Applicazioni Web e Mobile

Cristian Mercadante
UNIMORE | ULTIMO AGGIORNAMENTO: 20/12/2019

Lezione 1

Martedì 17 settembre 2019

Credenziali

Sito: http://web.ing.unimo.it/rlancellotti/awm/ Login: AWM1920 Password: aangaexa

Richiami sul Web dinamico

Il web si sviluppò perché i tempi erano maturi. C'è molta informazione numerica e possibilità di archiviare i dati. Nascono reti di calcolatori: nel momento in cui i sistemi di calcolo sono interconnessi, posso sviluppare servizi di rete su vasta scala. Si diffondono i Personal Computers e interfacce user friendly. Vengono quindi adottati lo stack TCP/IP, il sistema DNS e il meccanismo client-server si sviluppa ancor di più con questi. Ci sono tre nuovi standard: HTTP, HTML, URL. Questo meccanismo diventa potente perché mi mette a disposizione un modo per recuperare informazioni da altri nodi, mi dà un'interfaccia immediata, rimanendo molto semplice e aperto alle evoluzioni. Diventa quindi un meccanismo per fruire di servizi. Evolve passando da un meccanismo in cui il client deve solo presentare delle informazioni, a un sistemi client/server in cui il client serve per interagire con quel server. Via via, a mano a mano che il client diventa sempre più ricco di funzionalità, l'interazione con il server diventa sempre meno stretta, fino ad arrivare a un contesto in cui il servizio proposto viene in qualche modo prodotto a partire da una composizione di servizi fault-tolerant e intercambiabili (web services). Da un lato abbiamo una necessità di ridurre il più possibile quello che è il *time-to-market*, per cui cerco di sviluppare un servizio mettendo insieme dei servizi già "precotti". Questo mi porta a far sì che i componenti siano sempre meno monolitici e rigidi, ma piccoli, flessibili, facili da adattare a situazioni differenti. Parliamo quindi di **framework**, con elementi di base che possiamo adattare.

Classificazione dei sistemi per il Web dinamico

Le risorse vengono distinte fra **risorse statiche** e **risorse dinamiche**. Poi ci sono le **risorse volatili**: sono pagine che vengono rigenerate con una certa cadenza e vengono inserite all'interno di una cache nel web server. Se abbiamo un contenuto di cui conosciamo il ciclo di vita (ad es. le informazioni sul mercato azionario, le previsioni del tempo), o lo aggiorniamo a mano (infattibile), o lo aggiorniamo dinamicamente (con spreco di risorse di calcolo, perché il contenuto è generato ad ogni richiesta), oppure periodicamente aggiorniamo il contenuto e lo conserviamo nella cache del server (ad es. metodologia push nelle CDN). Infine, ci sono le **risorse attive**: risorse che per loro natura sono statiche, ma contengono istruzioni che vengono eseguite lato client (ad es. applet Java o codice JavaScript). Queste tecnologie le usiamo per la creazione di pagine personalizzate sulla base delle caratteristiche della richiesta (cookie, posizione geografica, ecc.). Il web è anche un intermediario verso database: motori di ricerca, applicazioni di e-commerce, ecc.

Un dato URL si mappa su una risorsa nel filesystem. Questo meccanismo può essere esteso mappando un certo URL su un'operazione dinamica, che produce la risorsa e la ritorna all'utente. Dal punto di vista del protocollo HTTP, dal punto di vista del client, quest'operazione è completamente trasparente: il problema è solo del web server, che mapperà l'URL nei contenuti opportuni. In questo modo non devo cambiare la struttura del web al variare dei contenuti: non dobbiamo quindi cambiare la struttura del sito al variare delle tecnologie sottostanti.

Dal punto di vista dello sviluppo di un servizio web, mi fa comodo vedere questo processo come suddiviso in più livelli, definendo funzioni base che possa sempre trovare. Da un lato ho l'interazione con l'utente (**user interface**), poi ho una logica di presentazione (come le informazioni sono fornite all'utente: pagina html o tuple json), poi ho la logica dell'applicazione (**business logic**) e infine ho la logica dei dati (**data logic**), ovvero tutto ciò che riguarda la gestione dell'informazione. Ad esempio, quando finalizzo un ordine, il carrello deve essere trasformato in un ordine. Ma come gestisco il carrello e l'ordine? Attenzione: abbiamo parlato di funzioni di base parlando di interazione di utente con servizio web, ma questi layer li troviamo anche all'interno dei vari componenti. Nel MVC abbiamo mappature di M, V e C su tre dei quattro livelli precedenti. Quindi queste funzioni logiche saranno archetipi che troveremo implementati in contesti differenti.

Quando sviluppiamo un sito web, queste funzioni devono essere mappate su componenti eseguibili: come si mappano? Nello stesso processo, in processi separati? Quali livelli logici tengo uniti e quali no? Dovrò poi mappare i processi sull'infrastruttura di calcolo: alcuni li vorrò mettere vicini sulla stessa macchina per migliorare l'interprocess communication, ma farli andare su macchine diverse migliora la scalabilità. Oppure posso aggiungere un altro livello di virtualizzazione, usando container, che metto sulle varie macchine. Attenzione: avere quattro processi su quattro macchine differenti può degradare le prestazioni nel momento in cui devo fare scambi di dati sulla rete. A tutto ciò aggiungiamo la virtualizzazione e container.

Tutto ciò accelera anche lo sviluppo del software, perché questi componenti possono essere sviluppati indipendentemente e poi messi assieme. Si parla allora di **microservizi**, interazioni **RESTful**, logiche di **orchestrazione** (definisco come i microservizi interagiscono fra di loro), **applicazioni mashup** (faccio centinaia di richiesta a microservizi e poi genero la risorsa dinamica). Questo meccanismo dinamico tende a muoversi in una certa direzione, perché in generale ci sono dei trend che rimangono: laddove posso avere una tecnologia che semplifica lo scale-up verso architetture sempre più distribuite, questo è un pregio. Alcuni elementi della nostra interfaccia vengono inoltre spesso spostate verso il client (JavaScript). Il grande vantaggio è che abbiamo un beneficio in termini di interattività.

Tipicamente la presentation logic è quella di un server HTTP. Per il middle-tier usiamo framework MVC che implementano una funzione di presentation logic. La data logic è un qualche tipo di backend (database relazionali, NoSQL o sistemi legacy). La business logic, il livello intermedio, può essere realizzato in tantissimi modi. Dobbiamo prima capire come avviene lo sviluppo: ci sono tecnologie orientate alla pagina (PHP, JSP, ecc.); ci sono approcci fortemente object oriented, dove ad esempio definisco dei Java Bean, che implementano una logica per interfacciarsi coi dati; oppure posso usare framework più complessi per implementare la logica MVC. Per implementare i processi posso creare processi per ogni richiesta (ad es. CGI); posso avere tutto integrato all'interno del web server; posso avere un server esterno (ho un processo separato che gestisce le interazioni per le pagine dinamiche e su certe richieste il server HTTP invoca questo server).

Nelle **CGI**, a richiesta viene instanziato un nuovo processo, con comunicazioni fra processi definite dallo standard. È una tecnologia poco scalabile: ci sono casi in cui è estremamente utile, perché non mette requisiti sul web server (funziona anche con il web server più scrauso) e può aver senso quando l'interazione è tipicamente statica, ma nella maggior parte dei casi dobbiamo creare un processo per ogni richiesta (estremamente oneroso, perché devo eseguire una *fork*, seguita da *exec*).

Quindi si è passati all'introduzioni di **API per estendere le funzioni del web server**, usando librerie che girano nello stesso spazio di indirizzamento del web server, con il rischio che un guasto su un modulo tranci il web server. Inoltre, se il mio modulo non è thread-safe, allora devo serializzare le mie richieste.

Le **tecnologie orientate alla pagina** sono tipicamente fatte con linguaggi di scripting di alto livello (che non devono definire i tipi delle variabili), che gira all'interno del web server. Un esempio è la servlet Java, che ha un approccio molto prolisso, e le JSP. Questo approccio presenta limiti a livello di scalabilità, perché la parte dinamica del sito è unita al web server, quindi mi porta problemi quando voglio scalare orizzontalmente e verticalmente. Mi trovo inoltre ad avere cattive pratiche nello sviluppo del software: se continuo a pensare "una pagina, un file", è facile che duplichi del software e che sia più difficilmente manutenibile.

Cominciano a diventare importanti i requisiti di modularità, portabilità e manutenibilità del codice. Se ho bisogno di queste caratteristiche, comincio ad isolare le varie logiche, quindi vado verso tecnologie OO. Il focus è l'*information hiding*: il fatto che ci sia un database o un altro, oppure ristrutturo i miei dati su DB differenti, cambierò solo gli oggetti che si occupano di manipolare quei dati. Inoltre, se cominciassi a mettere insieme i miei oggetti, potrei accorgermi di poter avere oggetti più generici e riutilizzabili. Questo approccio va in crisi quando mi trovo a dover realizzare oggetti che voglio implementare una parte di presentation logic.

Oggi un maggiore interesse si è spostato sui **framework MVC**. L'idea è quella di definire tre funzioni logiche ben separate l'una dalle altre. Il controller gestisce l'interazione con l'utente, la view definisce come verranno presentati i dati, mentre il model è quella parte del middle-tier che parla con il database. In questo modo ho buona manutenibilità al cambiamento della struttura dei dati (dovrò riscrivere solo il model). Se voglio modificare l'output, mi concentro sulla view. È un sistema che è abbastanza potente per consentirmi di ragionare, avendo una visione completa dell'applicazione, limitando il più possibile l'intervento in caso di manutenzione.

Le generazioni del Web

Inizialmente c'erano server che servivano risorse statiche (e anche CGI) al client. Le prestazioni all'inizio non sono critiche. A mano a mano che il web diventa popolare, si aggiungono altri requisiti, perché vengono veicolate informazioni e nuovi servizi vengono forniti. Bisogna fornire risposte tempestivamente, aumentare l'affidabilità e la sicurezza. Dopo il 2005 si cambia il paradigma del web: da risorsa esclusivamente pull (un produttore e tanti consumatori) a risorsa push (tanti produttori di contenuti). Tutti vogliono potere accedere in ogni momento ai servizi online (la multicanalità diventa fondamentale: devo avere più view). Il volume di dati esplode.

Lezione 2

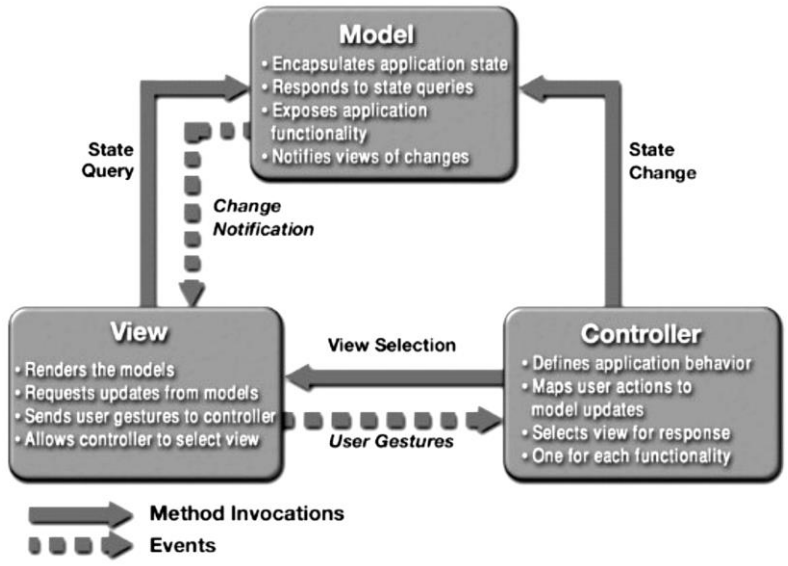
Venerdì 20 settembre 2019

Model-View-Controller – MVC

Pensiamo ai punti di debolezza delle altre tecnologie. J2EE ha come pecca l'uso delle servlet, che ricordano le CGI scritte in Java, ma Java è multi-threaded, quindi molto meno oneroso. Tomcat usa un modello con più thread pronti a servire delle richieste. Siccome Tomcat è sia un server web, ma anche un server J2EE, allora abbiamo molti vantaggi. Inoltre, le servlet hanno questi oggetti richieste e risposta che mi consentono di manipolare molto agilmente le richieste e le risposte (appunto). Mentre le CGI usano variabili d'ambiente, standard input, standard output, ecc. Infine, le servlet si portano dietro tutto il mondo J2EE, ovvero sono aperte a tutte le altre tecnologie di questo mondo: Enterprise Java Beans, API per interrogazione DB. Ma la servlet è terribile per fare pagine web: il codice è molto pesante. C'è un elemento un po' più critico, che è quello dello *skill mismatch*: un sito web moderno ha dietro un sacco di persone diverse, con profili professionali molto diversi fra di loro. Con le servlet abbiamo chi si occupa di programmazione, chi di grafica, e alla fine tutti insistono sullo stesso file molte volte. Diventa importante avere un modello di sviluppo che tenga separate le competenze: chi si occupa di design non deve guardare il codice, ma l'HTML, CSS, ecc.

Ci sono alcuni meccanismi per evitare ciò, come fornire API per includere dei frammenti di pagina. Un passo abbastanza deciso verso la soluzione dei problemi è la JSP, dove comincio a risolvere il problema del codice molto prolisso delle servlet. Per contro, lo *skill mismatch* resta, il codice boilerplate rimane duplicato, la redirezione può non essere agile. Esistono soluzioni, come i template JSP, che aiutano (se le so usare bene), ma non ho una vera guida che mi porta a lavorare secondo certi criteri. EJB sono potentissimi, perché hanno il focus sull'information hiding, dove definisco oggetti DAO (Data Access Object) che mi nascondono il backend. Questo va molto bene, ma per le parti di BL che hanno una struttura che si descrive meglio con un flow chart, piuttosto che con un class diagram, mi trovo a costruire delle librerie. Quindi c'è il rischio di mettere cose tutte insieme, andando a definire un EJB che si occupa dell'anagrafe clienti (ad esempio), e questo oggetto conterrà sia la parte di accesso ai dati, sia la parte di produzione dell'output. È deleterio, perché quando devo cambiare la PL devo fare una modifica su ognuna delle classi. Se devo implementare multicanalità, devo mettere mano a tonnellate di codice: poco manutenibile. Quindi per la parte di BL che ha a che fare coi dati, l'approccio agli oggetti va molto bene, ma per il resto non tanto.

La BL contiene a sua volta al suo interno alcuni elementi logici che sono differenti fra di loro (PL, DL, middle-tier). Nel **middle-tier** ci sono elementi differenti, ovvero la parte che produce le pagine dinamiche, interazione con DB ecc. Questo porta a definire il middle-tier come un qualcosa che è formato da tre aspetti diversi: Model, View, Controller. L'idea è che la V presenta i dati che poi verranno presentati all'utente attraverso il server web, l'utente interagisce con la V grazie alla mediazione del C, che agisce con il M, che si interfaccia con i dati. Il M sarà in grado di manipolare i dati. Quando c'è una modifica dei dati, le V opportune saranno notificate e andranno a modificare la rappresentazione stessa all'utente, che può interagire con l'interfaccia e così via. Questo approccio è estremamente ortogonale rispetto al suo campo di utilizzo: è stato introdotto negli anni 80 ed è il paradigma che ha guidato lo sviluppo dei componenti grafici di Java (classi Swing). Il punto è che grazie a questa struttura, l'approccio MVC è adeguato a sistemi con una forte interattività. Recepisce le best-practices dell'ingegneria del software: information hiding, disaccoppiamento, semplicità e controllo basato su deleghe.



Il M si occupa di incapsulare gli oggetti informativi con cui lavoro, offre metodi con cui la V può chiedere informazioni. Inoltre, prevede un meccanismo di notifica, per cui le V devono essere notificate a fronte di cambiamenti sul M. Questo è un elemento abbastanza critico su cui ci soffermeremo. La V a sua volta è oggetto di interazione con l'utente, per cui se questo fa un click, questo determina una notifica verso il C. A questo punto il C potrebbe notificare il M e potrebbe anche cambiare la V. Ad esempio, quando si verifica l'invio delle informazioni sulla carta di credito, il C dovrebbe determinare il cambio della V.

Il M tipicamente è quello che si interfaccia con i repository dati. Conterrà tutta la logica implementata con gli oggetti DAO. Si usa un modello *object relational*, dove vado a definire una struttura delle classi che ricalca la struttura del DB. In Django, il M è definito

come una serie di classi, che se viene cambiata, Django si accorge che ci sono cambiamenti e genera codice per migrare da una versione all'altra del DB. Le V sono le parti di codice che generano le pagine. Il M può avere diverse V. I C sono la parte più ambigua dell'MVC. Gestiscono eventi. Nella maggior parte dei framework web, i C fanno URL routing, ovvero mappano gli URL in chiamate opportune alle V.

Ci sono attività legate allo sviluppo di un sistema web che possono essere con il MVC portate avanti separatamente. Inoltre, è facile capire in quale delle tre parti risiede un problema, senza dover riscrivere parti sparse su tutto il progetto. Questo semplifica la parte di unit testing, perché posso testare le tre parti separatamente.

Il M nasce per contenere il modello concettuale dei dati della nostra applicazione. Deve essere progettato per essere indipendente da V e C, ma devo comunque sapere cosa fanno, perché deve esporre interfacce adatte. C'è un orientamento forte verso l'information hiding. Il C definisce le azioni che possono essere eseguite sul M. Ha un elemento di vicinanza con la V, pur essendo concettualmente separato. Dipende molto dal M, quindi è opportuno avere un'idea di come sia strutturato. La V è un osservatore passivo del modello: può solo chiedergli dati ed essere notificato sugli update dei dati stessi, con delle callback. In un contesto web, questa cosa è difficile. Se abbiamo una situazione in cui sappiamo che i dati sono soggetti a modifica, magari è necessario definire un qualche meccanismo di polling (se la frequenza degli aggiornamenti non è troppo alta, e gli aggiornamenti non sono estremamente critici) oppure (se l'interattività è molto importante), allora sono necessarie ad esempio web socket per le notifiche. A volte C e V possono essere unite fra di loro: nel caso J2EE, la V e la C sono uniti sotto il nome di *Delegate*. Questo meccanismo basato su deleghe e listener, lo usiamo quando serve una forte interattività (eventi asincroni).

Introduzione a Django (1)

Uno parte da un'applicazione fatta da zero, poi ne fa un'altra e si accorge che per parti sono simili. Quindi fa una libreria condivisa e ogni volta mi rendo conto che ogni pezzo di codice posso renderlo più generale: abbiamo fatto un framework. In Django avremo tre moduli che fungono da MVC. Abbiamo una serie di frammenti automatici per gestire l'interazione con database e la fase di autenticazione dell'utente. Nel M si definisce la struttura delle classi nel model e questa viene trasferita verso la struttura del DB. Questo viene fatto con un meccanismo di versioning, che prevede script generati per la migrazione del DB. Offre una serie di scorciatoie per fare le cose comuni, senza togliere strumenti di basso livello per casi atipici. Nel C definiamo una serie di regular expressions per fare matching sugli URL per capire quale V attivare.

La prima cosa che viene fatta è eseguire una prima migrazione per la versione 0 che contiene il codice così per creare il M come l'abbiamo previsto. Se il DB esiste, viene generato l'SQL per la migrazione. Non è necessario interagire direttamente col DB: ci pensa il framework sulla base di parametri di connessione. Ogni tupla nel nostro DB sarà identificata da un ID numerico gestito automaticamente: questo viene utilizzato per tutte le referenziazioni. A livello di struttura di codice, è possibile indicare le foreign key, il cui accesso viene fatto dal M in modo trasparente. Se abbiamo relazioni di tipo molti-a-molti, con Django non è necessario fare la reificazione, ci pensa lui. La V può essere vista come una funzione python per gestire l'output: è quindi possibile inserire HTML nel codice. Quando produciamo codice HTML, Django ci consente di definire un template, ovvero un file HTML con alcuni tag aggiuntivi. Il C è sostanzialmente un modulo python sotto forma di matching fra URL e V corrispondente. Il C permette inoltre di integrare altri moduli, come una serie di pagine di amministrazione per interagire direttamente con il M. Come si pensano gli URL? Devono essere il più possibili riutilizzabili, evitando anche l'hardcoding degli URL nel codice. I template servono per rendere più facile la vita della V. Bisogna passare delle informazioni da C a V. L'idea di base è cercare di evitare la ridondanza, quindi posso definire delle forme di ereditarietà. L'idea è quella che il template deve essere una scorciatoia. Se ho bisogno di un qualcosa di più complicato, uso python. I template introducono feature per rendere il codice più robusto: avendo dei modelli, è molto più difficile fare SQL-injection. Python introduce un meccanismo per evitare attacchi basati sulla sottomissione di form: offre protezione a CSRF. Un progetto può essere spezzettato in tante applicazioni, quindi può corrispondere a un'istanza di un server Tomcat.

Lezione 3

Martedì 24 settembre 2019

Introduzione a Django (2)

Prima applicazione

Andremo a costruire domande e risposte che gli utenti che possono selezionare. Ci sarà una parte rivolta al pubblico, per visualizzare le domande e di votare. Un'altra parte è un'interfaccia di amministrazione per inserire nuove domande, modificare le opzioni delle risposte. Verranno utilizzati diversi strumenti del framework: la parte di amministratore è già pronta in django. Come backend useremo SQLite.