ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
SEDE DI CESENA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea in Scienze dell'Informazione

GESTIONE TRAMITE WIKI DI SITI WEB BASATI SU JAVA E WEB-SERVICE

Relazione finale in:
Tecnologie Web

Relatore: **Prof. Mario Bravetti**

Presentata da: **Marianna Galeati**

Sessione II

Anno Accademico 2011/2012

Indice

INDICE **1**

INDICE DELLE FIGURE **1**

CAPITOLO 1 INTRODUZIONE **1**

CAPITOLO 2 CONCETTI DI BASE **5**

2.1 WEB SERVICE 5

2.1.1 Cosa sono e a cosa servono *5*

2.1.2 Come funzionano? *8*

2.1.3 Soap *9*

2.1.4 WSDL *11*

2.1.4.1 Struttura di un file WSDL *12*

2.1.5 UDDI *16*

2.2 JAX-WS 18

2.3 BREVE APPROFONDIMENTO SULLA LIBRERIA JAXB 28

2.3.1 XML-Schema ⟶ Classi JAVA *30*

2.3.2 XML-Schema ⟵ Classi JAVA *33*

CAPITOLO 3 GESTIONE DI SITI WEB TRAMITE WIKI E COMPILAZIONE **37**

3.1 INTRODUZIONE (ANALISI E SPECIFICHE DEL PROBLEMA) 37

3.2 COMPILAZIONE ONLINE 38

3.3 GESTIONE UNIFICATA 45

3.3.1 Idea di gestione unificata *45*

3.3.2 Compiti *46*

3.3.3 WebService *49*

CAPITOLO 4 GESTIONE REMOTA DI WEBSERVICE **51**

4.1 LATO CLIENT 53

4.1.1 Gestione *54*

4.1.2 Editing *58*

4.2 FUNZIONAMENTO WEBSERVICE 63

4.2.1 Operation: getDir *65*

4.2.2 Operation: getFile *67*

4.2.3 Operation: postFile *68*

4.2.4 Operation: compile *69*

4.3 STRUMENTI DI SVILUPPO E DEPLOYMENT 70

CAPITOLO 6 CONCLUSIONI 73

BIBLIOGRAFIA 75

APPENDICI 81

A. GESTIONE.JSP 81

B. EDITUNICO.JSP 86

C. MANAGER.JAVA 88

Indice delle Figure

FIGURA 1 –CARATTERISTICHE DI JAXB 29
FIGURA 2– DA XML-SCHEMA A CLASSI JAVA 30
FIGURA 3– CLASSI GENERATE IN UNA STRUTTURA AD ALBERO 32
FIGURA 4 – ALBERO RELATIVO ALLA STRUTTURA DELL' XML SCHEMA 32
FIGURA 5 – BOTTONE COMPILA 39
FIGURA 6 – ERRORE DI COMPILAZIONE 42
FIGURA 7 – BOTTONE VISUALIZZA NUMERAZIONE 43
FIGURA 8 – BOTTONI DI UN FILE JAVA 43
FIGURA 9 – TEXTAREA CON NUMERAZIONE 44
FIGURA 10 – LINK DELLA PAGINA DI GESTIONE PER I COMPITI 46
FIGURA 11 – VISUALIZZAZIONE DELLA LISTA DEI COMPITI 47
FIGURA 12 – VISUALIZZAZIONE ELENCO FILE DI UN COMPITO 47
FIGURA 13- PAGINA DI EDITING DI UN COMPITO 47
FIGURA 14- FORM DI NAVIGAZIONE 55
FIGURA 15- BOTTONE COMPILA DEI WEB SERVICE 57
FIGURA 16- BOTTONI EDIT E VEDI NUMERAZIONE 58
FIGURA 17- PAGINA DI EDITING 60
FIGURA 18 – PAGINA PER VISUALIZZARE NUMERAZIONE RIGHE 60
FIGURA 19-BOTTONI ELIMINA E RINOMINA 61
FIGURA 20 – INTERFACCIA PER CREARE OPERATION 64

Capitolo 1 Introduzione

Il tema principale di questa tesi riguarda i siti web che utilizzano un struttura multi-tier (multi-livello) [AMT01], cioè una particolare architettura che prevede la suddivisione del sistema, risultando in questo modo distribuita su più strati e fornendo così un modello per creare applicazioni flessibili e riutilizzabili. Ciò consente la progettazione e amministrazione modulare del sistema offrendo così la possibilità di risparmiare tempo e denaro.

L'architettura multi-tier svolge un ruolo importante nella progettazione di applicazioni Web, specialmente basate sulla distribuzione di più server, i quali risultano fondamentali per la realizzazione di questo tipo di struttura, poiché offrono servizi web al sito, nonostante siano completamente indipendenti da esso. Questo schema generale è piuttosto diffuso e costituisce un'architettura di riferimento per molte tecnologie moderne, per cui risulta molto interessante e importante cercare di approfondire il suo meccanismo e nello stesso tempo cercare di evolvere alcune funzionalità al suo interno.

In questo ambito una problematica importante riguarda le tecnologie per la gestione di siti web e, in particolare, quelle che consentono di apportare cambiamenti ad un sito da remoto, senza dover accedere fisicamente ai server o dover effettuare nuovamente il deployment dei loro servizi.

Per questo motivo la tesi proposta è incentrata proprio su questo argomento, cercando di evolvere l'utilizzo dei servizi e di sviluppare la possibilità di poterli gestire durante il loro utilizzo. Si cercherà dunque di affrontare il problema appena esposto, mettendo in evidenza una serie di problematiche che si possono incontrare nella realizzazione di un "modulo" che possa

essere aggiunto ad un sito web multi-tier per consentire la sua gestione dinamica.

L'esigenza di sviluppare un sistema uniforme e distribuito, costituito da funzionalità indipendenti, trova la sua massima espressione realizzativa, nell'utilizzo della tecnologia dei *Web Services* [WSDL03, TUR09a, W3S-WWS, WIK09a, CEW02A, CLA03A, DDDTWSTI03, DEI03A]. Per *Web Service* s'intende un servizio disponibile sul Web che usi un sistema di messaggistica XML (eXtensible Markup Language [W3C04A, CER06]) standardizzato (Protocollo SOAP [SOAP03]) e che non è in nessun modo legato ad alcun sistema operativo o linguaggio di programmazione. I Web Service sono una tecnologia di recente sviluppo, per quanto ormai affermata; sono componenti software che possono essere pubblicati, localizzati e consumati attraverso il web. Sono fondamentali per questo tipo di architettura, in quanto assicurano l'indipendenza dalla piattaforma e dal protocollo di trasporto, grazie al fatto che utilizzano tecnologie standard e costruite su XML (eXtensible Markup Language [W3C04A]). Si è rivelato fondamentale, inoltre, l' utilizzo dell' API JAXB (Java Architecture for XML Binding [JAXB07, OME03, APBWS07, GSD03A]), un insieme di classi, metodi, campi e librerie per eseguire in maniera efficiente e veloce il binding XML-JAVA, utile al fine di interfacciarsi in maniera dinamica con i documenti basati su XML.

Rispettando questi concetti fondamentali si cercherà di creare un semplice, ma efficace, "modulo" che permetterà di gestire web service durante il loro utilizzo. Tutto ciò è stato implementato usando tecnologie web basate su java [SM11A, AR06, RFGP04, JTRPC07].

Infatti il "modulo" della tesi, che si intende esporre nei seguenti capitoli, viene realizzato utilizzando quest'ultima tecnologia, in quanto è

particolarmente adatta alle esigenze del web, la quale permette di realizzare logiche anche molto complesse. In applicazioni con elevata complessità dal lato client, Java è utile per gestire operazioni di dialogo con l'utente; lato server riesce a ottenere scalabilità rispetto all'iterazione con client multipli, per effetto del multi-threading, cioè la capacità di gestire in modo concorrente processi multipli di dialogo con client diversi. Con il termine "java" ormai non ci si riferisce più al linguaggio di programmazione, ma ad un insieme di tecnologie che ne estendono la funzionalità.

Nella tesi di Gessica Valbonesi [TDGV] era stato realizzato un meccanismo per modificare il codice di un sito tramite la tecnologia wiki. Tutto ciò però era stato realizzando basandosi su un unico server, invece ora vorremmo espandere questa idea di modifica anche ad un livello di multi-tier.

Avendo analizzato tutti gli aspetti descritti sopra, che al giorno d'oggi sono fondamentali, si è deciso durante questa tesi di creare web services dotati di un meccanismo di "reflection", cioè in grado di mandare il sorgente del proprio codice, per poter essere modificato dall'utente, e successivamente in grado di ricevere un nuovo sorgente per poi auto modificarsi.

La soluzione proposta è basata sull'idea di creare un manager per ogni web service che offre un servizio al sito, per poter sviluppare quello che è stato descritto precedentemente. In particolare ogni web service manager ospita al suo interno tutte le operation necessarie per apportare modifiche al codice, per compilare,ecc… che verranno richiamate a livello client nelle jsp di gestione e di edit per effettuare determinate operazioni come ricevere il codice del web service, modificarlo, compilarlo e ricevere i file presenti nelle directory dei web service.

Come esempio di applicazione della metodologia e delle tecnologie proposte si è continuato il lavoro di sviluppo del modulo di gestione per il sito di Tecnologie Web del corso di laurea in Scienze e Tecnologie informatiche, portato avanti in [TDGV], realizzando l'unificazione delle pagine di gestione e l'auto modifica dei web service. Inoltre è stata aggiunta una nuova funzionalità, integrandola con quelle precedenti, per la gestione dei progetti tramite wiki.

La struttura della tesi è la seguente:

- nel secondo capitolo sono introdotti i concetti base per poter comprendere la tesi;
- nel terzo capitolo si affronta la descrizione di come è stato gestito in modo unificato il sito di tecnologie web e il concetto di compilazione online;
- nel quarto capitolo viene descritto come è stato possibile gestire i web service in modo remoto e tutti gli aspetti implementativi;
- nel quinto ed ultimo capitolo è stato riassunto il lavoro svolto durante la tesi, analizzando gli sviluppi futuri che potrà avere;
- infine nelle appendici A, B e C si presentano, rispettivamente, i frammenti di codice della jsp di gestione, di edit e il manager.

Capitolo 2 Concetti di Base

In questo capitolo verranno introdotti i concetti base utilizzati nel resto della tesi, in modo da permetterne la comprensione a qualsiasi persona avente delle conoscenze di base sull'informatica.

2.1 Web Services

La tecnologia dei Web Services rappresenta le fondamenta della struttura di quanto progettato; dunque sarà anche la prima tecnologia ad essere presentata.

2.1.1 Cosa sono e a cosa servono

Un Web Service è un componente applicativo. Possiamo definirlo come un sistema software in grado di mettersi al servizio di un' applicazione, comunicando su di una medesima rete tramite il protocollo HTTP[HTTP, WIK09b]. Un Web Service consente quindi alle applicazioni che vi si collegano di usufruire delle funzioni che mette a disposizione.

Per fare un esempio potremmo ipotizzare un Web Service che chiameremo "cambiavalute". Esso, fornisce le seguenti operazioni: cambio euro/dollaro e viceversa. Questo Web Service potrebbe essere offerto da un istituto bancario ed una nostra applicazione potrebbe utilizzarlo per effettuare le operazioni di cambio senza doversi preoccupare dei tassi in vigore al momento dell'operazione.

Già dopo questo primo esempio si dovrebbe aver notato che le operazioni svolte da un Web Service non sono nulla di eclatante, qualsiasi comune applicazione potrebbe infatti effettuare l'operazione di cambio. Ciò che una comune applicazione però non può fare è **mettersi in comunicazione** con un altro software come ha fatto cambiavalute nel nostro esempio. Un Web Service, infatti, comunica tramite protocolli e standard definiti "aperti", quindi sempre a disposizione degli sviluppatori.

I Web Service hanno un'altra caratteristica molto particolare ed utile al loro scopo: sono auto-contenuti ed auto-descrittivi. Due concetti molto semplici. Un Web Service è in grado di offrire un'interfaccia software assieme alla descrizione delle sue caratteristiche, cioè è in grado di farci sapere che funzioni mette a disposizione (senza bisogno di conoscerle a priori) e ci permette inoltre di capire come vanno utilizzate. Ciò significa che (sempre per rimanere al nostro esempio) con una semplice connessione a cambiavalute, anche senza conoscerlo, possiamo stabilire le operazioni che fornisce e possiamo subito iniziare ad usarle perché ogni operazione ha una sua descrizione comprendente i parametri che si aspetta di ricevere, quelli che restituirà ed il tipo di entrambi. Questa caratteristica dei Web Service è estremamente utile se si considera che possono essere trovati utilizzando l' UDDI (Universal Description, Discoverey and Integration)[UDDI], un servizio di directory disponibile sul Web dove gli interessati possono registrare e cercare servizi web.

Si propone di seguito un elenco nel quale vengono spiegati i motivi principali, per i quali si dovrebbe scegliere l' utilizzo dei Web Service:

- I Web Service permettono l'interoperabilità tra diverse applicazioni software e su diverse piattaforme hardware/software.

- Utilizzano un formato dei dati di tipo testuale, quindi più comprensibile e più facile da utilizzare per gli sviluppatori (esclusi ovviamente i trasferimenti di dati di tipo binario).

- Normalmente, essendo basati sul protocollo HTTP, non richiedono modifiche alle regole di sicurezza utilizzate come filtro dai firewall.

- Sono semplici da utilizzare e possono essere combinati l'uno con l'altro (indipendentemente da chi li fornisce e da dove vengono resi disponibili) per formare servizi "integrati" e complessi.

- Permettono di riutilizzare applicazioni già sviluppate.

- Fintanto che l'interfaccia rimane costante, le modifiche effettuate ai servizi rimangono trasparenti.

- I *servizi web* sono in grado di pubblicare le loro funzioni e di scambiare dati con il resto del mondo.

- Tutte le informazioni vengono scambiate attraverso protocolli "aperti".

Tuttavia, come ogni tecnologia anche i Web service presentano alcuni problemi:

- **Le performance**. I Web Service presentano performance drasticamente inferiori rispetto ad altri metodi di comunicazione utilizzabili in rete. Questo svantaggio è legato alla natura stessa dei *servizi web*. Essendo basati su XML[TRI03] ogni trasferimento di dati richiede l'inserimento di un notevole numero di dati supplementari (i tag XML) indispensabili per la descrizione dell'operazione. Inoltre tutti i dati inviati richiedono di essere prima

codificati e poi decodificati ai capi della connessione. Queste due caratteristiche dei Web service li rendono poco adatti a flussi di dati intensi o dove la velocità dell'applicazione rappresenti un fattore critico. Negli ultimi tempi, tuttavia, sono stati fatti notevoli passi avanti per contrastare questo problema.

- **Il protocollo base, HTTP**. Quando si sviluppa un Web service è necessario tener conto del protocollo di base. È quindi indispensabile disporre di un'applicazione terza che gestisca le richieste HTTP oppure è necessario includerla direttamente nel codice del nostro programma qualora si desideri la sua totale indipendenza. Va detto comunque che generalmente il codice che implementa un Web service viene fatto eseguire da un Web server (es. Apache) tramite CGI (per es. con Python) o tramite appositi moduli (vedi PHP). Eseguendo il codice del Web service attraverso un server web la gestione di HTTP è immediatamente assicurata.

2.1.2 Come funzionano?

Ora che è stato spiegato cosa sono e a cosa servono i *Web Service*, si cercherà di capire meglio come funzionano. Il loro protocollo di base è HTTP. Questo protocollo si occupa di mettere in comunicazione il *servizio web* con l'applicazione che intende usufruire delle sue funzioni.

Oltre ad HTTP però, i *servizi web* utilizzano molti altri standard web, tutti basati su XML, tra cui:

- **XML Schema**: è un linguaggio di descrizione del contenuto di un documento XML che ha lo scopo di stabilire opportuni vincoli per i contenuti.
- **SOAP** (Simple Object Access Protocol): è il protocollo XML di scambio dei messaggi che consente la programmazione distribuita dei servizi Web[SOAP03].
- **WSDL** (Web Services Description Language): consente di descrivere il servizio tramite un documento XML[WSDL03, KAO01].
- **UDDI** (Universal Description, Discovery and Integration)

È importante sottolineare che XML può essere utilizzato correttamente tra piattaforme differenti (Linux, Windows, Mac) e differenti linguaggi di programmazione. XML è inoltre in grado di esprimere messaggi e funzioni anche molto complesse e garantisce che tutti i dati scambiati possano essere utilizzati ad entrambi i capi della connessione. Si può quindi dire che i Web Service sono basati su XML ed HTTP e che possono essere utilizzati su ogni piattaforma e con ogni tipo di software.

2.1.3 SOAP

SOAP è stato introdotto dal consorzio W3C, il padre del web, e come spesso accade nel mondo dell' informatica il nome è un acronimo. Tendenzialmente si intende Simple Object Access Protocol ma in qualche testo viene anche inteso come Service Oriented Architecture Protocol e visto che quando si parla di Web Services spesso si fa riferimento alla filosofia SOA ovvero delle architetture orientate ai servizi, è opportuno ricordare anche tale interpretazione dell' acronimo, anche se meno gettonata.

Soap nasce dall' evoluzione di XML-RPC. Il termine RPC, Remote Procedure Control Calling, può rappresentare il concetto antesignano dei Web Services. Esso aveva dato risposta ad una esigenza inizialmente legata ad una realtà prettamente aziendale e che ha manifestato alcune lacune a confronto con il tentativo di distribuzione globale.

Ecco alcuni esempi di RPC:

- CORBA (Common Object Request Broker Architecture)
- RMI (Remote Method Invocation in Java)
- DCOM (Distribuited Component Model)

Soap è quindi un protocollo leggero basato su XML, per lo scambio di informazioni in un ambiente distribuito, finalizzato alla descrizione formale di messaggi che rappresentano richieste e risposte di esecuzione di chiamate a procedure remote.

Anche se in linea di massima, quando si parla di Soap, si tende ad accostarlo esclusivamente al classico HTTP basato su TCP/IP, in realtà dal punto di vista teorico esso prescinde dal protocollo di trasmissione ed è applicabile, infatti, anche ad Https, Smtp, Pop3, Imap, Java Messaging Services, Blocks Extensible Exchange Protocol.

Tuttavia, il ruolo preponderante di Soap è quello che lo lega ai Web Services, ed il legame è talmente stretto che è possibile trovare questi ultimi definiti come dei "servizi o componenti che rendono fruibili le loro funzionalità tramite messaggistica Soap con protocollo Http". La soluzione proposta da Soap permette, quindi, di superare l' enorme disomogeneità dei

formati proprietari, proposti dalle applicazioni, definendo un vero e proprio standard. In pratica si pongono le basi affinché si possa evitare di dover inventare ex novo volta per volta un formato ed una struttura di documenti XML per richieste e risposte di servizi remoti.

I servizi web, pur nella loro grande differenza e vastità di offerta, avranno perciò una cosa in comune: il linguaggio utilizzato per giovarsi delle loro performances sarà Soap e di conseguenza la grammatica indispensabile a tutti i programmatori per orientarsi in questo nuovo mondo sarà l' XML.

2.1.4 WSDL

Abbiamo quindi una dettagliata panoramica di come, tramite SOAP, si possano bypassare gli scogli dovuti alla incompatibilità di numerosi e differenti linguaggi in un ambiente distribuito. Abbiamo anche inizialmente evidenziato come i Web Services si pongano tra gli obiettivi quello di rendere i servizi interpretabili e interpellabili non solo dai comuni clients "umani" ma anche da altri agenti software. In tale prospettiva il comune protocollo client-server assume una diversa sfumatura che ci obbligherà a parlare di Server Consumer e Server Provider.

In questa nuova ottica si può affermare che SOAP fornisce i mezzi affinché un server consumer possa richiamare funzioni remote e possa poi interpretare i valori ritornati dall' elaborazione, avvenuta sul server provider, dei parametri passati dal server consumer. Permangono, tuttavia, diverse problematiche.

Prima di tutto è importante capire come si può venire a conoscenza dei metodi che possiamo chiamare, dei parametri che essi ricevono e dei valori

che ritornano, perché a nulla vale tutto ciò di cui abbiamo parlato se poi l'implementazione ed i metodi dei Web Services sono noti solo agli sviluppatori degli stessi.

Per ovviare anche a ciò, guarda caso, s'è fatto ricorso ancora una volta ad XML, questa volta nella forma e manifestazione di files WSDL (Web Services Description Language). Per questa sua funzione il linguaggio di descrizione dei servizi web può ricordare i Javadocs che parimenti offrono una dettagliata presentazione dei metodi e delle classi utilizzabili, con la differenza che un file WSDL rende disponibili anche le coordinate fisiche e le metodologie per l' accesso al servizio stesso.

2.1.4.1 **Struttura di un file WSDL**

Innanzitutto sottolineiamo il fatto che, mentre per SOAP parlavamo di messaggio, coerentemente con le caratteristiche che abbiamo espresso e che fanno di SOAP un linguaggio che riveste un ruolo determinante nel trasporto dell' informazione, per WSDL parliamo semplicemente di files, per la natura più statica di questo formalismo.

Dal punto di vista formale, un documento WSDL è composto da sette tipologie di elementi:

- Tipi
- Messaggi
- Operazioni
- Tipi di porta

- Binding
- Porte
- Servizi

Un TIPO è la forma più elementare d' elemento WSDL ed è analogo ad uno schema XML (file con estensione *.xsd) inserito in un file *.wsdl. Per chi non avesse dimestichezza con gli schemas XML si sappia che sono un formalismo (analogo alle dtd ma più potente ed espressivo) per indicare la struttura di documenti XML e validarli. Nel nostro caso, come abbiamo visto, possiamo esprimere una classe Java tramite XML, e attraverso gli Schema, sotto le mentite spoglie di elemento <types> in WSDL, possiamo controllare la correttezza sintattica degli oggetti utilizzati o restituiti dal nostro Web Service. Per fare le giuste distinzioni, vengono usati per indicare i corrispondenti dei campi di una classe Java (le classi senza i metodi, se è più chiaro) o una struct in C o C++.

Il concetto di MESSAGGIO è legato a quello di parametri e tipo ritornato dai metodi. Per la struttura dei Web Services bisognerà cominciare a ragionare secondo i tre canali principali: input, output ed errore. In tal senso ci sarà un messaggio per l' input che ospiterà i parametri richiesti dal metodo, uno per l' output che descriverà il dato ritornato, ed un messaggio cosiddetto fault che rappresenta un caso particolare (quello d' errore) di output.

Le OPERAZIONI sono i corrispondenti dei metodi e hanno come figli dei messaggi che rappresentano l' input, l' output e l' output in caso di errore, se ne possono individuare di quattro tipologie:

- **one-way**: ovvero operazioni monodirezionali dal client al servizio, che ricordano i metodi void, ovvero che non ritornano nulla. In questo caso, infatti, è ammesso solo il messaggio di input.

- **request/response**: classica operazione bidirezionale che va dal client al server e viceversa, ammette tutte e tre le tipologie di messaggi: input, output e fault.

- **notification**: operazione monodirezionale inversa alla one-way, che quindi va dal server al client. Ammette solo output.

- **solicit/responde**: operazione bidirezionale inversa alla più classica request/response. Essa va dal server al client e viceversa. Ammette tutti e tre i messaggi.

È indispensabile sottolineare come le operazioni, all' interno di WSDL, siano elementi figli di un altro elemento, il TIPO DI PORTA. Secondo l W3C, i TIPI DI PORTA sono insiemi di operazioni astratte e dei loro messaggi. Sono perciò collezioni di operazioni analoghe. L' elemento BINDING fornisce le informazioni indispensabili per utilizzare le porte, in particolare il metodo utilizzato per il trasporto e il formato di comunicazione.

Rimane da definire l' indirizzo IP della macchina che ospita il servizio. Ecco che tramite l' elemento wsdl:port possiamo indicare il binding, il nome dell'operazione e l' indirizzo di rete univoco atto a determinare univocamente il nostro Web Service.

L' ultimo elemento di un file wsdl che analizzeremo, ed il primo per importanza è SERVICE, che è padre degli elementi PORTA e racchiude

quindi le descrizioni di quelle che sono le effettive utilità offerte dal nostro Web Service.

Le varie analogie tra gli elementi WSDL e le parti di una classe Java, sono riassunte nella tabella sottostante:

WSDL	JAVA
Tipi	classi senza metodi
Messaggi	parametri, valori di ritorno, errori
tipi di porta e operazioni	Interfacce
Binding	protocollo di comunicazione remota
servizi e porte	classi remote

Tabella 1 – Mappatura tra elementi del WSDL e strutture JAVA

2.1.5 **UDDI**

Finora abbiamo visto come i Web Service comunicano tra di loro tramite SOAP e come siano definiti tramite WSDL. La prossima domanda sarà: come possono essere reperiti?

La risposta è semplice: attraverso un Web Service specifico. Quale soluzione migliore, infatti, che reperire un servizio web proprio tramite una tecnologia analoga a quella fin qui descritta, sfruttandone tutti i vantaggi messi a disposizione?

È un po' come andare a cercare la definizione di vocabolario, sfogliando le pagine dello stesso; nasce così lo Universal Description, Discovery and Integration.

UDDI
SOAP
XML
Protocolli di Trasporto (HTTP, HTTPS, ...)

Tabella 2 - Rappresentazione gerarchica di UDDI

I servizi offerti da UDDI sono in linea di massima due:

- Registrazione e pubblicazione di un proprio Web Service.
- Ricerca di un Web Service adatto alle nostre esigenze.

Si definiscono inoltre tre grandi blocchi di informazioni:

- Le Pagine Bianche: indirizzate più alle persone che alle applicazioni contenenti informazioni riguardo l' azienda ed i recapiti telefonici e web presso i quali poter contattare delle figure aziendali per supporti tecnici ed informazioni varie.

- Le Pagine Gialle: sono il cuore di UDDI, offrono la possibilità di effettuare ricerche di Web Services secondo numerose politiche di aggregazione per categoria, ragione sociale, tipologia di servizio, aziende o attività affini...

- Le Pagine Verdi: rivolte agli sviluppatori di software, contengono informazioni utili ai programmatori per usufruire del Web Service desiderato.

Elemento fondamentale di UDDI sono i tag businessEntity che mantengono una organizzazione razionale ed indicizzata delle principali informazioni delle aziende in questione. L' attributo più importante è tuttavia businessKey che rappresenta la chiave univoca che identifica le diverse attività ed è meglio noto come uuid (128 bit nel formato xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx).

Analizziamo ora le operazioni di pubblicazione.

Mentre le interrogazioni dei registri UDDI sono aperte a chiunque, la pubblicazione è soggetta a dover gestire le autenticazioni tramite un token rilasciato dal server UDDI a seguito della solita procedura di immissione di

un ID ed una Password. Terminata l'interazione tra server e client il contrassegno token viene eliminato. Le principali operazioni per cui è necessaria l' autenticazione sono:

- Registrazione: comando save.
- Rimozione: comando delete.

A causa dell' architettura di SOAP che non prevede strumenti nativi per gestire queste situazioni, avremo che, oltre ai parametri necessari alla memorizzazione e cancellazione, verrà passato un parametro stringa contenente il token.

2.2 JAX-WS

JAX-WS (Java API for XML-Based Web Services)[SUN09a, EMO06, SMA06, JCP-WS, CAR-JWS] è il modello di programmazione di servizi Web di generazione successiva che completa le basi fornite dal modello di programmazione JAX-RPC (Java API for XML-based RPC). Con JAX-WS, lo sviluppo di Client e Web Service è semplificato grazie a una maggiore indipendenza della piattaforma per applicazioni Java con l'utilizzo di proxy dinamici e annotazioni Java.

JAX-WS è un modello di programmazione che semplifica lo sviluppo di applicazioni attraverso il supporto di un modello standard basato su annotazioni, per lo sviluppo di client e applicazioni di servizi web. La tecnologia JAX-WS si allinea in maniera strategica con l'attuale tendenza del settore verso un modello di messaggistica maggiormente incentrato sui

documenti, e sostituisce il modello di programmazione di chiamata di procedure remote definito da JAX-RPC. Sebbene il modello di programmazione JAX-RPC e le applicazioni siano ancora supportati dai vari applicativi software, questo modello ha delle limitazioni e non supporta diversi servizi basati sui documenti. In sostanza, JAX-WS è il modello di programmazione strategico per lo sviluppo di Web Service e rappresenta un elemento fondamentale del "Java Platform, Enterprise Edition 5 (Java EE 5)". JAX-WS è noto anche come JSR 224.

La versione odierna di JAX-WS è la 2.2; essa amplia le funzionalità di JAX-WS 2.0 fornendo il supporto per WS-Addressing in un' API standardizzata. Web Services Addressing specification 1.0 definisce un meccanismo per indirizzare Web Services e messaggi indipendentemente dal trasporto usato. Questa API può essere utilizzata per specificare gli URI (Uniform Resource Identifier) associati alle operazioni WSDL (Web Services Description Language) del proprio servizio web. JAX-WS 2.2 introduce inoltre il concetto di *funzioni,* come metodo per controllare in maniera programmatica modalità di funzionamento e funzioni specifiche. Le funzioni standard sono tre: Addressing (per WS-Addressing), MTOM (quando si ottimizza la trasmissione di allegati binari) ed infine, RespectBinding (per estensioni wsdl:binding).

JAX-WS 2.2 richiede JAXB (Java Architecture for XML Binding) Versione 2.2 per il bind dei dati.

L'implementazione dello standard di programmazione JAX-WS fornisce i seguenti miglioramenti per lo sviluppo di Web Service e Client:

- **Maggiore indipendenza delle piattaforme per applicazioni Java**.

  Con le API JAX-WS, lo sviluppo di Client e Web Service è semplificato grazie a una maggiore indipendenza delle piattaforme per applicazioni Java. JAX-WS, sfrutta i vantaggi del meccanismo proxy dinamico per fornire un modello di delega formale con un provider collegabile. Questo è un miglioramento rispetto a JAX-RPC, che invece si basa sulla generazione di stub di fornitori specifici per il richiamo.

- **Annotazioni**

  JAX-WS introduce il supporto per l' annotazione di classi Java con metadati per indicare che la classe Java è un servizio Web. JAX-WS supporta l'uso di annotazioni in base alla specifica Metadata Facility for the Java Programming Language (JSR 175), la specifica Web Services Metadata for the Java Platform (JSR 181) e le annotazioni definite dalla specifica JAX-WS 2.2. L' utilizzo di annotazioni semplifica lo sviluppo di servizi Web. Esse sono usate per definire le informazioni che in genere venivano specificate nei file dei descrittori di distribuzione, nei file WSDL o nei metadati di associazione dai file XML e WSDL nelle risorse di origine.

  È possibile, ad esempio, incorporare una semplice tag @WebService nell'origine Java per esporre il bean come servizio Web.

```
@WebService

public class QuoteBean implements StockQuote {

     public float getQuote(String sym) { ... }

}
```

L'annotazione @WebService indica all' ambiente di runtime del server, di utilizzare tutti i metodi pubblici sul bean come servizio web. I livelli di granularità aggiuntivi possono essere controllati aggiungendo altre annotazioni sui singoli metodi o parametri. L'utilizzo di annotazioni semplifica l' esposizione di risorse Java come servizi web.

L'utilizzo delle annotazioni migliora inoltre lo sviluppo dei servizi Web all'interno della struttura di un team in quanto non è necessario definire ogni servizio Web in un singolo descrittore di distribuzione come richiesto con i servizi Web JAX-RPC.

L'utilizzo delle annotazioni con i servizi Web JAX-WS consente lo sviluppo parallelo del servizio e dei metadati richiesti.

Per i servizi Web JAX-WS, l'utilizzo del descrittore di distribuzione webservices.xml è facoltativo poiché è possibile utilizzare le annotazioni per specificare tutte le informazioni contenute nel file descrittore di distribuzione. È possibile utilizzare il file del descrittore di distribuzione per aumentare o sovrascrivere le annotazioni JAX-WS esistenti.

Ad esempio, se la classe di implementazione del servizio per il proprio Web Service JAX-WS include quanto segue:

- l'annotazione @WebService:

```
@WebService(wsdlLocation=”http://miohost.
com/EsempioService.wsdl”)
```

- il file webservices.xml, che specifica un nome file differente per il documento WSDL, come di seguito indicato:

```
<webservices>

<webservice-description>

<webservice-description-name>
EsempioService
</webservice-description-name>
<wsdl-file>
    META-INF/wsdl/EsempioService.wsdl
</wsdl-file>

…

</webservice-description>

</webservices>
```

In questo caso, il valore specificato nel descrittore di distribuzione, META-INF/wsdl/EsempioService.wsdl, sovrascrive il valore dell'annotazione.

- **Richiamo dei servizi web asincroni**

Con JAX-WS, i Web Service vengono richiamati sia in maniera sincrona che in maniera asincrona. JAX-WS aggiunge il supporto per un meccanismo di polling e callback per il richiamo dei Web Service in maniera asincrona. Grazie a un modello di polling, un client può emettere una richiesta e ricevere una risposta emessa per determinare se il server ha risposto. Quando il server risponde, viene quindi richiamata la risposta reale. Con il modello di callback, il client fornisce un gestore callback per accettare ed elaborare la risposta in ingresso. Entrambi questi metodi consentono al client di continuare l'elaborazione senza dover attendere una risposta, fornendo un modello più dinamico ed efficace per richiamare i servizi web.

Ad esempio, un' interfaccia di servizi web potrebbe avere dei metodi sia per le richieste sincrone che per quelle asincrone. Le richieste asincrone sono riportate in grassetto nel seguente esempio:

```
@WebService
public interface CreditRatingService {
     // operazione sync
     Score getCreditScore(Customer customer);
     // operazione async, con polling
```

```
        Response<Score>
        getCreditScoreAsync(Customer customer);

        // operazione async, con callback

        Future<?> getCreditScoreAsync(Customer
        customer, AsyncHandler<Score> handler);

    }
```

Il richiamo asincrono che utilizza il meccanismo di callback richiede un ulteriore input dal programmatore del client. Il richiamato è un oggetto che contiene il codice dell' applicazione, che viene eseguito quando viene ricevuta una risposta asincrona.

Si usa il seguente codice di esempio per richiamare un gestore di callback asincrono:

```
    CreditRatingService svc = ...;

    Future<?> invocation =
    svc.getCreditScoreAsync(customerFred,
    new AsyncHandler<Score>(){
         public void handleResponse
         (Response<Score> response)
{
Score score = response.get();
// istruzioni da eseguire…
}
   }
    );
```

Si utilizza il seguente codice di esempio per richiamare un client di polling asincrono:

```
CreditRatingService svc = ...;

Response<Score> response =
svc.getCreditScoreAsync(customerFred);

while (!response.isDone()) {

// elaborazione

}
```

// Nessun cast necessario, grazie ai valori generici.

```
Score score = response.get();
```

- **Bind di dati con JAXB 2.2**

JAX-WS sfrutta gli strumenti e l'API JAXB (Java Architecture for XML Binding) 2.2 come tecnologia di bind per associazioni tra oggetti Java e documenti XML. La strumentazione JAX-WS si basa sulla strumentazione JAXB per il bind di dati predefiniti per associazioni bidirezionali tra oggetti Java e documenti XML. Il bind di dati JAXB sostituisce il bind di dati descritto dalla specifica JAX-RPC.

La specifica JAXB 2.2 fornisce perfezionamenti quali il miglioramento del supporto di compilazione e introduce il supporto per l'annotazione @XMLSeeAlso. Con il miglioramento del supporto di compilazione, attualmente è possibile controllare in maniera flessibile se un nuovo file XML Schema viene generato utilizzando il

generatore di schema, **schemagen,** ed è possibile configurare il compilatore schema **xjc**, in modo che non generi automaticamente nuove classi per uno schema particolare. È possibile utilizzare l'annotazione @XMLSeeAlso per accertarsi che JAXB riconosca tutte le classi incluse in una gerarchia di eredità per un'interfaccia di endpoint del servizio. JAXB verrà trattato approfonditamente più tardi.

- **Client statici e dinamici**

  L'API del client dinamico, per JAX-WS, è detta dispatch client (client di consegna), dal momento che andrà ad utilizzare l'interfaccia javax.xml.ws.Dispatch[BIS06, STH09, IBM-CWS, FUS08]. Tale dispatch client è un client orientato alla messaggistica XML. I dati vengono inviati in modalità PAYLOAD o MESSAGE (come vedremo in seguito). Il modello di programmazione del client statico per JAX-WS è detto roxy client. Esso richiama un servizio web basato su un' interfaccia SEI (Service Endpoint interface) che deve essere fornita.

- **Supporto per MTOM.**

  Grazie a JAX-WS, è possibile inviare allegati binari, come immagini o file, insieme alle richieste dei servizi web. JAX-WS aggiunge il supporto per la trasmissione ottimizzata di dati binari come specificato dal meccanismo MTOM (Message Transmission Optimization Mechanism).

- **Tecnologie di bind di dati.**

JAX-WS permette l' uso delle seguenti tecnologie di bind all'utente finale:

- XML Source;
- SOAP Attachments API for Java (SAAJ) 1.3;
- JAXB (Java Architecture for XML Binding) 2.2.

XML Source consente a un utente di inviare un javax.xml.transform.Source all' ambiente di runtime che rappresenta i dati in un oggetto Source da elaborare (cioè che deve essere inviato al runtime).

SAAJ 1.3, ora è in grado di inviare un intero documento SOAP attraverso l'interfaccia piuttosto che soltanto il payload. Tale azione viene eseguita grazie al client che invia l'oggetto SAAJ SOAPMessage attraverso l' interfaccia.

JAX-WS sfrutta il supporto JAXB 2.2 come tecnologia preferibile per il bind di dati tra Java e XML.

- **Supporto per SOAP 1.2**

Il supporto per SOAP 1.2 è stato aggiunto a JAX-WS 2.0. JAX-WS supporta sia SOAP 1.1 che SOAP 1.2, pertanto è possibile inviare allegati binari come immagini o file insieme alle richieste dei servizi Web.

- **Strumenti di sviluppo**

JAX-WS fornisce gli strumenti della riga comandi **wsgen** e **wsimport** per la generazione di risorse per i servizi web JAX-WS.

Quando si creano servizi web JAX-WS, è possibile iniziare sia con un file WSDL che con una classe di bean di implementazione.

- Se si inizia con una classe di bean di implementazione, si può utilizzare lo strumento **wsgen** per generare tutte le risorse server dei servizi web, compreso un file WSDL, se richiesto.
- Se si inizia con un file WSDL, utilizzare lo strumento **wsimport** per generare tutte le risorse dei servizi web per il server o il client. Lo strumento della riga comandi wsimport elabora il file WSDL con le definizioni di schema per generare le risorse portatili, che includono la classe di servizi, la classe dell'interfaccia dell'endpoint di servizi e le classi JAXB 2.2 per il corrispondente schema XML.

2.3 Breve approfondimento sulla libreria JAXB

L'acronimo JAXB sta a significare: Java Architecture for XML Binding. Fondamentalmente questa libreria mette a disposizione degli sviluppatori, tool e API per eseguire il mapping tra classi Java e file XML.

JAXB è dotato principalmente delle seguenti due caratteristiche[SBR09]:

1) l' abilità di fare marshal di oggetti Java, in documenti XML.
2) la possibilità di fare unmarshal di documenti XML, in oggetti Java.

Come da Figura 1.

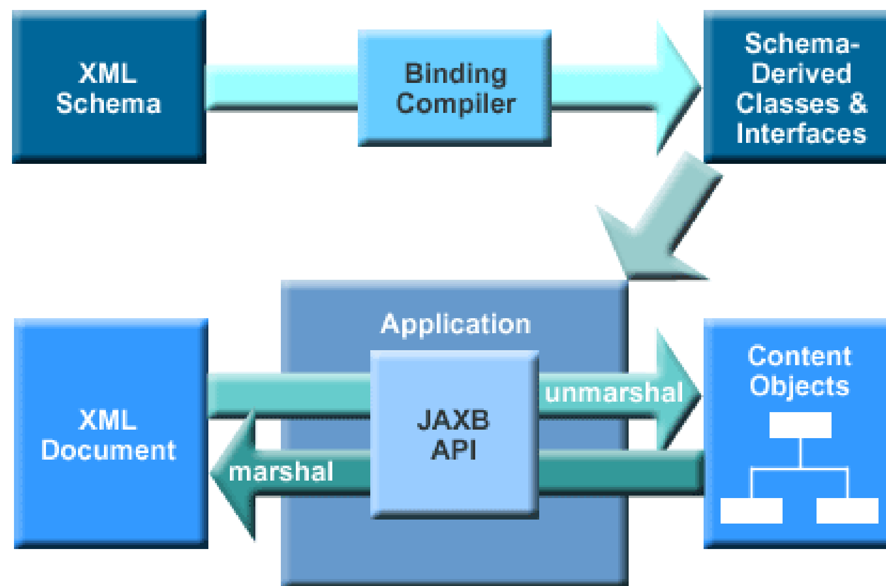

Figura 1 Caratteristiche di JAXB

In altre parole, JAXB permette di memorizzare e ritrovare dati all' interno di qualunque formato XML, senza il bisogno di effettuare il "parsing" e la generazione manuale dei documenti XML da Java. È quindi un prezioso strumento nel caso in cui si vogliano utilizzare esclusivamente oggetti Java generando poi l' XML relativo tramite "marshalling".

JAXB, insieme a SAX e a DOM fa parte di quelle tecnologie per la manipolazione di documenti XML, raccomandate da Sun; in particolare, JAXB, è più semplice da utilizzare rispetto a SAX, e può non richiedere così tanta memoria, quanta ne può richiedere, invece, DOM.

Ma andiamo con ordine. Come abbiamo potuto osservare dalla figura precedente, JAXB prevede diversi passaggi per la conversione da documento XML a oggetto java e viceversa. Si inizia obbligatoriamente con il binding dell' XML Schema per il documento XML. Quindi, a scelta, si effettua l'unmarshal del documento XML in un albero di oggetti Java, oppure il

viceversa di tale passo, ovvero il marshal. A seguire tratteremo, nell' ordine, questi 4 punti:

XML Schema → Classi JAVA (passo obbligatorio)

XML Schema ← Classi JAVA (non necessario)

2.3.1 XML-Schema → Classi JAVA

In questa sezione si tratterà il passaggio per effettuare il binding dell' XML-Schema in Classi Java. Per far ciò, si utilizzerà il wizard di NetBeans adibito a tale scopo.

Figura 2 – Da XML-Schema a Classi Java

Si crea un nuovo progetto, ad esempio una Java Application, ed al suo interno si mette, per comodità, il file schema relativo al file xml. Dopodiche si genera l' interfaccia ObjectFactory e si deducono le classi corrispondenti ai tipi che abbiamo definito all' interno dello schema.

Per far ciò, su netBeans, si selezionerà "New Files" → "XML" → "JAXB Binding". Nella form che compare, si immette un nome a scelta sotto il

campo “Binding Name” mentre sotto la voce “Schema File” si seleziona il nostro file xsd. Si termina cliccando su “Finish”.

Ora si passa alla vista “Projects” per cliccare sul progetto attualmente al lavoro. Espandere quindi il nodo. Si noti il sotto-nodo “JAXB Bindings”. Cliccandoci sopra con il tasto destro del mouse si nota la voce “Regenerate Java Code”. Nel caso in cui si voglia andare ad apportare modifiche al file xml-schema [W3CXS01], per aggiornare le classi generate, si deve cliccare la voce di cui sopra. Per il momento non c’ è bisogno di eseguirlo, dal momento che la prima volta che si effettua il binding è fatto in automatico. Per poter constatare i risultati inerenti il mapping tra “Classi Java – XML Schema”, ci si sposta alla vista “Files”, quindi espandere prima il nodo relativo al progetto e successivamente il sottonodo relativo a build. Si avranno le due directory:

- “classes” (per i *.class);
- “generated” (per i *.java).

In particolare, si andrà a dipanare una struttura ad albero come rappresentato in Figura 3.

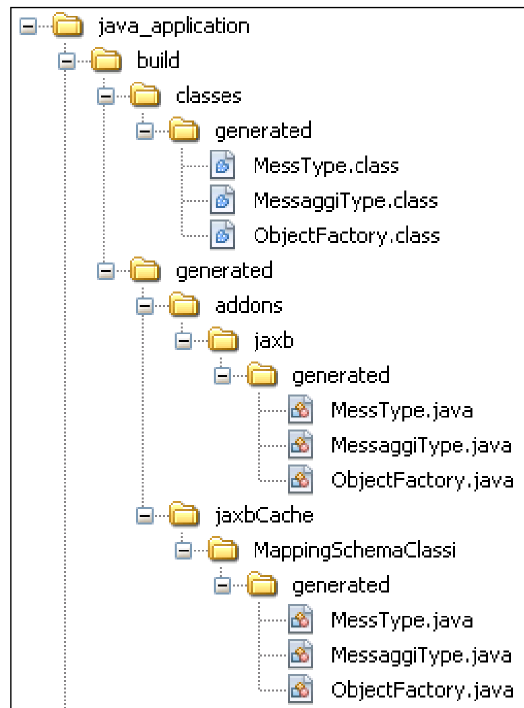

Figura 3

Confrontando le classi generate (Figura 3) con la struttura del file XML Schema oggetto di trasformazione, è possibile iniziare a comprendere con quale ragionamento venga eseguito il mapping. L' albero relativo alla struttura dell' XML Schema utilizzato è la seguente:

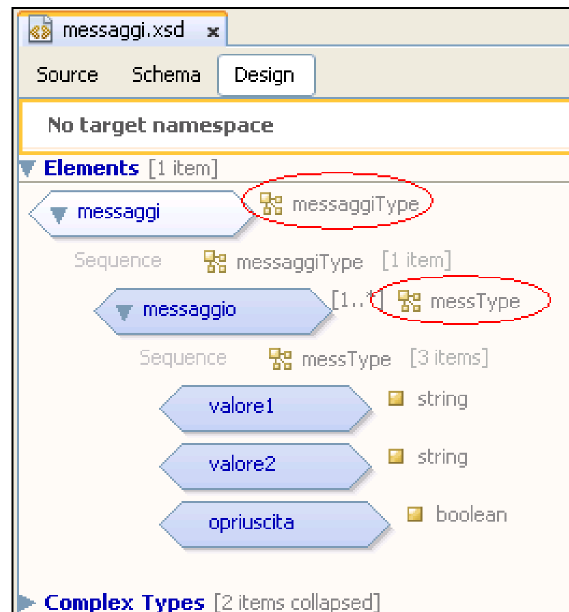

Figura 4

Sono stati cerchiati in rosso i due complex type, dai quali sono state generate le uniche due classi (si è preferito questo stile di visualizzazione dell' XML Schema, anziché riportarne il codice, per avere una più veloce e diretta comprensione della struttura ad albero).

Dalla Figura relativa all' albero dell' attuale progetto, vediamo la classe "ObjectFactory". Essa è una classe che viene generata di default, indipendentemente, quindi, dalla complessità del file Schema e contiene metodi per la generazione di istanze delle interfacce.

2.3.2 XML-Schema ← Classi JAVA

Con questo passaggio, si andrà, invece, ad effettuare il binding di una Classe Java in un XML-Schema. La procedura, viene descritta via codice. Si procede creando il metodo "generateSchema" che si preoccupererà di eseguire tutto il lavoro richiesto; ma andiamo con ordine:

```
JAXB_Progetto azioniXML = new JAXB_Progetto();
try {
   azioniXML.generateSchema(daMappare.Persona.class,
Config.getPATH_XMLSCHEMADIR());
} catch (Throwable e) {
   e.printStackTrace();
}
```

Il metodo di cui sopra, converte una classe in un file XML Schema e memorizza quest' ultimo in una data directory. La classe da specificare rappresenta quindi il primo parametro del metodo, mentre la

directory\stringa rappresenta il secondo parametro da inserire. Si vede di seguito il metodo nel dettaglio:

```
public void generateSchema(Class classe, String dir) throws Throwable {

final File dirBase = new File(dir);


class MySchemaOutputResolver extends SchemaOutputResolver {

public Result createOutput(String namespaceUri, String suggestedFileName)
throws IOException {

        return new StreamResult(new File(dirBase, suggestedFileName));

}

}


JAXBContext context = JAXBContext.newInstance(classe);

context.generateSchema(new MySchemaOutputResolver());

}
```

Il metodo generateSchema definito per la classe JAXBContext, serve a generare i documenti XML Schema per il presente Context. Come argomento, generateSchema, accetta un oggetto di tipo SchemaOutputResolver. L' implementazione della classe astratta "SchemaOutputResolver" deve essere provvista dall' applicazione chiamante al fine di generare l' XML Schema. Il metodo createOutput

decide dove il file schema (del dato namespace URI) sarà scritto, e lo ritorna come un oggetto di tipo Result.

Volendo mostrare un esempio di ciò che si può ottenere con questo metodo, è stata creata una classe “Persona”, all’ interno del package “daMappare”:

```
package daMappare;

public class Persona {
   public String nome;
   public String cognome;
   public int eta;
   public boolean sposato;
}
```

Lo schema corrispondente\generato è il seguente:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:complexType name="persona">
      <xs:sequence>
         <xs:element name="nome" type="xs:string" minOccurs="0"/>
         <xs:element name="cognome" type="xs:string"
minOccurs="0"/>
         <xs:element name="eta" type="xs:int"/>
         <xs:element name="sposato" type="xs:boolean"/>
      </xs:sequence>
   </xs:complexType>
</xs:schema>
```

Capitolo 3 Gestione di siti web tramite wiki e compilazione

3.1 Introduzione (analisi e specifiche del problema)

Durante questa tesi ho cercato di sviluppare ulteriormente il lavoro svolto da Gessica Valbonesi, la quale si era basata sulla tecnologia wiki per modificare pagine web direttamente online, sfruttandola per gestire non solo codice HTML puro, ma anche codice CSS, codice JS non incluso nelle pagine, JSP non raggiungibili o mal funzionanti, file di configurazione, servlet e librerie. Tutto questo era reso possibile grazie allo sviluppo di un modulo di gestione costituito da un bottone Modifica Pagina che realizzava la funzionalità wiki e da un sistema di navigazione del file system che permetteva di caricare, rimuovere, rinominare, modificare file di qualunque natura e compilare file java. In questa prima parte della tesi si è estesa la funzionalità di compilazione per le classi che utilizzano delle librerie. Inoltre si è migliorata la struttura del modulo realizzando una unificazione delle due funzionalità principali di gestione ed editing (creazione di due jsp unificate) ed estendendo ognuna di tali funzionalità alla possibilità di correzione dei progetti d'esame (introdotta in questa tesi) e alla modifica dei web service.

Prima alcune parti del funzionamento del sito infatti, erano realizzate con i web service, ma si assumeva che questi ultimi risiedessero sullo stesso computer del sito e quindi che fossero accessibili/modificabili direttamente tramite il file system. In questa tesi invece si abbandona questa assunzione e come vedremo si utilizzano operation dei web service per modificarli da remoto (funzionalità di auto modifica).

In questo capitolo verranno illustrate in dettaglio le modifiche apportate al sito, l'ottimizzazione della compilazione online e la realizzazione della gestione unificata.

I punti fondamentali sono:

- Aggiungere parametri al comando di compilazione in modo tale da riuscire a compilare classi che utilizzano librerie;
- Creazione si un sistema che permetta di visualizzare l'errore di compilazione;
- Creazione di un sistema di gestione unificata, portando ordine al sito e al lavoro fatto precedentemente;
- Aggiunta della funzionalità per modificare anche i compiti online;
- Integrazione della funzionalità di "auto modifica" dei web service;

3.2 Compilazione online

Durante lo sviluppo della mia tesi ho cercato di migliorare il funzionamento del bottone "Compila", associato ad alcuni tipi di file, poiché prima era molto semplice, infatti non erano utilizzate le librerie per far si che compilasse anche un file ".java" e non veniva visualizzato l'errore di compilazione. Per cui ho aggiunto tutte le librerie, per poter appunto compilare un file di questo tipo, mettendole dentro una directory e aggiungendo una parte di codice alla pagina di gestione.

Quando un utente preme il pulsante "Compila" all'interno della pagina di gestione, viene richiamata la pagina stessa passandogli però un parametro

che ci aiuterà ad identificare il fatto che dovremo eseguire anche il codice di compilazione.

Bottone Compila

Figura 5 Bottone Compila

Nella parte della pagina in cui risiede il codice html ho aggiunto un frammento di codice java all'interno di un tag "pre", che a suo volta risiede dentro ad un "if", il quale controlla il parametro che indica se deve essere fatta o meno la compilazione, per cui se è stato premuto il tasto "Compila"; in questo modo avremo che la finestra con gli errori di compilazione comparirà nella pagina solo quando è necessario.

Per poter ricompilare una servlet e altre classi è stato necessario importare all'interno della pagina jsp "gestione.jsp" la classe sun.tools.javac.Main situato in jdk1.7.0_03/lib/tools.jar che permetterà l'utilizzo del comando main.compile() che si occuperà di compilare la servlet/classe desiderata. Inoltre, per far funzionare il metodo main.compile() bisogna passargli un argomento che specifichi dove andare a trovare tutte le librerie delle servlet o della classe che si intendono compilare, nel nostro caso delle servlet è stato inserito il percorso della libreria servlet-api.jar.

```
<%@page import="com.sun.tools.javac.Main" %>
```

Per poter quindi compilare classi che usano librerie si deve aggiungere al comando di compilazione la libreria servlet-api.jar per poter ricompilare le servlet, come avevamo già detto, inoltre devono essere aggiunte anche tutte

le librerie usate dal context in cui si effettua la compilazione, le quali sono collocate all'interno della directory WEB-INF/lib/.

Analizziamo ora il codice vero e proprio di compilazione nel caso in cui dobbiamo compilare un file ".java". Inizialmente ho preso tutte le librerie che sono dentro alla directory e le ho messe dentro un file e successivamente grazie al comando "path.list()" le ho inserite all'interno di un array di stringhe e infine le ho riportate concatenandole in una stringa "librerie" grazie ad un ciclo "for". Successivamente ho poi inserito dentro l'array di stringhe "optionAndSources" tutti i parametri e i percorsi utili per poter effettuare la compilazione, inserendo anche la mia variabile "librerie".

Infine ho effettuato il comando "Main.compile()" passandogli come parametri l'array di stringhe appena enunciato e un "PrintWriter" che mi permette la stampa dell'errore.

Come si è intuito è stata introdotta anche la stampa dell'errore, a differenza del codice precedente, poiché si è pensato che se la compilazione dovesse dare errore sarebbe utile poter vedere il motivo e appunto quale tipo di errore si è verificato; altrimenti, per come era strutturato prima il codice, dopo avere visto che la compilazione dava errore, l'utente sarebbe dovuto uscire e compilare il codice da riga di comando per capire qual'era l'errore effettivo. In questo modo quindi si è facilitato molto il lavoro di compilazione e di correzione degli errori da parte dell'utente.

In questa immagine mostro il codice appena descritto:

```
 <%
    if (request.getParameter("compila") != null) {
       File tocompile = new File(request.getParameter("compila"));
%>
```

```
<pre class="comp">
```

```
<%
   File path = new File(Settaggi.getPATH_BASE()+"WEB-INF/lib/");
   String[] list;
   String librerie = new String();
   list = path.list();
   for(int i = 0; i < list.length; i++){
        librerie = librerie + Settaggi.getSEPARATORE() +
             Settaggi.getPATH_BASE() + "WEB-INF/lib/" + list[i];
   }
```

```
String[] optionsAndSources =
      {"-classpath",Settaggi.getPATH_BASE()+"WEB-INF/classes/"+
       Settaggi.getSEPARATORE()+Settaggi.getPATH_BASE()+"servlet-api.jar"+
       librerie,request.getParameter("compila")};
```

```
int status =  Main.compile( optionsAndSources, new PrintWriter(out));
```

Il comando “Main.compile” mi restituisce un intero, che vale 0 se la compilazione è avvenuta correttamente oppure 1 se ci sono stati degli errori. Sfruttando quest’ultimo valore riesco a far visualizzare all’utente anche una stringa indicante se il tutto è avvenuto con successo o meno, come possiamo vedere nel codice sottostante.

```
if(status==0)
{
    successo = true;
    messaggioSuccesso = "File\""+tocompile.getName()+
                        "\"  compilato correttamente";
}
else
 {
     errore = true;
     messaggioErrore = "File \""+tocompile.getName()+
                         "\" non Compilato!!  "+
                         "status: "+status;
 }
```

```
</pre>
<%
      }
%>
<% if (errore) {%>
  <p class="error"><strong>Errore:</strong> <%= messaggioErrore %> </p>
<% }%>
<% if (successo) {%>
  <p class="success"><strong>Successo:</strong> <%= messaggioSuccesso %></p>
<% }%>
```

L’immagine sottostante mostra cosa verrà visualizzato dall’utente in caso di errore nella compilazione:

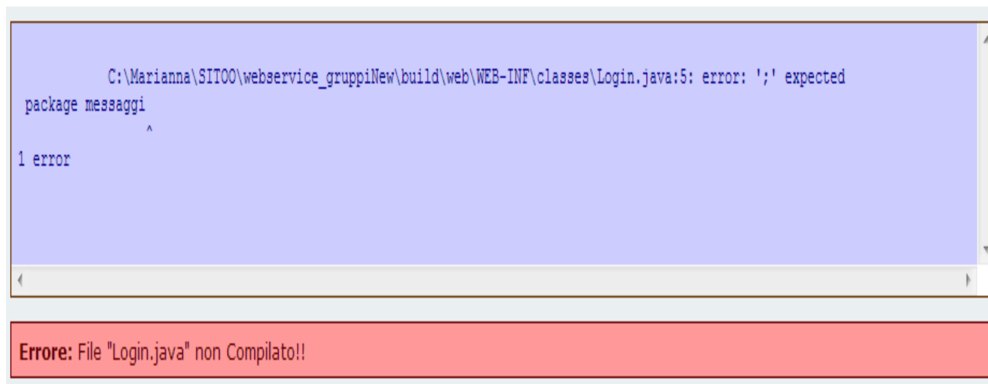

Figura 6 Errore di compilazione

Successivamente abbiamo notato che, siccome l’errore di compilazione indica il numero di riga in cui questo è stato rilevato, sarebbe stato utile poter visualizzare il codice che ha dato errore affiancandogli però i numeri di riga.

Per far questo è stato aggiunto, assieme al bottone di compilazione, un bottone “VediNum” all’interno della pagina che naviga nel file system, il quale mi permette di aprire una pagina che è di sola visualizzazione (non è quindi possibile fare modifiche), grazie alla quale si può individuare precisamente qual è e dov’è l’errore.

Bottone Visualizzazione numerazione

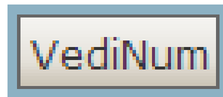

Figura 7 Bottone VediNum

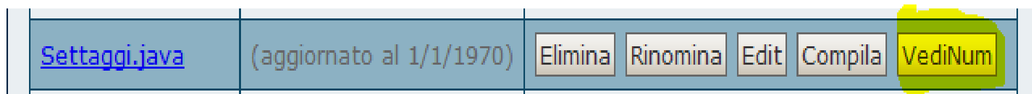

Figura 8

E’ stata quindi creata una jsp con la stessa struttura di “EditUnico.jsp”, che permette di visualizzare il codice, omettendo però tutta la parte relativa alla modifica e al salvataggio di file e inserendo un comando che permette di affiancare i numeri di riga al codice visualizzato nella textarea.

Per la visualizzazione dei numeri di riga è stato utilizzato un “LineNumberReader”, grazie al quale è stato possibile utilizzare il metodo “getLineNumber()”, il quale restituisce appunto il numero della riga in questione.

```
LineNumberReader input = new LineNumberReader(new FileReader(sorgente));
```

```
contenuto =  contenuto.concat(input.getLineNumber()+ ": "+line) + System.getProperty("line.separator");
```

Ottenendo così questo tipo di output all’interno della textarea:

Contenuto della pagina

```
1: package settaggi;
2:
3: import java.io.*; // InputStream, OutputStream
4: import org.w3c.dom.*; // Document
5: import javax.xml.transform.*; // TransformerFactory
6: import javax.xml.transform.dom.*; // DOMSource
7: import javax.xml.transform.stream.*; // StreamResult
8: import javaxxml.parsers.*; // DocumentBuilderFactory, DocumentBuilder
9:
10: /**
11:  * @author  francesco.sbrighi extends Menx
12:  */
13: public class Settaggi {
14:
15:     //public static final int NUMERO_CANALI = 8;
16:     //public static final int PORT_FORUM = 2050; //2051 successivamen
17:     //public static final int PORT_WS = 2039;
18:
19:     private static final String PATH_BASE = "C:/Marianna/SITOO/sitoTW
Settings/pavelo/Desktop/sito/public_webapp/";  //"C:/Documents and
```

Figura 9 Textarea con numerazione

3.3 Gestione unificata

3.3.1 **Idea di gestione unificata**

Dopo varie modifiche apportate al sito si è notato che c'erano alcune pagine di gestione che erano molto simili tra loro, tra cui "gestioneClassiLibrerie", "gestioneCompiti" e "gestioneRisorse" , le quali avevano tutte la stessa struttura, ma visualizzavano risorse diverse. Per questo motivo si è pensato di unificarle in un'unica jsp chiamata "gestione" che, grazie ad alcuni parametri settati differentemente per ogni pagina precedentemente esistente e utilizzando alcuni "if", riesce a fare tutto ciò che facevano prima le singole pagine separate.

Tutto questo ha reso una manutenzione molto più semplice, poiché per modificare una singola cosa precedentemente era necessario accedere ad ogni pagina diversa e apportare le modifiche a ciascuna, mentre ora basta accedere alla singola pagina di gestione per poter visualizzare tutto il codice utile ed effettuare dei cambiamenti. In questo modo è stato anche più facile inserire in ogni tabella, che permette di navigare nel file system e mostra tutte le risorse presenti, una prima riga ".." che ti consente di tornare indietro nel percorso attuale (path).

Oltre alle pagine di gestione erano presenti anche due pagine di modifica ("editClassi" ed "editCompiti"), le quali avevano la stessa identica struttura, l'unica differenza era che venivano richiamate da due jsp differenti e quindi per risorse differenti.

Per cui, anche in questo caso, è stato possibile unificarle in un'unica jsp chiamata "editUnico", utilizzando sempre i parametri che avevamo usato per le pagine di gestione.

Prima di entrare nel dettaglio della realizzazione dell'unificazione della gestione e dell'edit si presenta l'estensione introdotta con questa tesi relativa alla gestione dei compiti.

3.3.2 Compiti

Durante lo sviluppo del lavoro si è potuto notare che per l'amministratore (il professore) era molto scomoda la correzione dei compiti, poiché dovevano essere scaricati tutti i compiti consegnati dagli studenti e aperti con NetBeans (o con altri programmi) per poter correggere anche solo piccole parti. Per questo motivo si è arrivati ad una conclusione: perché non utilizzare lo stesso meccanismo utilizzato per la modifica del codice del sito anche per i compiti, facilitando così in modo rilevante la correzione e la visualizzazione del codice di essi?

Così abbiamo creato una pagina, come quella di gestione dove si possono vedere tutti i compiti (per default quelli dell'ultimo appello) e dove si posso modificare delle parti direttamente online dal sito.

- Gestione compiti

Figura 10 Link della pagina di gestione per i compiti

Per questo motivo ho creato una pagina "GestioneCompiti"(che come ho già detto precedentemente verrà unificata con le altre pagine di gestione), dove viene visualizzato l'elenco di tutti i compiti consegnati nell'ultimo appello e vicino ad ogni file ho inserito un bottone di edit, che rimanda alla stessa pagina di modifica degli altri file, il quale permette appunto di fare modifiche e visualizzare il codice dei compiti.

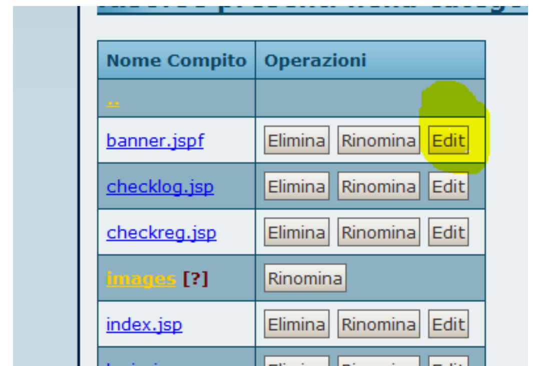

Figura 11 Visualizzazione lista compiti

Figura 12 Elenco file in un compito

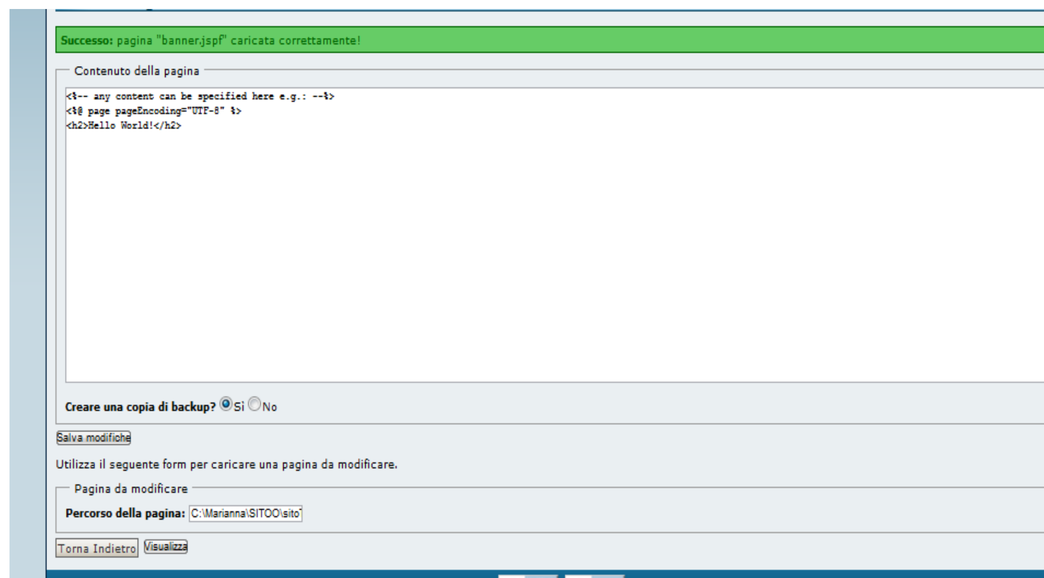

Figura 13 Pagina di editing dei compiti

Unificazione della gestione

Analizziamo ora nel dettaglio il codice utilizzato per unificare le pagine di gestione dei file (vedi appendice punto A).

Per prima cosa è stato creato un parametro “ris”, che viene passato quando viene chiamata la pagina di gestione, il quale assume un valore diverso per ogni pagina preesistente, quindi per ogni tipo di risorsa:

- viene impostato a “compiti”, quando desideriamo visualizzare la directory dei compiti consegnati dagli studenti, in questo caso è stato necessario anche un altro parametro ”corso” il quale identifica se stiamo considerando la parte di tecnologie web o quella di tecnologie web avanzate;

- viene impostato a “r1” quando si vuole visualizzare la directory del materiale didattico(quindi dei lucidi) e anche in questo caso è stato necessario utilizzare il parametro “corso” per capire quale materiale era necessario visualizzare;

- viene impostato a “classlib” quando si vuole accedere al codice vero e proprio delle pagine del sito e alle sue classi.

Una volta passati questi parametri sarà la jsp di gestione che effettuerà dei controlli per capire che parte di codice dovrà effettuare o meno, tutto questo è stato reso possibile grazie a vari “if” iniziali, che permettono di settare il path base (“destPath”), cioè il percorso che dovremo utilizzare per tutte le operazioni successive e il nome della risorsa con cui abbiamo a che fare.

Successivamente tutto il resto del codice è rimasto invariato poiché sono state utilizzate tutte le variabili che abbiamo inizializzato dentro agli if iniziali, senza dover apportare altre modifiche sostanziose al codice originale.

Unificazione dell’editing

Anche per questa unificazione sono stati utilizzati i parametri illustrati precedentemente, facilitando in questo modo la sua realizzazione e solo nella parte della visualizzazione, nell’html, è stato necessario fare alcune distinzioni, poiché non tutte le risorse erano trattate e visualizzate nello stesso modo.

Per prima cosa ogni pagina di editing possedeva una form diversa per spostarsi da una directory all'altra, ovviamente inerente alla risorsa iniziale, inoltre anche nella tabella che visualizza le vere e proprie risorse ci sarebbero state parti in più che non sarebbero servite per alcune risorse se avessimo lasciato lo stesso identico codice per tutte (per esempio la colonna per visualizzare la data dell'ultimo aggiornamento).

Quest'ultimo problema è stato facilmente risolto utilizzando i parametri iniziali per fare degli if ed ottenere così una visualizzazione più ordinata e coerente alle risorse da visualizzare e all'organizzazione precedente.

3.3.3 **WebService**

Come vedremo nel capito successivo l'idea di gestione unificata è stata estesa anche per la chiamata alle operation dei web service manager, che consentono di effettuare le varie operazioni di gestione,compilazione,ecc… da remoto sui web service del sito. Estensione sia per quanto riguarda la pagina unificata gestione dove andremo a chiamare web service per i vari bottoni, sia per la pagina di edit/visualizzazione dove recupereremo il codice da visualizzare tramite web service e spediremo il codice da modificare ai web service.

Dopo questa breve introduzione al concetto base di questo nuovo meccanismo sviluppato, nel capitolo successivo illustreremo passo passo ciò che è stato fatto e come è stato possibile farlo.

Capitolo 4 Gestione remota di Web Service

In questo capitolo introduco il nucleo della mia tesi, spiegando l'idea base e le motivazioni del perché è stato fatto il lavoro. L'idea è nata pensando ad un ulteriore sviluppo del sito web, molto utile per tutti quei siti realizzati tramite web services, e consiste nell'espandere il concetto di modificare e ricompilare, non solo le servlet ma anche il codice dei web service.

Come già spiegato, prima alcune parti del funzionamento del sito erano realizzate con i web service, ma si assumeva che questi ultimi risiedessero sullo stesso computer del sito e quindi che fossero accessibili/modificabili direttamente tramite il file system. In questa tesi invece si abbandona questa assunzione e come vedremo si utilizzano operation dei web service per modificarli da remoto.
I web service sono oramai un parte fondamentale per lo sviluppo sul web. Allora perché non creare un sistema che permetta a questi ultimi di auto modificarsi utilizzando la struttura già presente nel sito ma inserendo il codice di (auto)modifica direttamente all'interno di essi? Questo porterebbe ad una forte semplificazione da parte dell'amministrazione e della gestione del sito, in quanto non ci si dovrebbe più preoccupare della locazione vera e propria dei web service e di come poter interagire con loro per poterli modificare.

Una possibile soluzione a questo problema è quella di creare web service dotati di un meccanismo di "reflection", cioè in grado di mandare il sorgente del proprio codice, per poter essere modificato dall'utente tramite l'edit, e in grado di ricevere un nuovo sorgente per poi auto modificarsi. Questo nuovo meccanismo permette di gestire i web service (per esempio modificarli,ricompilarli,…) indipendentemente (sia fisicamente che logicamente) dalla locazione della pagina che li richiama, ma nello stesso tempo porta alla possibilità di svilupparli mentre li si utilizza.

Questa idea è stata sviluppata applicandola sempre al sito di Tecnologie Web, il quale utilizza cinque web service sviluppati i cinque progetti distinti tra loro e ad ognuno è associato un certo servizio offerto al sito. Lo scopo di tutto questo lavoro per i web service è stato quello di ottenere un meccanismo non statico che permetta l'auto modifica di questi ultimi senza che necessariamente dipendano totalmente dalla pagina che li richiama, in modo tale che offrendo loro stessi le operazioni che possono essere richiamate ottengano indipendenza e sicurezza a livello di codice.

Elenchiamo ora i web service con cui abbiamo lavorato, per capire meglio di cosa parleremo:

- "webservice_admin", il quale gestisce la parte amministrativa del sito (per esempio la login dell'utente amministratore);

- "webservice_avvisi", che gestisce gli avvisi inseriti all'interno del sito;

- "webservice_gruppiNew", web service che permette la gestione dei gruppi di tecnologie web;

- "webservice_gruppiTW", web service che gestisce i gruppi di tecnologie web avanzate;

- "webservice_progetti", contenente le operation che permettono la gestione dei progetti.

I web service sono stati realizzati con JAX-WS per cui si usa il binding tra classi java e xml-schema, in particolare la metodologia utilizzata è quella di sviluppare la classe per il tipo di parametri e dei valori di ritorno in java e poi far creare l'xml-schema a NetBeans.

Durante questo capitolo verrà affrontata l'implementazione specifica che è stata utilizzata per creare il "modulo" all'interno del sito di Tecnologie Web. Nei seguenti paragrafi si potrà visionare parte di codice da me sviluppato, così da rendere più completa possibile la mia trattazione.

Ho voluto suddividere questo capito il due parti fondamentali, perché sia più chiaro e ordinato il lavoro svolto: nella prima parte ho descritto ciò che ho fatto lato client, quindi tutte le chiamate dalle jsp e i parametri passati; mentre nella seconda parte ho descritto il funzionamento dei web service, quindi tutto ciò che è stato fatto a livello server, descrivendo tutte le operation create e come sono state inserite e utilizzate nei web service.

4.1 Lato Client

Come è già stato accennato in precedenza in questo sotto capitolo verrà descritta dettagliatamente la struttura a lato client e quindi a livello di pagine jsp.

La jsp principale che ho dovuto modificare è stata quella di gestione (che avevamo precedentemente unificato), la quale dovrà richiedere al web service stesso la directory che dovrà visualizzare nella tabella che elenca i file, sui quali successivamente potranno essere effettuate varie operazioni.

Entriamo ora nel dettaglio del codice, innanzitutto è stato necessario introdurre un nuovo parametro che viene passato ogni qual volta che si richiama la pagina di gestione. Questo parametro è chiamato "wsname", il quale sarà settato a "no" quando non stiamo lavorando con i web service, altrimenti dentro a esso ci sarà il nome del web service che ci interessa.

Per rendere più facili delle operazioni successive legate ai percorsi da settare si è deciso di inserire all'interno del parametro l'ultima parte del nome di essi per identificarli, avendo notato che la prima parte era identica per tutti. Per esempio per "webservice_admin" il parametro varrà "admin", per "webservice_progetti" varrà "progetti" e così via anche per gli altri.

4.1.1 Gestione

La pagina di gestione, come è già stato detto, è stata strutturata in modo tale che all'inizio vengano settati alcune variabili, che identificano il path e il nome delle risorse che ci interessano.Per questo motivo è stato necessario introdurre un if iniziale che controlli se il parametro "wsname" valga "no" o meno, in modo tale che se al suo interno c'è una stringa contenente il nome di un web service nelle variabili vengono inseriti i vari percorsi relativi per accedere ai web service. Concentriamoci meglio su come sono state settate queste variabili nel caso dei web service: nel percorso di default (cioè quello iniziale quando si apre la pagina per la prima volta, senza aver navigato all'interno dei file della pagina o dentro alla sua form per la selezione di risorse differenti da quelle di default) viene inserito l'indirizzo relativo "/WEB-INF/classes", che andrà successivamente completato dal web service stesso, in base all'indirizzo assoluto di quest'ultimo; questo grazie alla classe "Settaggi.java" presente all'interno di ogni web service (ma tutto questo verrà approfondito successivamente quando parleremo del codice lato server). Questo percorso ci permette di entrate nella directory contenente i file con il codice del web service trattato.

Oltre a questo percorso di default è stato necessario distinguere tutti i diversi casi che si potessero verificare nel caso in cui venisse utilizzata la form per navigare all'interno del web service considerato.

Elenchiamo ora tutti i campi della form e i percorsi a cui rimandano:

-classi, che denota appunto il percorso di default;

-librerie, che indica il percorso "/WEB-INF/lib" che è la directory contenente tutte le sue librerie;

-configurazione, il quale riporta al percorso "/WEB-INF";

-directory radice, che come dice il nome stesso rimanda alla alla cartella principale;

-altra directory, infine grazie a questo si ha la possibilità di indicare un altro percorso a scelta per mezzo di una label.

La figura sottostante riporta la form sopra descritta.

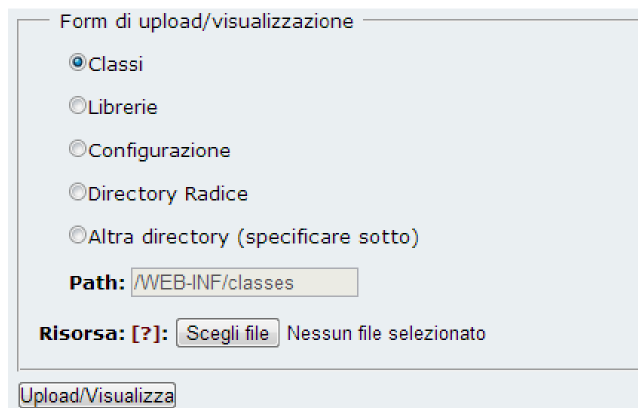

Figura 14 Form di navigazione

Analizziamo ora il codice che ha permesso la visualizzazione dei web service; anche in questo caso è stato necessario effettuare inizialmente un controllo per capire se si trattava di un web service o meno.

Nel caso stessimo lavorando con un web service, è necessario chiedere al web service stesso la lista dei file da visualizzare nella tabella, richiamando un sua

operation “getDir”, la quale sarà vista nel dettaglio successivamente, ora ci soffermiamo a capire cosa riceve in input dalla jsp e cosa le restituisce. Passando a questa operation semplicemente il percorso settato inizialmente nella variabile destPath e il valore del parametro “wsname”, otterremo un oggetto “lista” di tipo ArrayString che è una classe creata da me contenente un array di stringhe, per facilitare il passaggio di questo tipo di oggetto.

```
ArrayList<File> risorse = new ArrayList<File>();
List<String> listadir = new ArrayList<String>();
ArrayString lista = new ArrayString();
if (!wsname.equals("no")){
     lista = WSCall.getDir(wsname,destPath);
    try{
     listadir = lista.getI();
     listadir.toArray();
    } catch (Exception ex) {
     // TODO handle custom exceptions here
}
```

Successivamente inserendo ogni elemento di questo array in una lista di stringhe è possibile leggere ogni nome di file singolarmente e inserirli quindi nelle righe della tabella di visualizzazione, facendo gli opportuni controlli sul fatto che sia una directory o meno il file che stiamo inserendo e, in base a quale estensione ha, decidere quali operazioni rendere possibili su di esso.

Analizziamo ora dettagliatamente quest’ultimo concetto che è stato espresso in modo molto semplificato e conciso per dare prima un idea di cosa si dovrà ottenere. Per prima cosa controllando la lunghezza della lista si può capire se sono presenti o meno risorse, quindi grazie ad un semplice if possiamo effettuare questo controllo e avvertire l’utente nel caso non ci siano risorse presenti. Successivamente è stato necessario introdurre un ciclo while che permette di scorrere tutta la lista finché non risulta vuota e in ogni iterazione considerare un solo elemento della lista ottenendo il nome del file sotto forma di stringa. Sempre all’interno del ciclo, ottenuto il nome del file, è stato reso possibile effettuare tutti

i vari controlli per capire se si tratta di una directory o meno e, nel caso non lo sia, capire di che tipo è il file trattato (grazie appunto alla sua estensione).

Bottone Compila

Figura 15 Bottone Compila

Infine sempre all'interno della jsp di gestione è presente il codice di compilazione di un file, poiché accanto a tutti i .java è presente un bottone "Compila", che richiama la pagina stessa indicando però, grazie ad un parametro che viene passato, che è necessario effettuare la compilazione di un determinato file.

Anche in questo caso si è fatto uso dei web service, cioè è stata richiamata l'operation "compile", la quale passandogli semplicemente il destPath e il valore del parametro "wsname" restituisce una stringa contenente un valore che indica se è avvenuto un errore o meno, e l'errore stesso.

```
if (!wsname.equals("no")){                                          /
    String risultato="";
    String nomeFile = tocompile.getName();
  risultato = WSCall.compile(wsname,nomeFile,destPath);
```

```
if(risultato.startsWith("OK")){
    successo = true;
    messaggioSuccesso = "File\"" + nomeFile + "\"  compilato correttamente";
    out.println(risultato.substring(risultato.indexOf(';')+1));
}
```

```
if(risultato.startsWith("ERRORE")){
      errore = true;
      messaggioErrore = "File \"" + tocompile.getName() + "\" non Compilato!!" ;
      out.println(risultato.substring(risultato.indexOf(';')+1));
}
```

L'immagine appena riportata contiene il frammento di codice in cui viene appunto chiamata l'operation del web service, inoltre illustra anche che grazie al comando ".startWith" è possibile capire se la compilazione ha dato errore o meno e grazie al comando ".substring" è possibile stampare solo l'errore tralasciando i caratteri che vengono utilizzati per capire l'esito della compilazione.

4.1.2 Editing

Oltre alla pagina di gestione sono state apportate delle modifiche anche ad altre due jsp: "EditUnico.jsp" e "VisualizzaNumerazione.jsp".

Figura 16 Bottoni edit e vedi numerazione

Analizziamo nel dettaglio la jsp EditUnico(vedi appendice punto B), che equivale alla modifica pagina ed è necessaria per poter visualizzare il codice di un file, caricarlo e salvare le modifiche apportategli. Ovviamente anche in questo caso le operazioni appena elencate sono state delegate al web service stesso, richiamando due sue operation.

In particolare nel caso si voglia visualizzare il codice di una determinata risorsa verrà chiamata l'operation "getFile", la quale prendendo in input il percorso della risorsa da visualizzare e il valore del parametro "wsname", restituisce una stringa contenente il codice della risorsa desiderata.

```
contenuto = WSCall.getFile(wsname, percorsoPagina);
```

Inoltre è anche necessario salvare il nuovo file nel caso vengano apportate delle modifiche al codice: in questo caso si utilizza l'operation "postFile", la quale

ricevendo in input il codice modificato, il percorso del file e il valore del parametro “wsname”, restituisce una stringa che ci permette di capire se le modiche apportate sono state caricate correttamente o meno.

```
String carica = WSCall.postFile(wsname, sorgente, percorsoPagina);
```

Infine consideriamo l’ultima jsp in cui è stato necessario apportare delle modifiche, parliamo di “VisualizzaNumerazione”, cioè della jsp che viene utilizzata per vedere il codice di una risorsa affiancando ad esso i numeri di riga, senza aver la possibilità di apportare modifiche.

Anche in questo, caso come in quello precedente, per la visualizzazione del codice viene chiamata l’operation “getFile”, ma a differenza di prima dovremo fare un passaggio successivo per inserire i numeri di riga.

```
String nomefile="";
contenuto ="";
String contenuto2="";
String line = "";
contenuto2 = WSCall.getFile(wsname,  percorsoPagina);
LineNumberReader input = new LineNumberReader(new StringReader(contenuto2));
while ((line = input.readLine()) != null){
        contenuto = contenuto.concat(input.getLineNumber()+ ": "+line)
                    + System.getProperty("line.separator");
}
```

Come possiamo vedere da questo frammento di codice, dopo aver ricevuto la stringa con al suo interno il contenuto della risorsa, dovremo utilizzare un LineNumberReader e un ciclio while che ci permetto il concatenamento del contenuto, affiancando ad ogni sua riga il numero corrispondente, quest’ultimo passaggio avviene grazie al metodo “getLineNumber()”.

Mostriamo ora come vengono visualizzate le pagine “EditUnico.jsp” e “VisualizzaNumerazione.jsp”:

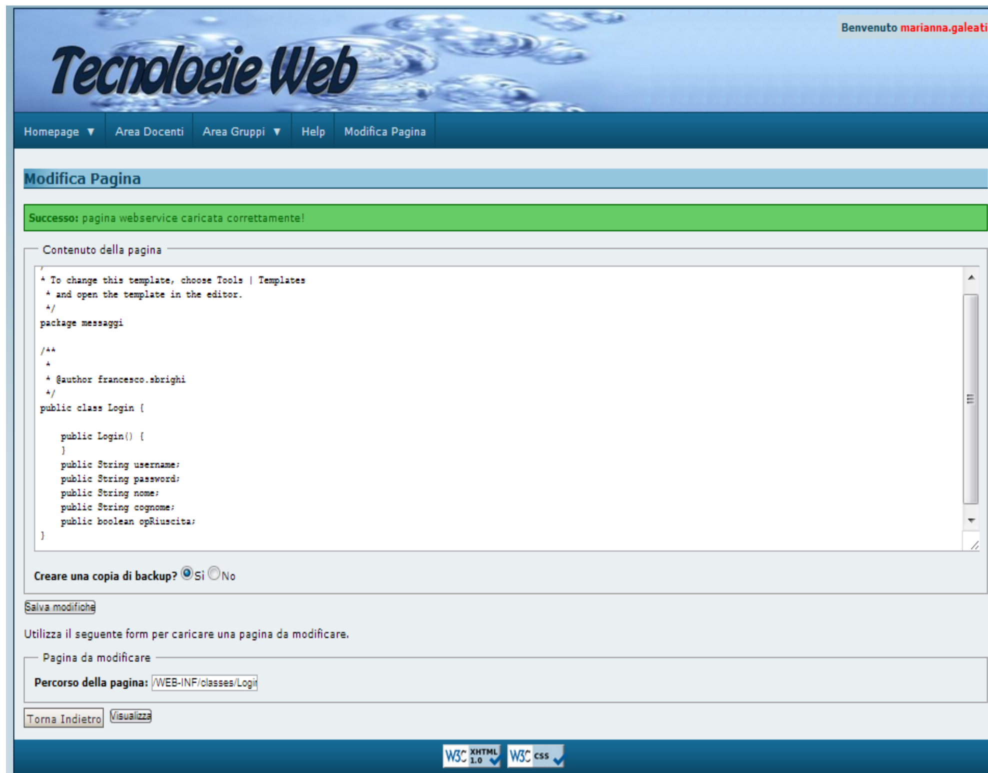

Figura 17 Pagina d’editing

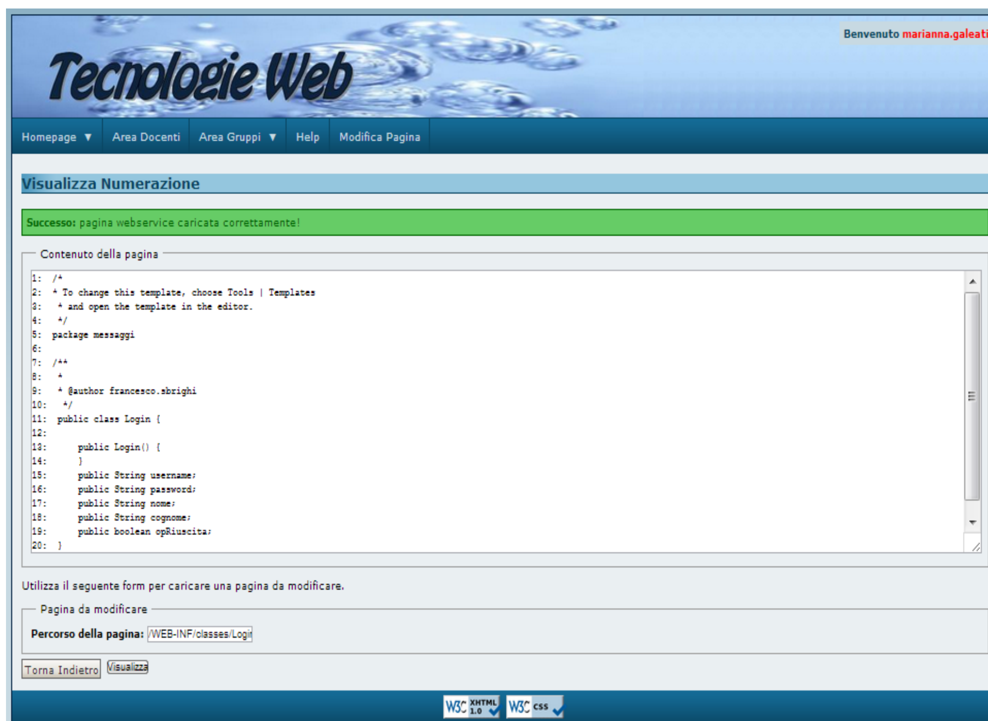

Figura 18 Pagina per visualizzare i numeri di riga

Consideriamo ora un aspetto che abbiamo tralasciato in tutta quest'ultima parte. Tutte le operation dei web service vengono richiamate dalla classe WSCall, ma non abbiamo ancora spiegato di cosa si tratta, l'abbiamo sempre tralasciata. Cos'è quindi WSCall? Perché non vengono usati direttamente i manager per richiamare le operazioni?

WSCall è una classe che permette di capire in che web service stiamo lavorando, infatti controlla il valore del parametro "wsname" e in base a questo chiama l'operation opportuna di quel determinato web service che è stato così identificato. Tutto ciò è stato necessario poiché le operation dei manager hanno tutte gli stessi nomi, per cui è necessario verificare precedentemente in quale web service si deve operare. Per rendere il codice più ordinato e pulito, questo controllo è stato fatto in una classe separata dalle jsp, altrimenti la lettura e la manutenzione del codice sarebbe risultata molto lunga e difficoltosa.

Infine nella pagina di gestione sono presenti altri bottoni oltre quelli già analizzati: il pulsante per eliminare i file e quello per rinominarli.

Figura 19 Bottoni elimina e rinomina

Cliccando sul pulsante di eliminazione, come per gli altri, si andrà a richiamare la pagina di gestione eseguendo questa parte di codice:

```
if (request.getParameter("delete") != null) {
    File toDelete = new File(request.getParameter("delete"));
    if (toDelete.exists()) {
        if (toDelete.delete()) {
            successo = true;
            messaggioSuccesso = "eliminazione del file \"" +
                    toDelete.getName() + "\" riuscita correttamente!";
        }
        else {
            errore = true;
            messaggioErrore = "eliminazione del file \"" +
                    toDelete.getName() + "\" non riuscita!";
        }
    }
    else {
        errore = true;
        messaggioErrore = "file \"" +
                toDelete.getName() + "\" inesistente!";
    }
}
```

Mentre cliccando sul pulsante di rinomi nazione, verrà effettuata la stessa procedura, eseguendo però la parte di codice sottostante.

```
if (request.getParameter("rename") != null) {
    String newName = request.getParameter("rename");
    if (!newName.equals("RENAME")) {
        File toRename = new File(request.getParameter("oldName"));
        if (toRename.exists()) {
            File parent = toRename.getParentFile();
            File newFile = new File(parent, newName);
            if (toRename.renameTo(newFile)) {
                successo = true;
                messaggioSuccesso = "file \"" + toRename.getName()+
                        "\" rinominato correttamente come \"" +
                        newFile.getName() + "\"!";
            }
            else {
                errore = true;
                messaggioErrore = "impossibile rinominare il file \""+
                        toRename.getName() + "\"!";
            }
        }
        else {
            errore = true;
            messaggioErrore = "file \""+toRename.getName()+"\" inesistente!";
        }
    }
```

Anche questi due pulsanti potrebbero essere realizzati facilmente con due operation dei web service, in quanto rappresentano due funzionalità base e basterebbe riportare il codice appena visto all'interno di due apposite operation nel manager.

4.2 Funzionamento Web Service

Avendo esposto pienamente il funzionamento del client nella parte precedente, soffermiamoci ora sul codice inserito nei web service (vedi appendice punto C).

Notando che ogni web service risiede in un progetto web application completamente indipendente dagli altri, si è pensato di inserire all'interno di ciascuno di essi, assieme al codice del webservcive vero e proprio, un'altra classe chiamata "ManagerServiceProgetti" per progetti, "ManagerServiceGruppi" per gruppi New, "ManagerServiceGruppiTW" per i gruppiTW e così via.

Dentro ad ogni manager ho inserito le varie operation (quelle che abbiamo già citato e intravisto nel sottocapitolo precedente), le quali ci permettono di effettuare tutte le operazioni principali per la modifica del codice del web service stesso. La creazione delle operation è stata resa molto immediata e semplice grazie all'IDE, poiché mette a disposizione un interfaccia per indicare il tipo dei parametri che l'operation deve ricevere in ingresso e il tipo di oggetto che restituisce, creando automaticamente lo scheletro del codice dell'operation e facilitando così il lavoro del programmatore. Inoltre creando una classe java (es. come vedremo ArrayString per una delle operation) e indicandola all'interfaccia, NetBeans automaticamente genera il relativo xml-schema da inserire nel wsdl (vedi sezione 2.2.2).

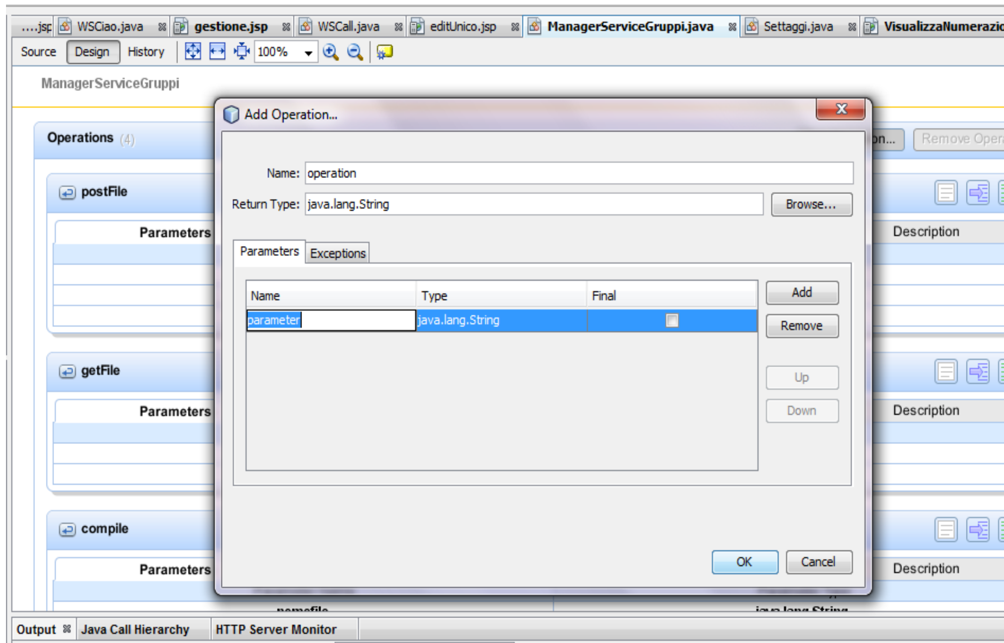

Figura 20 Interfaccia per creare operation

Le operation che vengono utilizzare e che successivamente andremo a descrivere nel particolare sono:

- getDir, la quale permette di visualizzare l'elenco dei file presenti in un determinata directory;

- getFile, che permette di scaricare, e quindi poter visualizzare in seguito, il contenuto di un determinato file;

- postFile, permette di caricare un determinato file, quindi anche di apportare determinate modifiche al codice stesso;

- compile, che permette di compilare i file .java.

Analizziamo ora in ogni sottocapitolo un'operation nel particolare.

4.2.1 Operation: getDir

L'operation getDir , riceve come parametri in ingresso una stringa contenente il percorso della directory e restituisce un oggetto di tipo "ArrayString" che contiene l'elenco di tutti i file presenti all'interno della directory d'interesse.

```
public ArrayString getDir(@WebParam(name = "path") String path)
```

Questa operazione, come abbiamo già detto, restituisce l'elenco di tutti i file contenenti nella directory che le viene passata come input.

ArrayString è la seguente classe creata da me:

```
public class ArrayString {
    public String[] i;
}
```

Tale classe è stata indicata, come già spiegato, a NetBeans durante la creazione (tramite interfaccia) della operation.

```
@WebMethod(operationName = "getDir")
public ArrayString getDir(@WebParam(name = "path") String path) {
File dir = new File(Settaggi.getPATH_BASE()+path);
ArrayList<File> risorse = new ArrayList<File>();
ArrayString lista = new ArrayString();
// aggiunge i file alla lista
if (dir.isDirectory()) {
     int x=0;
    File[] af = dir.listFiles();
    lista.i = new String[af.length];
    for(File r : af){
        if(!r.isDirectory())
        {
            lista.i[x]=r.getName();
        }else{
            lista.i[x]=r.getName()+"/";
        }
        x++;
    }
}
return lista;
}
}
```

Dal codice riportato sopra possiamo vedere meglio il funzionamento di questa operazione, che restituisce l'oggetto ArrayString costruito da me, in una classe che contiene un array di stringhe, questo per facilitare il passaggio di questo oggetto. All'interno del codice dell'operazione vediamo che ho costruito un file "dir" che contiene il contenuto del percorso, successivamente tramite il comando ".listFiles()" ho inserito sotto forma di lista l'elenco prima ottenuto in un array di file. Dopo di che ho costruito un array di strighe della stessa lunghezza dell'array di file e grazie ad un ciclio "for", che permette di scorrere tutto l'elenco, ho potuto controllare se la risorsa fosse o meno una directory, aggiungendo nel caso opportuno la "/" per identificarla e riconoscerla successivamente a livello client.

4.2.2 **Operation: getFile**

L'operation getFile, riceve come parametri in ingresso una stringa contenente il percorso del file d'interesse e restituisce una stringa al cui interno è presente il contenuto del file.

```
public String getFile(@WebParam(name = "path") String path)
```

Come già detto questa operazione legge il contenuto di un file e ne restituisce una stringa con al suo interno il contenuto, prendendo in input il semplice percorso della risorsa interessata, come si può vedere in questo frammento di codice:

```
/**
 * Web service operation
 */
@WebMethod(operationName = "getFile")
public String getFile( @WebParam(name = "path") String path) {
    //TODO write your implementation code here:
    try{
        BufferedReader input = new BufferedReader(new FileReader(Settaggi.getPATH_BASE()+path));
            String line = "";
            String contenuto = "";
            while ((line = input.readLine()) != null)
                contenuto = contenuto.concat(line) + System.getProperty("line.separator");
            return contenuto;

    } catch (Exception ex) {
          return "errore!";
    // TODO handle custom exceptions here
}
}
```

Come possiamo notare dal codice mostrato sopra, è stato necessario costruire un BufferReader che contenesse il file presente nel percorso ricevuto in input, dopo di che è servito un ciclo while che scorresse tutto i buffer e concatenasse il contenuto in una stringa, per poi restituirla in output.

4.2.3 **Operation: postFile**

L'operation getFile , riceve come parametri in ingresso una stringa contenente il percorso del file da modificare e una stringa in cui è presente il nuovo contenuto del file, infine restituisce una stringa che indica se l'operazione è avvenuta con successo o meno.

```
public String postFile(@WebParam(name = "file") String file,
                       @WebParam(name = "path") String path) {
```

Questa operazione permette la modifica del contenuto di un determinato file; infatti ricevendo in input il nuovo testo ed il percorso del file da modificare, sostituisce il contenuto di esso con quello nuovo.

```
@WebMethod(operationName = "postFile")
public String postFile(@WebParam(name = "file") String file, @WebParam(name = "path") String path) {
    try{
        BufferedWriter output = new BufferedWriter(new FileWriter(Settaggi.getPATH_BASE()+path));
        output.write(file);
        output.close();
        return "ok";
    } catch (Exception ex) {
        return null;
    }
}
```

Come si può vedere dal codice riportato sopra, in questo caso è stato necessario creare un BufferWriter che permette la scrittura, a differenza del BufferReader usato prima per "getDir", e con un semplice comando ".write()", che prende in input il nuovo contenuto da inserire, la modifica risulta molto semplice ed immediata.

Infine questa operation restituisce una semplice stringa "ok" per enfatizzare il fatto che il caricamento del nuovo contenuto è andato a buon fine, in modo che sia più facile capirlo a lato client.

4.2.4 **Operation: compile**

Infine analizziamo l'ultima operation che è quella che permette la compilazione di un file ".java", cioè l'operazione "compile", che riceve come parametri in ingresso una stringa che contiene il nome del file da compilare e un'altra stringa che indica il percorso in cui trovare il file interessato.

```
public String compile(@WebParam(name = "nomefile") String nomefile,
                      @WebParam(name = "path") String path) {
```

Osserviamo e analizziamo ora il codice di questa operation:

```
@WebMethod(operationName = "compile")
public String compile(@WebParam(name = "nomefile") String nomefile, @WebParam(name = "path") String path) {
    File percorso = new File(Settaggi.getPATH_BASE()+"/WEB-INF/lib/");
            String[] list;
            String librerie = new String();
            list = percorso.list();
            for(int i = 0; i < list.length; i++){
                librerie = librerie + Settaggi.getSEPARATORE() + Settaggi.getPATH_BASE() + "/WEB-INF/lib/" + list[i];
             }
     String[] optionsAndSources = {"-classpath",Settaggi.getPATH_BASE()+"/WEB-INF/classes/"+
                                Settaggi.getSEPARATORE()+Settaggi.getPATH_BASE()+"servlet-api.jar"+librerie,
                                Settaggi.getPATH_BASE()+path+"/"+nomefile};
     CharArrayWriter out = new CharArrayWriter();
     int status =  Main.compile( optionsAndSources,new PrintWriter(out));
     if(status==0)
     {
         return "OK;"+out.toString();
     }
     else
      {
         return "ERRORE;"+out.toString();
      }
}
```

A differenza delle altre operazioni si può notare che quest'ultima riceve in input anche il nome del file, oltre che al percorso del file da compilare, in modo da rendere più semplici alcune operazioni al suo interno.

Inizialmente ho creato un file contenente le librerie e tramite al comando ".list()" ho inserito l'elenco di queste ultime in un array di stringe;il passaggio successivo consiste nel riempire una stringa, grazie ad un ciclio for, concatenando al suo interno tutti i percorsi di ogni libreria, ottenuti precedentemente.

Dopo di che all'interno di un array di stringhe "optionsAndSource" ho inserito tutti gli elementi utili al comando di compilazione, cioè:

- “-classpath”;

- il percorso che porta alle classi del web service;

- il percorso per la libreria "servlet-api.jar";

- la stringa “librerie” che ho riempito in precedenza con tutti i percorsi delle singole librerie;

- il percorso assoluto del file da compilare, utilizzando il nome che è stato passato come parametro all’opration.

Infine ho chiamato il metodo “.compile()”, passandogli l’array di stringhe appena citato e inserendo un PrintWriter di un CharArrayWriter, quest’ultimo è necessario per la stampa dell’errore di compilazione. Infatti sfruttando il fatto che questo metodo restituisce un intero che vale 0 se la compilazione è corretta o 1 se ci sono stati degli errori, è stato possibile restituire due stringhe diverse per ognuno dei due casi, contenente una prima parte che indica il successo o meno della compilazione e una seconda parte che riporta tutto l’errore.

4.3 Strumenti di sviluppo e deployment

Per lo sviluppo ed il debug del codice è stato scelto come IDE quello direttamente consigliato da Sun, ossia NetBeans IDE [NBIDE11]. È stata installata la versione 7.2 per lo sviluppo di codice del progetto, perché al suo interno implementa il web server Tomcat 7.0.22[ASF08A], ovvero la penultima versione disponibile, la quale si avvia automaticamente ogni qual volta venga mandata in esecuzione la *web application* contenente le servlet. Oltre a tutte le funzionalità relative alle ultime librerie Java e al parsing XML, implementa anche la libreria specifica per il *building* dei progetti per la piattaforma Java Web Start e le classi per rendere

possibile la scrittura protetta su disco locale e la stampa di documenti. Compilando il codice creato con gli strumenti messi a disposizione, l'IDE crea automaticamente il file e le cartelle necessarie, esclusi di file jar delle librerie (che ovviamente dovranno essere copiati manualmente), nella cartella *dist* all'interno del progetto. Il materiale dentro questa cartella è pronto per l'esecuzione e per l'installazione su server web.

Inoltre durante il tirocinio da me svolto con il relatore della tesi abbiamo preso in considerazione che i web-service erano su Tomcat6 mentre il sito era su Tomcat7, per cui abbiamo deciso di mettere tutto su Tomcat7 per avere il tutto più ordinato ed efficiente. Per cui abbiamo spostato i web-service su Tomcat7 disattivando infine Tomcat6.

Inoltre è stato effettuato il deployment sul server del corso di laurea twisrv.csr.unibo.it. Tutto questo per dire che usando direttamente Tomcat su NetBeans, sarebbe stato molto più facile trasportare le modifiche fatte localmente direttamente sul sito, poiché non sarebbe stato necessario apportare alcuna modifica, differentemente sarebbe avvenuto se avessimo usato GlassFish su NetBeans.

Capitolo 5 Conclusioni

In questa tesi è stato sviluppato un modulo di gestione che ha reso possibile gestire i web service durante il loro utilizzo. In particolare, oltre a miglioramenti della struttura del sito e alla realizzazione della nuova funzionalità di correzione degli esami, sono state implementati dei manager all'interno dei web service del sito di Tecnologie Web, i quali contengono tutte le operation necessarie per interagire con i web service stessi da remoto.

Lo svolgimento del lavoro svolto è stato molto interessante e formativo, in quanto mi ha dato la possibilità di interagire con nuovi strumenti e sviluppare funzionalità molto utili per il mondo web attuale. Durante il percorso lavorativo sono emerse alcune difficoltà di utilizzo riguardanti NetBeans, l'ambiente di sviluppo (IDE) scelto. In particolare NetBeans non consente di utilizzare uno stesso nome per i manager (tutti identici) da includere nei vari servizi web del sito, che sarebbero comunque distinti dal fatto di essere deployati a indirizzi fisici differenti. Infatti, quando si tenta di invocare un'operation contenuta in uno di essi, l'IDE non è in grado di distinguere a quale si faccia riferimento. Per questo motivo i vari manager sono stati denominati in modo diverso in base al web service in cui sono inseriti. Nonostante questa problematica, NetBeans si è rilevato molto utile per creare le varie operation all'interno del manager, grazie all'interfaccia che mette a disposizione per crearle, per specificare i parametri e i valori di ritorno e generare automaticamente l'xml-schema a partire dalla loro classe java.

In futuro, ulteriori sviluppi del "modulo" di gestione potrebbero consistere nel rendere i deployment del modulo, sia per lato client che per lato server, totalmente indipendente dal sito a cui si applica, creando un meccanismo standard per interagire con i web service durante il loro utilizzo.

Un ulteriore sviluppo futuro, potrebbe essere incentrato sulla sicurezza di queste interazioni con i web service, in quanto trasportare il codice dei web service stessi nella rete può risultare abbastanza rischioso. A questo scopo possono essere utilizzati protocolli a diversi livelli (per esempio https o ws-security) per questo scambio di informazioni. Inoltre sarebbe opportuno installare un meccanismo di login a livello dei web service facendo in modo da ottenere, nell'insieme, una realizzazione che permetta l'assoluta sicurezza per le informazioni.

Bibliografia

[APBWS07] Antonio Pintus, Building Web Services with Java , 2007

[AR06] Alfredo La Rotonda,“Le applicazioni web e Java”, http://www2.mokabyte.it/cms/article.run?articleId=J74-ZHK-E4Y-62F_7f000001_30520983_51b9ed95, MokaByte 111 - Ottobre 2006

[AMT01] Wikipedia the free encyclopedia, “Architettura Multitier”, http://it.wikipedia.org/wiki/Architettura_Multitier

[ASF08A] Apache Software Foundation, “Apache Tomcat”, http://tomcat.apache.org, 2008

[BIS06] Rahul Biswas, “Document Handling Using JAX-WS Dispatch and Provider APIs and Using Multiple Databases in a Java Persistence Application”, http://java.sun.com/mailers/techtips/enterprise/2006/TechTips_July06.html, 2006

[CAR-JWS] Bryan Carpenter, “Introduction to JAX-WS” (JWS), http://www.omii.ac.uk/wiki/JaxWsTutorial, 2009

[CER06] Dr.ssa Katuscia Cerbioni, “Introduzione all’ XML”, http://www.di.unipi.it/~cerbioni/files/Intro_XML.pdf, 2006

[CEW02A] E. Cerami. "Web Services Essentials". O'Reilly. 2002.

[CLA03A] M. Clark. M. Waterhouse, P. Fletcher., "Web Services Business Strategies and Architectures'", 2003

[DDDTWSTI03], M. Deitel. P. J. Deitel, B. DuWaldt, L. K. Trees. "Web Services A Technical Introduction.", Deitel Developer Series, 2003

[DEI03A] H. M. Deitel, P. J. Deitel, B. DuWaldt, L. K. Trees., Web Services A Technical Introduction. Deitel Developer Series, 2003

[EMO06] Robert Eckstein, Rajiv Mordani, "Introducing JAX-WS 2.0 With the Java SE 6 Platform, Part 1", http://java.sun.com/developer/technicalArticles/J2SE/jax_ws_2/, 2006

[FUS08] FUSE Services Framework, "Working with Dispatch Objects", http://fusesource.com/docs/framework/2.1/jaxws/JAXWSDispatchWorking.html, 2008

[GSD03A] S. Graham, S. Simeonov, T. Boubez, G. Daniels, D. Davis, Y. Nakamura, R. N'eyama., "Building Web Services with Java: Making Sense of XML, SOAP, WSDL and UDDL", Paperback, 2003.

[HTTP] "HyperText Transfer Protocol" http://www.w3.org/Protocols/, 2009

[IBM-CWS] IBM, "Modello di programmazione del client JAX-WS" (CWS), http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/cwbs_jaxwsclients.html, 2009

[JAXB07] Java Architetture for XML Binding, http://java.sun.com/webservices/jaxb/index.jsp , 2007

[JCP-WS] Java Community Process, "JSP 224: Java API for XML-Based Web Services (JAX-WS)2.0", http://www.jcp.org/en/jsr/detail?id=224, 2009

[JTRPC07] Java Technology, www.jtechnology.it/jax-rpc/ , 2007

[KAO01] James Kao, "Overview of WSDL", http://developers.sun.com/appserver/reference/techart/overview_wsdl.html, 2001

[NBIDE11] NetBeans IDE http://www.netbeans.org, 2011

[OME03] Ed Ort, Bhakti Mehta, "Java Architecture for XML Binding", http://java.sun.com/developer/technicalArticles/WebServices/jaxb/, 2003

[RFGP04] V. Roberto, M. Frailis, A. Gugliotta, P. Omero, "Introduzione alle tecnologie web", The McGraw-Hill Companies, Milano Novembre 2004

[SBR09] Francesco Sbrighi, "Breve approfondimento sulla libreria JAXB", "Relazione di Tirocinio Curriculare obbligatorio", 2009, pp. 10-18

[SMA06] John Ferguson Smart, "Web Services Made Easy with JAX-WS 2.0" http://today.java.net/pub/a/today/2006/06/13/web-services-with-jax-ws-2.0.html, 2006

[SM11A] Sun Microsystems, "Java 2 Platform Enterprise Edition", http://java.sun.com/j2ee/ 2011

[SOAP03] W3C, "SOAP 1.2 Part 1: Messaging Framework", http://www.w3.org/TR/SOAP/, 2003

[STH09] Dan Sedov, Nikhil Thaker, per IBM, “JAX-WS client APIs in the Web Services Feature Pack for WebSphere Application Server V6.1, Part 1: Creating a Dispatch client”,
http://www.ibm.com/developerworks/websphere/library/techarticles/0707_thaker/0707_thaker.html, 2009

[SUN09a] Sun Microsystems, “Sviluppo di un’ implementazione dell’ endpoint di servizio per le applicazioni JAX-WS con annotazioni”,
http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/twbs_devjaxwsendpt.html, 2009

[TDGV] Tesi di Gessica Valbonesi, “Gestione di siti web basati su tecnologia java tramite wiki”, Anno Accademico 2010/2011

[TRI03] Ravi Trivedi, “Web Services Tutorial: Understanding XML and XML Schema, Part 1”,
http://www.developer.com/services/article.php/2195981/Web-Services-Tutorial-Understanding-XML-and-XML-SchemamdashPart-1.htm, 2003

[TUR09a] Giulio Turetta, “Guida Web Service”
http://xml.html.it/guide/leggi/100/guida-web-service/, 2009

[UDDI] “Uddi xml org”,
http://uddi.xml.org/wiki, 2009

[W3C04A], W3C Recommendation. "Extensible Markup Language (XML) 1.0 (Third Edition)", http://www.w3.org/TR/REC-xml/ . February 2004.

[W3CXS01] W3C. XML Schema.W3C Recommendation 02.05.2001, http://www.w3.org/TR/xmlschema-0/, 2007

[W3S-WWS] W3Schools.com, "Why Web Services?" (WWS),
http://www.w3schools.com/webservices/ws_why.asp

[WIK09a] Wikipedia the free encyclopedia, "Web Service"
http://en.wikipedia.org/wiki/Web_service, 2009

[WIK09b] "HyperText Transfer Protocol"
http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol, 2009

[WSDL03] W3C, "Web Services Description Language (WSDL) 1.1",
http://www.w3.org/TR/wsdl, 2003

Appendici

A. Gestione.jsp

- Frammento di codice che controlla i paramentri passati e setta le variabili, nel quale sono stati inseriti vari if che permettono di collocarsi nelle giuste risorse:

```
// prova ad eseguire una multipart request
try {
    multi = new MultipartRequest(request, ".");
} catch (IOException e) {
  // l'eccezione si verifica la prima volta che si accede alla pagina
}
// controlla se &egrave; stato impostato il corso
if (multi != null) {
    if (multi.getParameter("corso") != null) {
        corso = (String) multi.getParameter("corso");
    }
}
// controlla se &egrave; stato impostato il corso nella URL
if (request.getParameter("corso") != null) {
    corso = (String) request.getParameter("corso");
}
// controlla se &egrave; stato impostato la risorsa
if (multi != null) {
    if (multi.getParameter("ris") != null) {
        ris = (String) multi.getParameter("ris");
    }
}
// controlla se &egrave; stato impostato la risorsa
if (request.getParameter("ris") != null) {
    ris = (String) request.getParameter("ris");
}
if (request.getParameter("wsname") != null) {
    wsname = (String) request.getParameter("wsname");
}
if (!wsname.equals("no")){
    nomeRis = "WebService";
    destPath = "/WEB-INF/classes";

if (corso.equals("classi")) {
        nomeCorso = "Classi";
        destPath = "/WEB-INF/classes";
     //aggiungere in base quale ws o se wsname=no il pezzo prima

    }
 if (corso.equals("librerie")) {
        nomeCorso = "Librerie";
        destPath = "/WEB-INF/lib";
    }
```

```
 if (corso.equals("configurazione")) {
        nomeCorso = "Configurazione";
        destPath = "/WEB-INF";
    }
 if (corso.equals("directory")) {
        nomeCorso = "Directory Radice";
        destPath="build/web" ;
    }
 if (corso.equals("other") && multi != null) {
        nomeCorso = "Varie";
        destPath = multi.getParameter("otherPath");
     }
  if (corso.equals("other") && request.getParameter("otherPath") != null) {
        nomeCorso = "Varie";
        destPath = request.getParameter("otherPath");
    }
  }else{
if (ris.equals("r1")) {
    nomeRis = "Lucidi";
    if (corso.equals("tw")) {
    nomeCorso = "Tecnologie Web";
    destPath =  Settaggi.getPATH_BASE()+"lucidi/materialeDidattico/tw";
    destURL = "../lucidi/materialeDidattico/tw/";
   }
   if (corso.equals("twa")) {
    nomeCorso = "Tecnologie Web Avanzate";
    destPath = Settaggi.getPATH_BASE() + "lucidi/materialeDidattico/twa";
    destURL = "../lucidi/materialeDidattico/twa/";
   }
   if (corso.equals("shared")) {
    nomeCorso = "Risorse Condivise";
    destPath = Settaggi.getPATH_BASE() + "lucidi/materialeDidattico/shared";
    destURL = "../lucidi/materialeDidattico/shared/";
   }
}
if (ris.equals("compiti")) {
    nomeRis = "Compiti";
   destPath=Settaggi.getPATH_BASE()+"/progettiTW/"+dataApp+ "/unzip";
    destURL = "../progettiTW/" + dataApp + "/unzip";
}
if (ris.equals("classlib")) {
    nomeRis = "Classi";
    destPath = Settaggi.getPATH_BASE() + "WEB-INF/classes";
    destURL = "../WEB-INF/classes/";
    if (corso.equals("classi")) {
        nomeCorso = "Classi";
        destPath = Settaggi.getPATH_BASE() + "WEB-INF/classes";
        destURL = "../WEB-INF/classes/";
    }
    if (corso.equals("librerie")) {
        nomeCorso = "Librerie";
        destPath = Settaggi.getPATH_BASE() + "WEB-INF/lib";
        destURL = "../WEB-INF/lib/";
    }
```

```
    if (corso.equals("configurazione")) {
        nomeCorso = "Configurazione";
        destPath = Settaggi.getPATH_BASE() + "WEB-INF";
        destURL = "../WEB-INF/";
    }
    if (corso.equals("directory")) {
        nomeCorso = "Directory Radice";
        destPath = Settaggi.getPATH_BASE();
        destURL = Settaggi.getPATH_BASE();
    }
}
if (corso.equals("other") && multi != null) {
        nomeCorso = "Varie";
        destPath = multi.getParameter("otherPath");
         destURL = destPath.replaceFirst(Settaggi.getPATH_BASE(),
         "../") + File.separator;
    }
   if (corso.equals("other") && request.getParameter("otherPath")!=
   null)  {
        nomeCorso = "Varie";
        destPath = request.getParameter("otherPath");
         destURL = destPath.replaceFirst(Settaggi.getPATH_BASE(),
         "../") + File.separator;
    }
   }
```

- Frammento che permette di visualizzare le risorse, in cui è stato inserito un controllo per verificare se si tratta di un web service, inserendo in questo modo la chiamata all'operation:

```
// scorre tutti i file presenti nella directory di destinazione
    ArrayList<File> risorse = new ArrayList<File>();
    List<String> listadir = new ArrayList<String>();
    ArrayString lista = new ArrayString();
    if (!wsname.equals("no")){
        lista = WSCall.getDir(wsname,destPath);
       try{
        listadir = lista.getI();
        listadir.toArray();
       } catch (Exception ex) {
       }
    }else{
      File dir = new File(destPath);
      // aggiunge i file alla lista
      if (dir.isDirectory()) {
        for (String r : dir.list()) {
            risorse.add(new File(dir, r));
        }
      }
    }
    // converte l'ArrayList in un array semplice
    File[] risorseArray = new File[risorse.size()];
    risorse.toArray(risorseArray);
    // ordina l'array di file con una funzione personalizzata
    Arrays.sort(risorseArray, new Comparator() {
        public int compare(Object o1, Object o2) {
            String name1 = ((File) o1).getName();
            String name2 = ((File) o2).getName();
            return name1.compareToIgnoreCase(name2);
        }
    });
```

- Frammento di codice inserito che permette di compilare i file e mostrare l'errore:

```
if (request.getParameter("compila") != null) {
   File tocompile = new File(request.getParameter("compila")); %>
<pre class="comp">
  <% try{
      if (!wsname.equals("no")){
        String risultato="";
        String nomeFile = tocompile.getName();
        risultato = WSCall.compile(wsname,nomeFile,destPath);
        if(risultato.startsWith("OK")){
           successo = true;
            messaggioSuccesso ="File\""+nomeFile+"\"compilato
            correttamente";
            out.println(risultato.substring(risultato.indexOf(';')+1));
              }
        if(risultato.startsWith("ERRORE")){
              errore = true;
             messaggioErrore = "File \"" + tocompile.getName() + "\" non
            Compilato!!" ;
            out.println(risultato.substring(risultato.indexOf(';')+1));
         }
      }else{
           File path = new File(Settaggi.getPATH_BASE()+"WEB-INF/lib/");
           String[] list;
           String librerie = new String();
           list = path.list();
           for(int i = 0; i < list.length; i++){
                   librerie = librerie + Settaggi.getSEPARATORE() +
                   Settaggi.getPATH_BASE() + "WEB-INF/lib/" + list[i];
               }
             String[] optionsAndSources =
            {"-classpath",Settaggi.getPATH_BASE()+
            "WEB-INF/classes/"+ Settaggi.getSEPARATORE()+
            Settaggi.getPATH_BASE()+ "servlet-api.jar"+ librerie,
            request.getParameter("compila")};
            int status =  Main.compile( optionsAndSources, new
            PrintWriter(out));
        if(status==0) {
            successo = true;
            messaggioSuccesso = "File\"" + tocompile.getName() + "\"
            compilato correttamente";
        }else{
             errore = true;
             messaggioErrore = "File \"" + tocompile.getName() + "\" non
            Compilato!!  " + "status: " + status;
         }
       }
      } catch (Exception ex) {
      }
        %>
  </pre>
        <%}
            if (errore) {%>
      <p class="error"><strong>Errore:</strong><%=messaggioErrore%> </p>
            <% }
                if (successo) {%>
<p class="success"><strong>Successo:</strong><%=messaggioSuccesso%></p>
               <% }%>
```

B. EditUnico.jsp

- Frammento di codice per salvare il file, in cui è stato inserito un controllo per i web service:

```
if (salva) {
  String sorgente = request.getParameter("contenuto");
  percorsoPagina = request.getParameter("percorsoPagina");

  File destinazione = new File(percorsoPagina);
boolean backup = request.getParameter("backup").equals("si")? true:false;
  visualizza = true;
  if (!wsname.equals("no")){
            String nomefile="";
            String carica =
WSCall.postFile(wsname,sorgente,percorsoPagina);
      if(carica.equals("ok")){
             successoSalva = true;
      messaggioSuccessoSalva = "pagina webservice \"" +
      destinazione.getName() + "\" modificata correttamente!";
      }else{
         erroreSalva = true;
      messaggioErroreSalva = "pagina \"" + destinazione.getName()
      + "\" NON modificata correttamente!";
       }
  }else{
     if (destinazione.exists()) {
     try {
      BufferedWriter output = new BufferedWriter(new
      FileWriter(destinazione));
         try {
           output.write(sorgente);
           successoSalva = true;
             messaggioSuccessoSalva = "pagina \"" +
             destinazione.getName() + "\" modificata correttamente!";
         }
         catch (IOException e) {
             erroreSalva = true;
             messaggioErroreSalva = e.getMessage();
          }
          finally {
             output.close();
          }
       }
       catch (IOException e) {
           erroreSalva = true;
          messaggioErroreSalva = e.getMessage() + "<br />Nota:
      probabilmente è necessario impostare i permessi di scrittura per
      quel file con <code>chmod 775 " + destinazione.getAbsolutePath()
      + "</code>.";
       }
      }
      else {
            erroreSalva = true;
             messaggioErroreSalva = "la pagina \"" +
             destinazione.getName() + "\" è inesistente!";
        }
               }
    }
```

- Frammento di codice per visualizzare una file, in cui è stato inserito un controllo per i web service:

```
if (visualizza) {
    percorsoPagina = request.getParameter("percorsoPagina");
    int questionMark = percorsoPagina.indexOf("?");
    if (questionMark != -1) {
        percorsoPagina = percorsoPagina.substring(0, questionMark);
    }
    if (request.getParameter("js") != null) {
        File temp = new File(percorsoPagina);
         File parent =
         (newFile(temp.getAbsolutePath())).getParentFile();
        String realPath = Settaggi.getPATH_BASE() + parent.getName()
        + File.separator + temp.getName();
        percorsoPagina = realPath;
    }
     if (!wsname.equals("no")){
        String nomefile="";
        contenuto = WSCall.getFile(wsname, percorsoPagina);
        successoVisualizza = true;
         messaggioSuccessoVisualizza = "pagina webservice caricata
         correttamente!";
    }else{
    File sorgente = new File(percorsoPagina);
    File paginaModifica = new File(request.getRequestURI());
    if (sorgente.exists()) {
        if (sorgente.getName().equals(paginaModifica.getName())) {
            erroreVisualizza = true;
                messaggioErroreVisualizza = "non è possibile
                modificare questa stessa pagina!";
        }
        else {
          BufferedReader input =
          new BufferedReader(newFileReader(sorgente));
            String line = "";
            contenuto = "";
            try {
                while ((line = input.readLine()) != null)
                contenuto = contenuto.concat(line) +
                System.getProperty("line.separator");
                 successoVisualizza = true;
                messaggioSuccessoVisualizza = "pagina \"" +
                sorgente.getName() + "\" caricata correttamente!";

             }
            catch (IOException e) {
                erroreVisualizza = true;
                messaggioErroreVisualizza = e.getMessage();
            }
            finally {
                input.close();
            }
        }
    }
    else {
        erroreVisualizza = true;
         messaggioErroreVisualizza = "la pagina \"" +
         sorgente.getName() + "\" è inesistente!";
    }
}
}
```

C. Manager.java

Codice dei manager in cui sono presenti tutte le operation utilizzati per la gestione dei web service.

```
package manager;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import com.sun.tools.javac.Main;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.*;
import java.util.*;
import settaggi.*;

@WebService(serviceName = "ManagerServiceGruppi")
public class ManagerServiceGruppi {

@WebMethod(operationName = "postFile")
public String postFile(@WebParam(name = "file") String file,
@WebParam(name = "path") String path) {
     try{
       BufferedWriter output = new
       BufferedWriter(newFileWriter(Settaggi.getPATH_BASE()+path));
        output.write(file);
        output.close();
        return "ok";
     } catch (Exception ex) {
             return null;
        }
    }
```

```
@WebMethod(operation
Name = "getFile")
    public String getFile(@WebParam(name = "path") String path) {
        try{
             BufferedReader input = new BufferedReader(new
             FileReader(Settaggi.getPATH_BASE()+path));
                String line = "";
                String contenuto = "";
                       while ((line = input.readLine()) != null)
             contenuto = contenuto.concat(line) +
             System.getProperty("line.separator");
                return contenuto;
        } catch (Exception ex) {
             return "errore!";
        }
    }
```

```
@WebMethod(operation
Name = "compile")
    public String compile(@WebParam(name = "nomefile") String nomefile,
         @WebParam(name = "path") String path) {
File percorso = new File(Settaggi.getPATH_BASE()+"/WEB-INF/lib/");
                String[] list;
                String librerie = new String();
                list = percorso.list();
                for(int i = 0; i < list.length; i++){
                    librerie = librerie + Settaggi.getSEPARATORE() +
                    Settaggi.getPATH_BASE() + "/WEB-INF/lib/" + list[i];
                 }
             String[] optionsAndSources = {
             "-classpath",Settaggi.getPATH_BASE()+
             "/WEB-INF/classes/"+ Settaggi.getSEPARATORE()+
             Settaggi.getPATH_BASE()+
             "servlet-api.jar"+ librerie,
             Settaggi.getPATH_BASE()+path+"/"+nomefile};
         CharArrayWriter out = new CharArrayWriter();
int status =  Main.compile( optionsAndSources,new PrintWriter(out));
         if(status==0)
         {
             return "OK;"+out.toString();
         }
         else
          {
             return "ERRORE;"+out.toString();
          }
    }
```

```
@WebMethod(operation
Name = "getDir")
    public ArrayString getDir(@WebParam(name = "path") String path) {
        File dir = new File(Settaggi.getPATH_BASE()+path);
    ArrayList<File> risorse = new ArrayList<File>();
    ArrayString lista = new ArrayString();
    if (dir.isDirectory()) {
         int x=0;
        File[] af = dir.listFiles();
        lista.i = new String[af.length];
        for(File r : af){
            if(!r.isDirectory())
            {
                lista.i[x]=r.getName();
            }else{
                lista.i[x]=r.getName()+"/";
            }
            x++;
        }
    }
        return lista;
    }
}
```