle precedenti categorie tramite operatori, per esempio "¿" è l'operatore figlio. In CSS ci sono più di 40 diversi combinatori (il prof ha caricato un documento con 40 diversi selettori, dagli un'occhiata).

9.2 Rendering

Parliamo ora al processo di renderizzazione del browser.
Le fasi in ordine sono le seguenti:

- Declaration collection: fase di recupero di tutte le regole attive e valide (regole che appartengono a un ostyle sheet, regole il cui selettore selezionano un elemento, regole sintatticamente corrette).
- Cascading: è il processo di analisi di tutte le regole rilevanti per la proprietà di un determinato elemento della pagina, mette in ordine le dichiarazioni a seconda della loro precedenza e sceglie sulla base di un criterio di precedenza quale sia il valore da applicare.
  Il cascading applica un criterio di precedenza fatto di quattro termini:
  - origine e importanza (poco usato): per origine si intende chi è l'autore della regola (notiamo che anche il browser o user agent, ha un foglio di stile, oppure il lettore può crearlo, esistono strumenti particolari che i client possono usare), per importanza invece si intende che alcune regole possono essere marcate come importanti (poco usate). L'ordine considerato è il seguente: regola improtante ¿ regola del foglio di stile del lettore ¿ regola scritta da noi.
  - scope (poco applicato);

  - specificità (molto usato): il criterio di specificità indica che in caso di conflitto, vince il selettore più specifico, per capire il livello di specificità di un selettore si contano gli operatori utilizzati e gli si attribuisce un peso, per esempio il selettore per id (tipo a) è molto più importante di quello per classe (tipo b), che è molto più importante di quello per tag (tipo c);
  - ordine di apparizione (molto usato): l'ordine di apparizione è il criterio che fa si che l'inline styling sia sempre prevalente;

- Defaulting: nel caso in cui non venga specificato nessun valore per una determinata proprietà, viene scaturito il defaulting, cioè viene applicato un criterio di eredità (inheriting, ma che si esprime meglio in italiano come criterio di contenimento). Un elemento senza un valore per una proprietà, deriva quella proprietà da chi lo contiene, se pure chi lo contiene non ha tale proprietà, allora viene derivata dal contenitore ancora più a monte, e così via. Se si arriva fino al body e tale proprietà non è ancora stata definita si assume un valore iniziale (initial value). Da notare che non tutte le proprietà CSS vengono tramandate per ereditarietà, per esempio "background-color".
- Dependency resolution: vengono calcolati i valori computati dai valori relativi (es. percentuali).
- Formatting: vengono calcolati i valori usati tramite le informaizoni di layout (es. auto).
- Rendering: il rendering sfrutta i valori effettivi.

LEZIONE 12 23/04/20 **link** clicca qui

10 Rich Internet Application

10.1 Client side scripting

Lo scripting a lato client è una programmazione focalizzata alla gestione degli eventi prodotti dall'interazione dell'utente.

Nella versione "thin client" o "pure HTML" il client non fa praticamente nulla: semplicemente invia richeiste, riceve risposte e le mostra a video. Ad ogni evento dell'utente viene ricaricato l'intera interfaccia, si ha una brutta user experience.

L'architettura "fat client" rende anche il browser un contenitore di funzioni applicative.

10.1.1 eventi

Nell'architettura "fat client" gli eventi diventano di tre tipi:

- trattati solo dal cliente: si cambia l'interfaccia senza senza interpellare il server;
- trattati solo dal server: tramite una richiesta HTTP si interpella il server, la cui response sarà la nuova interfaccia, questi sono gli eventi classici dell'architettura "pure HTML";
- trattati dal cliente e dal server: il client invia una richiesta HTTP al server, la cui risposta viene usata per modificare parzialmente l'interfaccia.

Eventi "ibridi" gestiti da client e server
Gli eventi gestiti da client e server permettono di separare l'interazione tra utente e browser da quella tra browser e server (interazione asincrona), tipicamente per:

- caricare contenuti dinamicamente;
- refreshare dinamicamente contenuti di un interfaccia;
- inviare al server un comando senza modificare l'interfaccia utente.

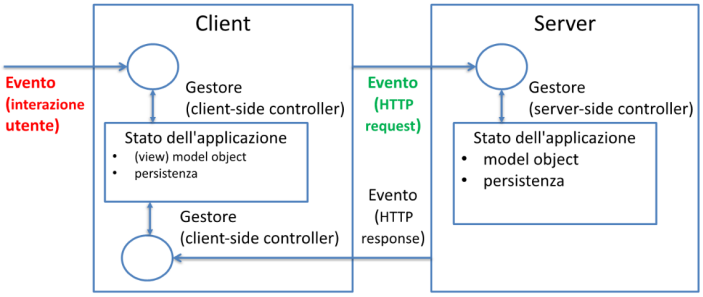
La risposta HTTP può produrre l'invio di codice HTML per sostituire una porzione dell'interfaccia, oppure di dati per aggiornare il contenuto dell'interfaccia. E' solitamente preferibile inviare i dati in modo da poter eventualmente usufruire della funzioen anche da applicazioni che non utuilizzano HTML per renderizzare l'interfaccia utente, ma non è sempre così perchè alcune volte calcolare il più possibile a lato server può portare a dei vantaggi.

In una tipica architettura "pure HTML" si ha un'interazione **sincrona**. A fronte di un evento che interpella il server, il browser prende il controllo e invia una richeista HTTP e si pone in attesa di una risposta HTTP. In questo periodo di attesa l'utente non può far nulla se non aspettare che la richiesta venga soddisfatta. Quando la riposta arriva il browser ne fa il parsing e ne restituisce un rendering visivo, a questo punto il browser cede nuovamente il controllo all'utente.

Nell'architettura "fat client" si usano interazioni **asincrone**. Mentre il browser invia una richiesta HTTP e attende una risposta HTTP, permette comunque all'utente di continuare la sua interazione. Viene utilizzato il concetto di **callback**, o chiamata di ritorno, cioè una funzione da chiamare quando un flusso asincrono è giunto termine.

10.1.2 Architettura fat client

Nell'architettura "fat client" viene duplicata la stessa architettura a lato server anche a lato a client.



Elementi di un'architettura fat client

- Eventi;
- Controllore: associa funzion ad eventi;
- Stato: stato della presentazione e stato dell'applicazione client.

Sorgono quindi nuovi quesiti su come progettare un'applicazione fat client:

- quante view servono?
- quali eventi ci sono? dove vengono trattati (client, server, client-server);
- dove (in html o javascript) rappresentare lo stato dell'applicazione? dividiamo in stato dell'applicazione e stato dell'interfaccia o farò una cosa sola?
- etc...

Notiamo quindi che ci sono tantissimi modi diversi per progettare un fat client, vediamone due estremi opposti:

- Soluzione 1: applicazione a paggine separate e interazioni asincrone per il rinfresco parziale della vista. Le funzionalità principali sono allocate in pagine diverse, ma vengono usati eventi solo a lato client per gestire eventi che non modificano il contenuto, eventi ibridi server e client vengono usati per gestire eventi che cambiano il contenuto ma non le funzionalità, e eventi che cambiano le funionlità vengono trattati da eventi solo a lato server e il conseguente refresh della pagina.
- Soluzione 2: applicazione a pagina singola. Tutte le funzionalità sono realizzate in un unica pagina e tutti gli eventi vengono gestiti almeno a lato client
- esistono molte altre soluzioni intermedie a queste due.

LEZIONE 13 30/04/20 **link** clicca qui

10.2 Javascript

Javascript è un linguaggio interpretato, quindi non ha una fase di compilazione.

In Javascript è assente la tipizzazione delle variabili, il principale aspetto negativo di un linguaggio di programmazione a tipizzazione debole è che spesso, siccome non c'è un compilatore, gli errori passano inosservati finchè non si esegue il programma.
Inoltre pure la dichiarazione di una variabile non è obbligatoria, una variabile non dichiarata viene automaticamente identificata come una variabile globale (scope globale), cosa da evitare assolutamente. L'ambiente di sviluppo è estremamente liberale.

Gli script javascript agiscono su un documento HTML, il cuore della programmazione a lato client consiste nell'interagire con un'interfaccia.

10.2.1 Numeri

Javascript ha un solo tipo di numero (64-bit floating point).

Esistono inoltre valori speciali: NaN (not a number), col relativo metodo isNaN(), e infinity o -infinity, col relativo metodo isFinite().

10.2.2 Stringhe

Le stringhe possono essere racchiuse dai simboli " e ' indifferentemente: non esiste il tipo char.

Le stringhe sono immutabili, cioè, una volta create, non possono essere modificate.

10.2.3 Istruzioni

Le istruzioni sono eseguite in ordine di apparizione.

Un blocco è un insieme di istruzioni racchiuse da parentesi graffe, che in javascript non rappresentano un ambito di scope. Gli unici scope che esistono sono dati dalle funzioni e dal "main" (scope globale). Questo concetto è importante per la programmazione asincrona.

Scope di livello intermedio possono essere definiti con clausole speciali per la dichiarazione di variabili (let invece di var).

Il temrinatore di istruzione ";" è opzionale.

`http://www.jslint.com/` è un ottimo sito per scovare cattive pratiche all'interno di codice javascript.

10.2.4 Uguaglianza

Gli operatori di uguaglianza sono "===" e "!==", cioè se due oggetti sono dello stesso tipo ed hanno lo stesso valore, allora "===" ritorna true.

L'operatore "==" (meno stringente di "===") prova a comparare oggetti di tipo differenti.

In javascript la nozione di tipo è diversa da quella a cui siamo abituati per i linguaggi fortmente tipati, ovvero l'operatore typeof, che mostra il tipo di una variabile può ritornare solo uno di questi valori: number, string, boolean, undefined, function e object. Da notare è che typeoff null ritorna object.

10.2.5 Oggetti

La nozione di oggetto in javascript è diversa da quella per un linguaggio a oggetti. Per esempio in Java una classe definisce un tipo di dato e permette la creazione di oggetti. In javascript tutto è un oggetto, gli oggetti sono contenitori utilizzati per organizzare dati e possono essere definiti "al volo", è però fortemente consigliato usare funzioni costruttrici che sfruttano il riferimento "this". Esempio: "function Person(name, surname)this.name = name; this.surname = surname;" e poi "var person = new Person();". Ci sono poi vari modi per ottenere informazioni ulteriori su un oggetto in javascript: abbiamo visto che "typeof" ci ritorna solo che è un "object", invece ".constructor" ci ritorna il costruttore che abbiamo usato per creare tale oggetto.

Le proprietà degli oggetti possono essere eliminate con il comando "delete".

E' possibile accedere alle proprietà di un oggetto oltre che con la notazione punto, anche con la notazione a parentesi quadre: oggetto.proprietà, oppure oggetto["proprietà"]. Questo metodo è utile perchè all'interno delle quadre c'è una stringa e di conseguenza si possono usare variabili oppure ciclare fra le proprietà, etc...

Gli oggetti sono sempre passati per riferimento, non vengono mai copiati (!).

10.2.6 Funzioni

Uno dei tipi principali di Javascript sono le funzioni. Oltre ad essere usate nella maniera classica, esse sono dei veri e propri valori e di conseguenza possono essere assegnate a variabili. I metodi degli oggetti sono variabili che contengono funzioni.

10.2.7 Prototipi

In assenza di classi nel senso proprio del termine è possibile definire proprietà che si applicano a gruppi di oggetti con il concetto di prototipo (prototype object). Ogni oggetto è associato a un prototipo dal quale può ereditare proprietà, per default ogni oggetto è associato a Object.prototype, un oggetto standard di javascript. Il prototipo è dinamico, se si aggiunge una nuova proprietà al prototipo, essa sarà visibile in tutti gli oggetti basati su quel prototipo.
Esempio:

```
var property = {value : 1};
function MyObject() {};
MyObject.prototype = property;
var myObject = new MyObject();
myObject.value === 1 //true
```

La ricerca della proprietà di un oggetto parte dalla definizione delle proprietà locali possedute direttamente e risale la catena dei prototipi fino a Object.prototype.

10.2.8 Undefined

Undefined è uno dei tipi fondamentali di javascript. Il tentativo di computare con valori undefined genera un'eccezione TypeError exeption.

10.2.9 Vettori in JavaScript

I vettori in JavaScript non sono i classici array che conosciamo. Esiste un costruttore predefinito Array che crea oggetti che si comportano come array e la notazione con le parentesi quadrate equivale all'utilizzo di questo costruttore.

Per tutto il resto gli array in javascript sono veri e propri oggetti, di conseguenza possiamo farci tutto quello che si può fare con un oggetto. In più possiede la funzione lenght (calcolato come 1 + il numero della proprietà di massimo valore nel vettore, non è il numero degli elementi), la funzione splice e degli indici con cui è possibile accedere alle proprietà. Notiamo che si possono eliminare alcune proprietà lasciano dei veri e propri buchi all'interno.

LEZIONE 14 06/05/20 **link** clicca qui

10.3 Funzioni in JavaScript

Le funzioni in javascrip sono oggetti, perciò possono essere memorizzate in variabili, passate come argomento ad altre funzioni o ritornate come risultato di altre funzioni. Le funzioni possono avere proprietà e metodi, come un oggetto qualsiasi.

In javascript si usano molto spesso le funzioni anonime, il nome della funzione è superfluo. Solitamente le funzioni vengono salvate all'interno di variabili, che quindi conterranno il riferimento all'indirizzo di memeoria della funzione.
Esempio:
var add = function sum(a,b){ return a +b; };
Osserviamo che in questo caso il nome della funzione "sum" non viene neanche registrato nella tabella dei simboli del programma, quindi si può ignorare e riscrivere come:
var add = function (a,b){ return a +b; };
Il nome "sum" è quindi sconosciuto al di fuori del body della funzione, può quindi per esempio essere usato per scrivere funzioni ricorsive.

Esistono comunque le funzioni classiche il cui nome è essenziale e che hanno bisogno della definizione della funzion.
Esempio:

```
function sum(a,b)\{ return a+b;\}
```

Una funzione viene chiamata se si aggiungono le parentesi tonde alla fine "()", altrimenti, se si omettono, la funzione non viene chiamata e ci si sta usando il riferimento all'indirizzo della funzione.

Omettendo le parentesi quindi è possibile passare funzioni come parametri di altre funzioni, o avere come valori di ritorno una funzione, o assengare a una variabile una funzione.

Se una funzione non ha ritorna un valore specifico (non ha definito la clausola return), JavaScript assume automaticamente il valore undefined come valore di ritorno.
Esempio:

```
funtion a()\{
    console.log("A");
\}
function b()\{
    console.log("B");
    return a(); //a() non ha un valore di ritorno e quindi b() ritorna undefined
\}
```

Le funzioni sono l'unico elemento (oltre allo scope globale) che rappresenta uno scope per le variabili all'interno di JavaScript. Una variabile dichiarata all'interno di una funzione, esiste solo all'interno di quest'ultima.
Una delle utilità delle funzioni anonime è proprio quella di vare da delimitatore per le variabili.

Gli argomenti di una funzione sono gli i parametrti che il chiamante gli passa, inoltre, se la funzione è stata chiamata come proprietà di un oggetto, allora è presente il parametro aggiuntivo "this", che rappresenta un riferimento all'oggetto chiamante.
I parametri passati alla funzione possono anche non coincidere con la definizione della funzione, cioè una funzione può essere chiamata con un numero arbitrario di parametri, nel caso in cui fossero di meno, vengono assunti undefined quelli mancanti (se fossero di più c'è modo per accederci all'interno della funzione ...).

Le funzioni possono essere definite all'interno di altre funzioni, una funzione interna può accedere ai parametri e alle variabili locali della funzione in cui è innestata.
Si chiama chiusura (closure) una funzioen innestata che può accedere all'ambito di visibilità (scope) della funzione padre, anche quando quest'ultima ha terminato l'esecuzione (per esempio nel caso di funzioni asincrone).
Esempio di chiusura:

```
function esterna()\{
    var a = 2;
    function interna()\{
        console.log(a);
    }
    return interna;
\}
var prova = esterna();
prova(); //esterna finisce, ma prova() vede ancora a, l'interna riesce a sopravvive
```

LEZIONE 15 07/05/20 **link** clicca qui

Function statement: function f(){...;}; definisce una funzione; la funzione si invoca con l'operatore di chiamata (); deve avere un nome; la funzione è soggetta a "hoisting" per cui è disponibile anche prima di dove è stata definita nel programma.

Function expression: var f = function (){...;};; definisce la variabile f che ha come valore una funzione; non necessita di un nome, ma nel caso in cui venga specificato, il nome è utilizzabile solo all'interno del body della stessa; la funzione è invocata tramite il nome della variabile; la funzione è definita solo quando avviene la valutazione dell'espressione, cioè finchè la riga di codice contenente la variabile f non viene eseguita, la funzione non esiste; la funzione può vedere le variabili dello scope in cui è definita (chiusura).

Per creare variabili non globali in JavaScript bisogna creare uno scope tramite le funzioni. Mostriamo un esempio:

```
var scope = ( function(){
```

```
    var variabileNonGlobale = 0;
    return function(){
        //Possiamo usare la variabile non globale qua per via della chiusura.
    }
})();
```

Questa è una cosiddetta "iife", cioè immediately invoked function expression.
Una iife è un'espressione che comporta la dichiarazione e l'esecuzione di una funzione e sono utilizzate per delimitare una porzione di codice evitando che le variabili finiscano nello scope globale.
La presenza delle parentesi tonde esterne (grouping operator) rendono la frase un'espressione valida, altrimenti sarebbe un function statement e non un function expression.
Le iife possono essere usate anche per realizzare un meccanismo di protezione delle variabili, cioè si riesce a imitare il comportamento dei getter e setter di attributi privati:

```
var Sequence = (function sequenceIIFE(){
    var current = 0; //variabile privata
    return { //ritorna un oggetto
        getCurrentValue: function(){
            return current; //rimene vivo per chiusura
        },
        getNextValue: function(){
            current = current +1;
            return current; //rimane vivo per chiusura
        }
    };
})();
```

In questo caso la variabile current non è accessibile se non attraverso i metodi Sequence.getCurrentValue() e Sequence.getNextValue().

Le funzioni possono essere assegnate a proprietà di oggetti (metodi di un oggetto), il parametro this denota l'oggetto su cui la funzione è chiamata, ma un metodo non può usare una funzione innestata che operi direttamente sull'oggetto this, cioè la funzione innestata non condivide il riferimento a this. per permetterle di accedere a this bisogna salvarne un riferimento in una variabile (var self = this;) che sarà visibile nella funzione innestata.

Invocazione indiretta di funzione: i metodi call e apply di un oggetto di tipo funzione permettono di invocare la funzione, passandogli un valore da usare come this e nel caso di call una lista di argomenti, nel caso di apply un vettore di argomenti.

L'oggetto arguments è disponibile all'interno delle funzioni quando vengono invocate e contiene i valori dei parametri, include anche i valori opzionali ed è utile per la definizione di funzioni con un numero non specificato di parametri. Notare che arguments non è un vero e proprio vettore, ha la proprietà lenght, ma manca di tutti gli altri metodi tipici dei vettore, si dice essere un oggetto "array-like" o "pseudo-array".

Una funzione restituisce sempre un valore. Casi speciali:

- undefined: se il valore di ritorno non è specificato;
- this: se la funzione è un costruttore invocato con l'operatore new.

Arrow function: le arrow function sono una notazioen abbreviata per la scrittura di espressioni funzionali. Le uniche differenze sono:

- non possiede l'oggetto arguments;
- si può usare this, che eredità il valore dalla più vicina funzione normale che la racchiude;
- non può essere usata come costruttore con la parola new.

```
var add = function(x,y){
    return x + y;
};
//equivale a:
var add = (x,y) => {
    return x + y;
};
```

Una funzione di callback è un meccanismo usato per gestire eventi asincroni che causano la chiamata di una funzione:

- la funzione da chiamare è passata come parametro alla funzione che gestisce l'evento;
- la funzione passata come parametro è invocata quando l'evento si manifesta.

Un tipico esempio sono le richieste HTTP asincrone al server.

LEZIONE 16 13/05/20 **link** clicca qui

10.4 JavaScript e il DOM

Si può utilizzare JavaScript per interagire con l'user interface agendo sul documento HTML.

I metodi di interazione sono standardizzati dal DOM (document object model) e dall'HTML APIs:

- DOM API: permettono a un programma di accedere e modificare il contenuto, la struttura, il comportamento e lo stile di documenti HTML e XML.
- HTML5 API: permettono di accedere a funzionalità dell'ambiente (il browser) che ospita il documento, come la finestra, la cronologia, le notifiche, la memoria, la connessione, etc.

Il DOM definisce una struttura logica del documento e i metodi per accederci e manipolarlo.
La rappresentazione dei documenti è presentata come una gerarchia di nodi (oggetti JavaScript).
Il DOM è strutturato in diversi moduli che contengono funzionalità basiche e specializzate, le più improtanti e di nostro interesse sono i moduli: core, HTML, Events.

le HTML APIs, in breve, ci permettono di riscrivere il comportamento del browser.
L'oggetto più importante è l'oggetto window, che è globale, e grazie al quale un programma JavaScript può interagire con l'ambiente (il browser). L'oggetto window ci espone tutte le funzionalità necessarie per interagire con il browser, una delle proprietà più usate è la proprietà window.document, che è il punto di ingresso del DOM, cioè è la radice più alta di tutto il documento HTML.

Quindi, riassumendo, JavaScript può interagire con l'ambiente, cioè il browser, attraverso HTML APIs. Come? L'oggetto globale window ci offre tutti le proprietà necessarie per farlo, fra cui la proprietà window.document, che rappresenta l'ingresso al DOM. Quindi DOM e HTML APIs sono strettamente collegate. L'oggeto document si accede tramite le DOM API, che mostrano funzioni che permettono l'accesso ai vari elementi. GLi element object hanno a loro volta proprietà che rappresentano il loro contenuto, gli attributi, le classi, lo stile, etc.

Window, document, element objects sono associati a eventi che possono essere monitorati e di conseguenza provocare la chiamata a funzioni che le gestiscono (handler function).

Per evitare ritardi di caricamento della pagina, è consigliabile mettere il tag script nell'head e usare il metodo defer per fare il caricamento in parallelo.

Il DOM core permette di accedere agli attributi e ai metodi degli elementi che rappresentano il documento.
In partocolare permette di:

- cercare e accedere a elementi;
- attraversare il document tree;

- cercare e settare attributi di elementi;
- cercare, settare e modificare il contenuto di elementi;
- creare, inserire, eliminare, spostare elementi nel document tree;
- lavorare con le form.

DOM events specifica un sistema di eventi, che permette la registrazione di event handler, descrive i metodi di propagazione di eventi attraverso la struttura del documento, offre informazioni sugli eventi. Permette anche la creazione di eventi custom (ancora in corso di standardizzazione).

Uno degli oggetti più importanti è il Node object, che poi si specializza in document, character data (contenunto di un tag), HTML element (tag), attr.

Le tipiche funzioni DOM ritornano spesso degli array-like object (NodeList), cioè oggetti che hanno una struttura uguale a quella di un array, solo che non hanno la funzione lenght gestita automaticamente e hanno una funzione splice. Notiamo che possono essere trasformati in array grazie alla funzione Array.from(...). Stare attenti a manipolare le NodeList, che spesso la proprietà lenght potrebbe dare problemi.

E' sconsigliato avere un documento HTML "vuoto" e costruire dinamicamente la pagina con JavaScript, perchè si consumano risorse di calcolo.

Il contenuto di un elemento è rappresentabile in tre maniere diverse:

- innerHTML: rappresenta il contenuto come un frammento HTML e quindi richiede del parsing per lavorarci, il che è spesso inefficiente, l'unico utilizzo consigliato è per eliminare il contenuto: .innerHTML= "";
- textContent: rappresenta il contenuto come plain text;
- childNodes: è una proprietà che può essere usata per accedere alla lista dei sotto-nodi di un elemento.

La prima e la terza rappresentazione sono la stessa cosa, solo che la terza è cioè che si ottiene dopo il parsing della prima.

LEZIONE 17 14/05/20 **link** clicca qui

10.5 JavaScript e la gestione degli eventi

L'event handling è la funzionalità più importante della programmazione JavaScrip client side.

Un evento è un avvenimento che il browser segnala a JavaScript e che il DOM "oggettifica" tramite l'interfaccia Event. Tutti gli oggetti che potrebbero produrre un evento implementano l'interfaccia EventTarget. L'interfaccia Event ha la proprietà target che denota l'oggetto dal quale l'evento si è scaturito.

Il programma JavaScript deve registrare una funzione (handler function) che risponda allo scatenarsi dell'evento. Un event handler (listener) è una funzione che gestisce la risposta all'evento.

L'interfaccia Event ha una proprietà type che denota il tipo d'evento. I tipi di evento sono molto numerosi, alcuni definiti da DOM API (es. mutation event) e altri da HTML APIs (es. load).

Per registrare un evento si usa il metodo EventTarget.addEventListener(event, listener, useCapture) in cui:

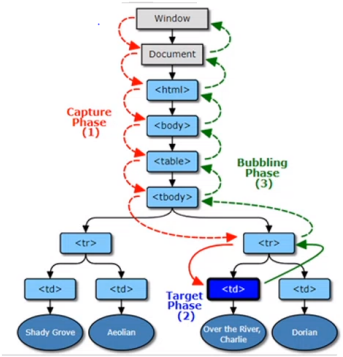
- event: tipo dell'evento;
- listener: funzione o metodo di un oggetto da chiamare quando l'evento si scatena;

- useCapture: valore booleano facoltativo (di default è false) che specifica come gestire il listener nella fase di preocessamento di bubbling/capturing;

oppure si può registrare sull'html tramite gli attributi (metodo sconsigliato).

Quando un evento si scatena, la sua funzione handler potrebbe aver bisgno di sapere quale sia l'evento e il target su cui si è scatenato. Se l'handler è una funzione, this rappresenta l'elemento target, non l'event object, se l'handler fosse stato un oggetto, this rappresenterebbe proprio l'oggetto gestore. Quindi è poco consigliato usare this, perchè in dipendenza di chi gestisce l'evento (un oggetto, una funzione), potrebbe assumere valori diversi, non è un valore costante, dipende da come è stata registrata la funzione handler. Piuttosto l'event object, che è molto utile per estrarre informazioni dall'evento, può essere acceduto tramite il primo parametro passato automaticamente dal contenitore alla funzione handler (tipicamente indicato con e) oppure tramite una variabile globale chiamata "event".

Trattiao ora la propagazione degli eventi.
La propagazione degli eventi è la sequenza di azioni che avvengono quando si scatena un evento. Siccome più evento possono essere registrati all'elemento target e ai suoi antenati, dobbiamo capire quali vengono chiamati e in che ordine.
La propagazione degli eventi avviene in tre fasi: capture, target e bubbling.
La fase di discesa dell'albero del documento si chiama capturing, la fase fase di trattazione locale si chiama treeatment, la fase di risalita si chiama bubbling.
Da qui capiamo che per avere controllo sulla fase di capturing possiamo usare il flag di capturing della funzione che registra gli eventi (addEventHandler()), se è true, significa che l'handler viene invocato anche se l'evento è in fase di capturing.
Per gestire la fase di bubbling invece si usa il metodo event.stopPropagation() che interrompe la propagazione ovunque sia arrivato. L'utilizzo di stopPropagation() non è una buona pratica, è piuttosto consigliato cercare di posizionare meglio i propri eventi.



Ci sono degli eventi che hanno una gestione standard da parte del browser, il metodo preventDefault() può essere usato per cancellare il comportamento di default associato all'evento (es. form, anchor).
A volte può essere utile simulare l'interazione di un utente, cioè programmare lo scatenamento di un evento.

LEZIONE 18 20/05/20 **link** clicca qui

10.6 AJAX: javascript e l'interazione client server asincrona

L'architettura fat client permette di separare (rendere asincorna) l'interazione dell'utente col browser da quella del browser (client) col server e di evitare il ricarico completo dell'interfaccia.

Uno script può richiedere al browser l'invio di una richiesta HTTP e la ricezione (asincron) della risposta.
XMLHttpRequest è un'interfaccia che permette ddi creare oggetti (normalmente gestiti dal browser) utili per:

- inviare richieste HTTP (o HTTPs) ad un web server;
- Restituire i dati provenienti dalla risposta del server allo script.

L'oggetto globale window contiene l'oggetto di utilità XMLHttpRequest che è, in poche parole, l'interfaccia di rete del browser.

L'oggetto XMLHttpRequest possiede uno stato, attributi e metodi e può generare eventi.
La caratteristica principale è proprio il suo stato che evolve nel tempo e che descrive l'avanzamento della richiesta HTTP. gli stati ammissibili sono:

- UNSENT = 0;
- OPENED = 1;
- HEADERS_RECEIVED = 2;
- LOADING = 3;
- DONE = 4.

Gli attributi principali sono invece:

- readyState: memorizza lo stato corrente del processamenteo della richeista; ha come valori i possibili stati elecanti prima;
- onreadystatechange: memorizza lafunzione che gestisce l'evento dicambio dello stato (event handler);
- upload: memorizza un oggetto di tipo XMLHttpRequestUpload che permette di monitorare lo stato di richieste che comprendono un body;
- status, statusText: memorizza lo status code e message della risposta HTTP;
- response, responseText, responseXML: memorizza il contenuto (body) della risposta.

I metodi principali sono:

- open(metodo, url) oppure open(metodo, url, asyncFlag, username, pwd): permettono di creare una richiesta con metodo specificato indirizzata a un URL specificato; per default la richiesta è asincrona.
- setRequestHeader(nome, valore): permette di assegnare valore a header della richiesta;
- send(body-opzionale): effettua l'invio della richiesta HTTP, opzionalmente allegando un request body;
- abort(): interrompe la richiesta.

Gli eventi principali sono:

- readystatechange: generato quando l'attributo readyState cambia valore, è l'evento più usato e più importante;
- loadstart, progress, abort, error, load, timeout, loadend: altri eventi meno usati, l'evento readystatechange permette già di gestire tutti i casi, questi eventi sono più moderni e specifici, ma non sono supportati da tutti i browser.

Inviare una richiesta asincrona non è un lavoro difficile, basta usare i metodi open e send dell'oggetto XMLHttpRequest, gestire la risposta è invece la parte più complessa solitamente.
Lo stato della richiesta può essere monitorato controllando le proprietà dell'oggetto XMLHttpRequest:

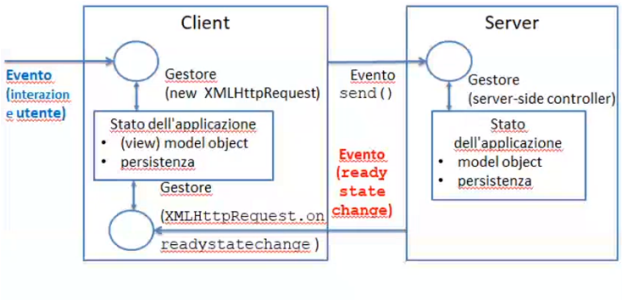
- XMLHttpRequest.readyState mantiene traccia dello stato della richiesta;
- XMLHttpRequest.onreadystatechange contiene un riferimento a una funzione che viene chiamata ogni volta che XMLHttpRequest.readyState cambia, cioè si produce l'evento readystatechange;
- XMLHttpRequest.status permette di controllare l'esito, per esempio vale 200 quando la risposta è OK oppure 404 quando la risorsa richiesta non è stata trovata.

Il contenuto della risposta del serer si trova come valore della proprietà responseText o responseXML (c'è anche JSON: studio autonomo).

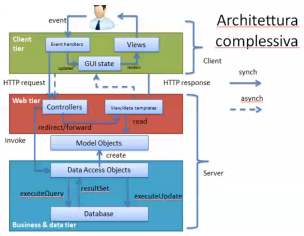
Vediammo ora la tipica funzione che effettua la chiamata:

```
function makeCall(method, url, formElement, cback, reset=true){
    var req = new XMLHttpRequest();
    req.onreadystatechange = function(){
        cback(req);
    };
    req.open(method, url);
    if(formElement == null){
        req.send();
    }
    else{
        req.send(new FormData(formElement));
    }
    if(formElement !== null && reset === true){
        formElement.reset();
    }
}
```

Vediamo ora un'immagine riassuntiva dell'architettura asincrona con XMLHttpRequest:



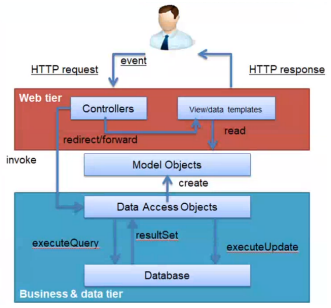
Notiamo che ora la nostra applicazione a lato client deve gestire due fonti di eventi: quelle dell'utente e quelle del server (readystatechange).
Vediamo ora un'immagine che mostra l'architettura complessiva:



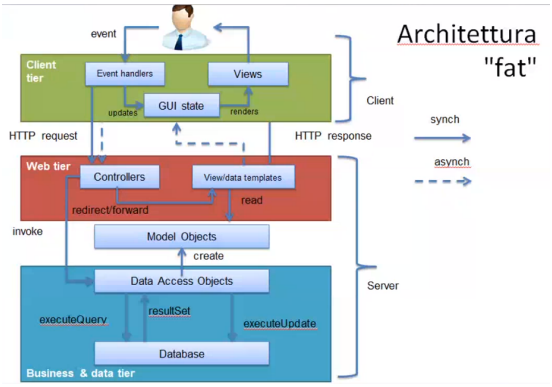
LEZIONE 19 21/05/20 **link** clicca qui

11 Patter pure HTML vs RIA

Architettura THIN:



Architettura FAT:



Differenze negli eventi:

Pure HTML:

- gli eventi sono richieste HTTP;
- gli eventi provengono dal client;
- gli eventi sono creati dal browser che gestisce per default gli elementi ancora e i buttoni di submit;
- gli eventi sono ricevuti dal web server che li inoltra al servlet container (tomcat).

RIA:

- gli eventi sono eventi di DOM e HTML API;
- gli eventi provengono dal browser (oggetto window) o dal server (response HTTP);
- gli eventi sono prodotti dall'interazione dell'utente o dall'interfaccia di rete (oggetto XMLHttpRequest);
- gli eventi sono dal browser e notificati al gestore tramite l'oggetto contenitore window.

Mappatura degli eventi:

Pure HTML:

- il risponditoe è una servlet (controller);
- l'associazione tra evento e rispondditore è configurata fuori dall'applicazione (web.xml).

RIA:

- il risponditore è una funzione JavaScript;
- l'associazione tra evento e risponditore è a carico del programmatore (addEventListener).

Risposta agli eventi:

Pure HTML:

- la risposta agli eventi è sincrona: il client si sospende e attende la response HTTP;
- la response HTTP può rimpiazzare il contenuto della pagina corrente oppure forzare l'emissione di una nuova request (redirect);
- la response HTTP, il suo stato e il suo contenuto sono gestiti dal browser;
- a lato server avvengono sia la trattazione della richiesta sia la selezione della prossima vista.

RIA:

- eventi di interazione: la funzione che risponde può operare solo a lato client oppure creare una richiesta al server, tipicamente asincrona;
- eventi di callback: la response HTTP, il suo stato e il suo contenuto sono gestiti dalla funzione che tratta l'evento.
- a lato server avviene la trattazione della richiesta;
- a lato client avviene la selezione della prossima vista: servlet manda response code e dati, la callback decide la prossima vista.

Cambio della vista:

Pure HTML:

- decisa a lato server;
- resonse.getWriter.println, sendError, sendRedirect forzano il cambio integrale del contenuto della vista corrente o la richiesta di una nuova vista;

RIA:

- decisa a lato client;
- la proprietà window.location.href permette la sostituzione della vista corrente;

Submit e risultato:

Pure HTML:

- per default la history del browser registra le richieste emesse (URL, POST data. . .);
- dopo l'invocazione con metodo POST (passo 1) bisogna fare una nuova richiesta (passo 2) per una nuova pagina che mostri il risultato;
- altrimenti il rinfresco della pagina iniziale del processo rimanda la richiesta POST.

RIA:

- la pagina formula una richiesta POST asincrona;
- la richiesta asincrona non cambia window.location.href;
- l'eventuale rinfresco della pagina produce la visualizzazione della stessa pagina;
- il risultato della richiesta POST è visualizzato nella stessa pagina dalla funzione di callbacl deòòa richiesta asincrona;
- NB: per ragioni storiche la servlet che gestisca la chiamata asincrona deve essere annotata con @MultipartConfig.

Presentazione del contenuto dinamico:

Pure HTML:

- il controller orchestra l'estrazione del contenuto dinamico, aggiorna il modello e cede il controllo alla view;
- la view inserisce il contenuto nel template;
- il template attualizzato con il contenuto dinamico viene inviato come response al client;
- il browser usa la response per aggiornare l'intera pagina.

RIA:

- il controller orchestra l'estrazione del contenuto dinamico, lo formatta (in JSON) e lo invia come response;
- la funzioen di callback del gestore dell'evento attualizza il contenuto della pagina con il contenuto della response.

Passaggio parametri e stato dell'ultima interazione:

Pure HTML:

- quando un evento produce parametri questi sono comunicati al componente che li consuma mediante la request HTTP, anche se servono all'interno della stessa pagina;
- è un metodo di preservazione dello stato dell'applicazione di breve termine (dell'ultima interazione), per esempio per preservare la selezione di un elemento da un elenco che aggiorna il contenuto della stessa o di un'altra pagina;
- per lo stato di lungo termine (più di un'interazione) serve la sessione o il database.

RIA:

- quanddo un evento produce parametri che servono all'interno della stessa pagina questi sono preservati come parte dello stato della pagina mediante variabili dei componenti o parametri delle loro funzioni, per esempio la selezione di un elemento da un elenco che aggiorna il contenuto della stessa pagina.

Mantenimento dello stato:

Pure HTML:

- quando deve essere preservata informazione per più di un'interazione si usa la sessione del server;
- il controller può creare, vericare e terminare la sessione;
- il client può solo trasmettere il proprio session ID.

RIA:

- quando deve essere preservata informazione per più d un'interazione si usa la memoria persistente del client (localStorage e sessionStorage);
- il client può creare, verificare e terminare la sessione;
- il server può ricevere il contenuto dello stato se necessario, per esempio creare un ordine persistente dai prodotti temporanemente nel carrello.

N.B. si trovano esempi completi di ognuno di questi concetti nelle slide del prof di questa lezione.