



Università degli Studi “Roma Tre”

Sezione di Informatica ed automazione

Dipartimento di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea

Sviluppo di un framework per la sperimentazione di Architetture Software

Relatore

Prof. L. Cabibbo

Università Roma Tre

Candidato

Francesco Paris

matricola 453908

Anno Accademico 2013 – 2014

*Ai miei genitori, Giuseppe e Monica,
per il costante e prezioso
supporto in questi anni di studio*

INDICE

INTRODUZIONE	pag. 8
CAPITOLO 1: MAVEN	
1.1 Premessa	pag. 12
1.2 Cos'è Maven	pag. 12
1.3 Breve storia di Maven	pag. 13
1.4 Obiettivi primari di Maven	pag. 14
1.5 Vantaggi derivati dall'utilizzo di Maven in un progetto	pag. 15
1.6 I principi base di Maven	pag. 16
1.7 Concetti Maven	pag. 18
1.7.1 File di POM	pag. 18
1.7.2 Plugin e Goal	pag. 20
1.7.3 L'archetipo	pag. 21
1.7.4 Ereditarietà	pag. 22
1.8 Maven ed Eclipse	pag. 23

CAPITOLO 2 : GESTIONE DI GRUPPI DI MACCHINE VIRTUALI

2.1 Premessa	pag. 24
2.2 Cos'è Vagrant	pag. 25
2.3 Perché Vagrant	pag. 26
2.4 Requisiti – Hypervisor	pag. 27
2.5 Concetti base	pag. 27
2.5.1 Vagrant Box	pag. 27
2.5.2 Vagrantfile	pag. 28
2.5.3 Cartelle sincronizzate	pag. 29
2.6 Provisioning	pag. 30
2.6.1 Chef	pag. 31
2.6.2 Puppet	pag. 33
2.7 Ciclo di sviluppo	pag. 37

CAPITOLO 3: ANALISI ARCHIETTURALE SPRING FRAMEWORK

3.1	Premessa	pag. 39
3.2	Cos'è Spring	pag. 39
3.2.1	Core Spring Container	pag. 40
3.2.2	AOP	pag. 41
3.2.3	Data Access & Integration	pag. 41
3.2.4	Web and Remoting	pag. 42
3.2.5	Testing	pag. 43
3.3	Iniezione delle dipendenze	pag. 45
3.4	Programmazione Orientata agli Aspetti	pag. 48
3.5	Semplificare l'integrazione con JEE	pag. 50
3.6	Alternative a Spring	pag. 54
3.7	JMS	pag. 54
3.7.1	La relazione tra JMS e Spring Integration	pag. 54
3.8	Spring e Web Service	pag. 57
3.9	Spring Cloud	pag. 58
3.10	Spring Vs JEE	pag. 59
3.10.1	Comparazione	pag. 61
3.10.2	Cosa usare?	pag. 63

CAPITOLO 4: GESTIONE DEI CAMBIAMENTI, IL CONTROLLO DI VERSIONE

4.1	Problematica	pag. 64
4.2	Gestione dei cambiamenti	pag. 65
4.3	Importanza della gestione dei cambiamenti	pag. 66
4.4	Gestione delle configurazioni software	pag. 66
4.5	Il processo di gestione delle configurazioni	pag. 67
4.6	Controllo delle versioni	pag. 67
4.7	GitHub	pag. 69
4.8	Cooperative Learning	pag. 70

CAPITOLO 5: LO STUDIO DI CASO

5.1	Premessa	pag. 72
5.2	Applicazione di Car-Sharing	pag. 72
5.2.1	Generalità dell'applicazione	pag. 72
5.2.2	Lo Scenario	pag. 73
5.2.3	Requisiti funzionali	pag. 75
5.2.4	Gestione dell'accesso ai client	pag. 78
5.2.5	Vista Funzionale	pag. 79
5.2.6	Requisiti di qualità	pag. 83

CAPITOLO 6: ARCHITETTURA ESAGONALE

6.1	Premessa	pag. 84
6.2	Evoluzione Architettura Enterprise	pag. 84
6.3	Conseguenze evolutive	pag. 87
6.4	L'architettura esagonale	pag. 88
6.5	Come funziona	pag. 94
6.6	Tecniche di implementazione comuni	pag. 94

CAPITOLO 7: PERSISTENZA

7.1	Problematica	pag. 95
7.2	Il modello	pag. 95
7.3	Strutturazione dei Data Access Object	pag. 98
7.4	Hibernate	pag. 98

CAPITOLO 8: LOGICA DI BUSINESS

8.1	Realizzazione della Logica di Business	pag. 103
-----	--	----------

CAPITOLO 9: INTERFACCIA WEB

9.1 Il pattern MVC	pag.105
9.2 MVC in Spring	pag.106
9.2.1 Gestione di una richiesta in Spring MVC	pag.107
9.2.2 Settare Spring MVC	pag.109
9.3 Controller	pag.110
9.4 Applicazione Web	pag.114
9.5 Deployment	pag.122

CAPITOLO 10: REMOTE METHOD INVOCATION – RMI

10.1 Premessa	pag.123
10.2 JAVA RMI	pag.123
10.3 RMI e Spring	pag.126
10.4 Progetto Rmi	pag.128

CAPITOLO 11: MESSAGING

11.1 Premessa	pag.133
11.2 Messaging	pag.133
11.3 Spring JMS	pag.134
11.4 Progetto Jms	pag.135

CAPITOLO 12: ENTERPRISE JAVA BEAN

12.1 Premessa	pag.136
12.2 Enterprise Bean	pag.136
12.3 Benefici nell'uso degli Enterprise Bean	pag.137
12.4 Quando usare gli Enterprise Beans	pag.138
12.5 Progetto EJB	pag.138
12.5.1 Prima versione (uso di Stateful Session Bean)	pag.140
12.5.2 Seconda versione (senza uso di Stateful Session Bean)	pag.143
12.6 Integrazione Spring MVC con EJB	pag.143

CAPITOLO 13: WEB-SERVICE

13.1 Premessa	pag.146
13.2 Web Service	pag.146
13.3 Web Service Soap	pag.147
13.4 Web Service Rest	pag.149

CAPITOLO 14: CONCLUSIONI E SVILUPPI FUTURI

pag.153

BIBLIOGRAFIA

pag.157

INTRODUZIONE

Il mercato attuale richiede lo sviluppo di applicazioni che siano distribuite, transazionali, portabili, sicure, veloci e affidabili. Tali sistemi devono essere sviluppati nel minor tempo possibile per permettere ad una azienda di restare competitiva in un mercato costantemente soggetto a variazioni [OJEE]. La realizzazione di tali applicazioni rappresenta una sfida formidabile per i team di sviluppo che si devono districare in una ragnatela di tecnologie costantemente variabili e con peculiarità diverse. Per rispondere alle esigenze del cliente, sia a livello di funzionalità sia di qualità, uno sviluppatore deve possedere un forte background metodologico e tecnologico. A valle di tali riflessioni si denota immediatamente l'importanza di un corso di Architetture Software erogato in ambito universitario. Tale insegnamento ha lo scopo di presentare ad un discente aspetti sia metodologici sia tecnologici relativi alle architetture software, che hanno un ruolo fondamentale nel raggiungimento degli obiettivi di qualità dei sistemi software. Tali conoscenze, per uno studente, rappresentano un bagaglio culturale immediatamente spendibile nel mondo lavorativo. Nel corso sono affrontate diverse tecnologie. Alcune sono state ampiamente usate nello sviluppo del software, altre rappresentano le tendenze attuali e le possibili evoluzioni future. Tuttavia per avere una vera e propria conoscenza è necessario che lo studente sperimenti con mano tali tecnologie. Una sperimentazione efficace, che prenda in considerazione e mostri il loro reale uso, può essere difficoltosa.

L'obiettivo di questa tesi è definire un'infrastruttura di supporto alla sperimentazione dei concetti affrontati in un corso di Architetture Software. Lo scopo del lavoro di tesi è pertanto quello di proporre delle idee e degli strumenti, per permettere allo studente di utilizzare tecnologie ormai consolidate nello sviluppo del software ma anche di entrare in serio contatto con tecnologie all'avanguardia. Per ottemperare a tale obiettivo, una prima fase del lavoro di

tesi è stata impiegata per rintracciare una serie di strumenti che ben si prestassero ad essere introdotti in un corso come Architetture Software per sperimentare quanto appreso durante le lezioni. Gli strumenti identificati hanno la caratteristica di essere semplici da usare e con una curva di apprendimento bassa in modo da non costituire un aggravio per lo studente. Successivamente l'attenzione è stata rivolta ad individuare una applicazione d'esempio che ben si prestasse ad essere impiegata per una realizzazione che sfruttasse gli strumenti identificati, le tecnologie e le metodologie affrontate durante il corso di Architetture Software. Tale applicazione è stata quindi implementata in una pluralità di modi pensando ad una possibile erogazione come homework da destinare agli studenti del corso. Per quanto concerne gli strumenti, è stato proposto l'utilizzo di Maven il cui scopo è quello di aiutare lo sviluppatore nella gestione di progetti Java e fornire funzioni di build automation. Nel corso di Architetture Software vengono trattati argomenti, sia di carattere tecnico che metodologico, che spesso hanno a che fare non con sistemi standalone ma distribuiti. Per far simulare allo studente un ambiente distribuito è stato usato Vagrant, in combinazione con Virtual Box, con lo scopo di effettuare in modo automatizzato il provisioning di macchine virtuali. Attraverso questi strumenti lo studente, con uno sforzo di configurazione assai ridotto, potrà creare una rete virtuale in cui testare applicazioni client-server distribuite. Vagrant inoltre ben si presta ad essere integrato nell'ambiente cloud di Amazon. Seguendo una tendenza attuale del mercato, è stato inoltre utilizzato Spring Framework, celebre framework Java lightweight. Nel lavoro di tesi, Spring è stato usato in maniera intensa per la realizzazione dell'interfaccia web e sono state anche testate le capacità del modulo Spring-Integration. Visto l'ampio utilizzo di tale framework si è cercato di fare un analisi architetturale di Spring ed un confronto con Java Platform Enterprise Edition (Java EE). Come strumento di supporto per la gestione delle configurazioni software è stato proposto GitHub, celebre piattaforma di social-coding che permette con estrema facilità a gruppi di

studenti di collaborare suddivisi in team. Un discente, lavorando in piccoli gruppi eterogenei, ha la possibilità di accrescere le proprie competenze come sancito dallo psicologo Vygotskij nell’ambito della teoria del “Cooperative Learning”. Come scenario di caso d’uso è stata ideata un’applicazione di un servizio di car-sharing. Lo scopo di tale applicazione è quello di gestire il noleggio e la prenotazione di un parco auto all’interno del quadrante del Grande Raccordo Anulare di Roma. Un utente può registrarsi al sistema ed effettuare prenotazioni di veicoli. Una volta che un veicolo è stato prenotato, verrà simulata la guida ottenendo dati sulla distanza e sul tempo necessario a completare il percorso secondo il traffico attuale grazie all’interazione con i servizi messi a disposizione da Google Maps. Nell’applicazione è presente un amministratore che ha il compito di gestire le utenze nonché decretare la necessità di effettuare i rifornimenti dei veicoli, pulizia interna o riparazione dell’auto. Un utente può interagire con l’applicazione mediante un’interfaccia web. Parte del lavoro di tesi prevedeva quindi la realizzazione delle funzionalità dell’applicazione in vari modi mantenendo però fissa la parte di presentazione realizzata mediante delle pagine web. Al fine di strutturare l’applicazione in modo consono è stato seguito lo stile architettonicale “Ports and Adapters”. In questo modo è possibile utilizzare servizi/funzionalità di natura diversa senza dover cambiare il resto dell’applicazione. Le varie funzionalità sono state realizzate prima come semplici managed-bean di Spring, poi mediante RMI, Enterprise Java Beans, Web Service (Soap e Rest). E’ stato anche previsto l’erogazione di un caso d’uso che sfrutti JMS. Nel lavoro di tesi le principali difficoltà hanno riguardato la configurazione di ambienti virtuali che fanno uso di application server Java EE come Glassfish e il far colloquiare tecnologie estremamente diverse. Nella realizzazione della presente tesi, in ogni capitolo, si è cercato di inquadrare una possibile problematica e proporre una soluzione andandone ad evidenziare quali sono i punti di forza ed, eventualmente, le debolezze. In totale sono stati redatti 14 capitoli.

Nel primo capitolo è stato motivato l'uso di Maven andando ad illustrare le principali azioni da adottare per facilitare il lavoro di sviluppo.

Nel secondo capitolo è stato affrontato Vagrant con un'ampia discussione su Chef e Puppet, linguaggi alternativi per effettuare il provisioning di macchine virtuali.

Nel terzo capitolo è presente l'analisi architettonale di Spring, spiegando in modo dettagliato quali sono i punti di forza che hanno decretato il successo di tale framework nella comunità degli sviluppatori Java.

Nel quarto capitolo vengono affrontate le problematiche relative alla gestione dei cambiamenti ed alla necessità di utilizzare un sistema per il controllo di versione. Come strumento viene suggerito l'uso di GitHub.

Nel quinto capitolo viene presentato lo studio di caso. Si affronta pertanto l'analisi dell'applicazione di car-sharing andando ad analizzare i requisiti funzionali e di qualità.

Nel sesto capitolo viene illustrato il pattern “Ports and Adapters – Architettura Esagonale” utilizzato per strutturare l'applicazione di car-sharing.

Nel settimo capitolo vengono descritte le scelte progettuali effettuate per l'implementazione dello strato di persistenza e le tecnologie utilizzate.

Scopo dell'ottavo capitolo è quello di spiegare le decisioni assunte nell'implementazione della logica di business del sistema.

Il nono capitolo illustra le scelte per l'implementazione dell'interfaccia web, a livello di decisioni progettuali e di tecnologie adottate.

Nel decimo capitolo sono motivate le scelte effettuate per la realizzazione di una versione dell'applicazione basata su RMI, mentre nell'undicesimo capitolo è spiegato l'uso di JMS.

Nel dodicesimo capitolo è illustrata una versione dell'applicazione basata su EJB.

Nel tredicesimo capitolo viene descritto come sia possibile usare i WebService per erogare le funzionalità dell'applicazione.

CAPITOLO 1: MAVEN

In questo capitolo si illustra Maven come strumento d'ausilio per la gestione di progetti Java complessi.

1.1 Premessa

Nello sviluppo di applicazioni Java complesse, per un programmatore, gestire in modo manuale le librerie è difficoltoso e può comportare diversi errori. Per aiutare lo sviluppatore nella gestione di un progetto è stato introdotto Maven. Maven è riconosciuto come un rivale di Apache Ant ma, rispetto a quest'ultimo, fornisce funzionalità più avanzate mantenendo una estrema facilità di utilizzo. Molti IDE hanno specifici plug-in per aiutare il programmatore ad utilizzare Maven che, pertanto, non deve utilizzare la riga di comando. Visto il sempre più crescente utilizzo di tale strumento dalla comunità degli sviluppatori, nell'ambito di questa tesi è stato proposto l'utilizzo di Maven. In questo modo gli studenti non dovrebbero incontrare difficoltà nella configurazione e gestione di un progetto Java ed inoltre verrebbero a conoscenza di uno strumento prezioso per l'arricchimento del loro bagaglio culturale.

1.2 Cos'è Maven



Maven è un potente strumento open source per la gestione di progetti Java, in termini di compilazione del codice, distribuzione e documentazione. Maven è basato sul concetto di Project Object Model (POM), un file XML, che descrive le dipendenze tra il progetto e le varie versioni di librerie e standardizza la struttura delle directory. In questo modo si separano le librerie dalla directory di progetto utilizzando questo

file descrittivo per definirne le relazioni. Maven effettua automaticamente il download di librerie Java e plug-in Maven da vari repository per poi scaricarli in locale o in un repository centralizzato lato sviluppo. Questo permette di recuperare in modo uniforme i vari file JAR e di poter spostare il progetto da un ambiente all'altro avendo la sicurezza di utilizzare sempre le stesse versioni delle librerie. Apprenderne il suo uso può semplificare di molto il difficile compito di compilazione e di gestione dei progetti Java viste le numerose funzionalità che esso offre. Ne deriva un'estrema semplificazione del processo: con Maven non è più necessario conoscere nel dettaglio i meccanismi di compilazione.

I suoi vantaggi principali sono:

- Standardizzazione della struttura di un progetto e del processo di compilazione.
- Test ed esportazione automatizzati.
- Gestione e download automatico delle librerie necessarie al progetto con risoluzione delle eventuali dipendenze transitive.
- Facilità di ampliarne le funzionalità iniziali tramite l'utilizzo di plugin.
- Creazione automatica di un semplice sito di documentazione del progetto.

1.3 Breve storia di Maven

L'idea iniziale, come spesso accade nel mondo dell'Open Source è nata dall'esigenza di risolvere un problema pratico. La realizzazione di Maven è partita nell'ambito del progetto Jakarta Alexandria (attualmente abbandonato) per poi migrare all'interno del progetto Turbine (framework Servlet disegnato per la rapida produzione di applicazioni web). L'obiettivo iniziale era di implementare un tool per semplificare, uniformare e automatizzare il processo di build di sistemi complessi. Per ottemperare a tali scopi serviva sia creare un modello di progetto, sia una struttura di file

system standard. Era necessario anche fare in modo che i diversi progetti Apache funzionassero in maniera analoga. Prima del rilascio di Maven, infatti, ciascun progetto Apache, presentava differenti approcci alla compilazione, alla distribuzione e alla generazione del sito web relativo al progetto, creando evidenti problemi di allocazione delle risorse umane, di riutilizzo di script, di implementazione di best practice. Questi problemi non erano di poco conto, visto il molto tempo speso dagli sviluppatori per configurare i vari ambienti, per comprenderne il funzionamento e per mantenere i vari script di compilazione. Oltre a creare elementi di distrazione dal loro obiettivo principale (realizzare sistemi software di elevata qualità), tutto ciò spesso finiva anche per far desistere nuovi potenziali collaboratori. Dato l'elevato livello di interdipendenza dei progetti open-source, la capacità di Maven di fornire una struttura comune per i vari progetti, di permetterne il reperimento dei file di distribuzione attraverso un meccanismo di repository condiviso, ha rappresentato un elemento chiave che ne ha assicurato il successo fin dalle prime versioni. Già Ant (strumento simile a Maven) aveva risolto molti problemi relativi al processo di build. Maven ha permesso di effettuare l'ulteriore passo in avanti, risolvendo le rimanenti problematiche presenti nello sviluppo di sistemi complessi basati su sotto-progetti interdipendenti. Dalla versione iniziale, Maven ha fatto molti progressi e ha contribuito enormemente a semplificare le quotidiane attività del team di sviluppo. [MOKB]

1.4 Obiettivi primari di Maven

Gli obiettivi principali di Maven sono:

1. **Semplificazione del processo di build dei sistemi Java.** In particolare, Maven si fa carico di risolvere tutta una serie di dettagli senza ricorrere all'utilizzo di file di script,

- 2. Sviluppo di un ambiente uniforme di build.** Maven gestisce progetti basati sul proprio modello a oggetti del progetto (POM, Project Object Model). Sebbene molti aspetti di Maven possano essere personalizzati, Maven dispone di una serie di "standard", studiati per essere efficacemente utilizzati per i vari progetti. Pertanto, una volta compreso il funzionamento di un progetto gestito da Maven, si sa come gestirne quasi ogni altro (a meno di forti personalizzazioni). Il che si traduce in un notevole risparmio di tempo, energie e frustrazione.
- 3. Produzione di informazioni qualitative circa il progetto.** Sebbene Maven non sia né uno strumento di documentazione né un generatore di siti web, rappresenta un'ottima infrastruttura che permette di utilizzare tutta una serie di plug-in per la generazione di informazioni utili relative ai progetti. Per esempio, permette di generare documenti relativi alle variazioni effettuate (interagendo con i sistemi di source control), ai riferimenti incrociati dei sorgenti, alle mailing list, alle dipendenze, ai rapporti relativi alla copertura del codice da parte dei test, etc. Il tutto in maniera assolutamente trasparente e automatica.
- 4. Erogazione di linee guida corredate da un opportuno supporto per l'applicazione di best practice per lo sviluppo di sistemi.** L'esempio più evidente è relativo alla presenza esplicita, nel processo di build standard, della definizione ed esecuzione dei test di unità.
- 5. Supporto alla migrazione verso nuove feature.**

1.5 Vantaggi derivati dall'utilizzo di Maven in un progetto

Date le caratteristiche di Maven dovrebbero essere ormai chiari quali siano i vantaggi dell'utilizzo di tale strumento:

- **Coerenza:** le varie organizzazioni possono standardizzare la gestione dei progetti Java utilizzando l'insieme di best practice alla base di Maven. Accettando una serie di standard, si accede a tutta una serie di servizi

predefiniti. Il livello di qualità dei vari progetti, inevitabilmente, si eleva; i progetti stessi diventano più trasparenti, si minimizza il tempo necessario per comprendere i vari progetti e si facilita il movimento delle risorse umane.

- **Riutilizzo:** si tratta di uno degli elementi alla base di Maven, il cui uso è già esso stesso un primo riutilizzo di best practice. Un ulteriore livello di riutilizzo è garantito dal fatto che la business logic è encapsulata in moduli (plug-in);
- **Maggiore agilità:** utilizzando Maven si semplifica il processo di generazione di nuovi componenti, di condivisione di file eseguibili ed inoltre la curva di apprendimento di ciascun progetto viene incredibilmente ridotta.
- **Semplificazione della manutenzione:** non è più necessario investire tempo e risorse per manutenere gli ambienti e script di build, i quali, oltre ad essere standardizzati, sono gestiti da Maven.

1.6 I principi base di Maven

L'idea base durante la progettazione di Maven era di creare un linguaggio condiviso per lo sviluppo di progetti software basati su Java ovvero ciò che Christopher Alexander ha formalmente definito pattern. Avere un linguaggio condiviso ad alto livello permette agli sviluppatori di trattare ad un elevato livello di astrazione quei progetti la cui struttura sia fortemente standardizzata. Ne consegue una rete di comunicazioni più efficaci, dei servizi automatizzati e quindi un miglioramento della qualità generale e della produttività del progetto. Definire un pattern per lo sviluppo dei progetti software richiedeva i seguenti principi base:

- convenzioni sulla configurazione (convention over configuration) e in particolare:

- organizzazione standard della directory dei progetti (ubicazione delle risorse del progetto, dei file di configurazione, di quelli generati, della documentazione, etc.);
- definizione del vincolo base secondo cui un progetto Maven genera un solo output (file di distribuzione). Ciò porta naturalmente alla realizzazione di file pom.xml con un obiettivo limitato e ben definito, e quindi all'applicazione del principio di separazione delle responsabilità;
- convenzione sui nomi;
- esecuzione dichiarativa;
- riutilizzo di logica di build;
- organizzazione coerente delle dipendenze.

Per quanto concerne il primo punto, l'idea base è che se si accettano una serie di valide convenzioni (frutto di lungo studio ed esperienza), si ottiene una serie di vantaggi, non solo legati al fatto che lo stesso problema è già stato ben analizzato e validamente risolto. Per esempio, se si organizza la struttura dei propri progetti in base allo standard Maven, quest'ultimo è in grado di fornire automaticamente una serie di servizi, come la generazione dei file di progetto per i vari IDE, la compilazione automatica, etc.

Convention over configuration non è altro che la ripetizione di una serie di semplici regole di buon senso relative all'applicazione di standard. In particolare, l'applicazione di standard fa risparmiare tempo, semplifica la comunicazione, facilita il reperimento di risorse, permette di creare ulteriore valore aggiunto partendo da solide infrastrutture, permette di condividere il lavoro.

Per quanto riguarda l'esecuzione dichiarativa (punto 2) basti dire che tutto in Maven ha una natura dichiarativa. L'elemento base di Maven, il file pom.xml, è probabilmente l'esempio più classico di struttura dichiarativa.

Con una decina di righe di carattere dichiarativo, è possibile compilare, verificare, generare la documentazione e il file di distribuzione di semplici progetti.

Per quanto attiene il punto 3, Maven semplifica enormemente il riutilizzo del codice, incapsulando la business logic in appositi moduli: i plug-in, di cui esiste un insieme base (come quello per compilare, per eseguire i test, per creare file .jar, etc.) oltre a un insieme vastissimo di plug-in forniti da terze parti.

Per l'organizzazione coerente delle dipendenze (punto 4), questa è ottenuta attraverso una serie di meccanismi di repository. In primo luogo esiste una sezione all'interno del file pom.xml, dedicata alla dichiarazione delle dipendenze. Una volta definita una nuova dipendenza nel pom.xml, non è necessario copiare il relativo file in un'apposita directory del progetto, poiché risiederà in un repository. La presenza dei repository risolve una serie di problemi tipici riguardanti l'indicazione dei path e le diverse versioni delle librerie.

Maven dispone di due tipi di repository: locale e remoto. Durante il normale funzionamento, Maven, interagisce con il repository locale; qualora però una dipendenza non sia presente all'interno di tale repository, Maven si occupa di consultare i repository remoti ai quali ha accesso, al fine di risolvere la dipendenza mancante.

1.7 Concetti di Maven

Di seguito alcuni concetti basilari di Maven utilizzati nel lavoro di tesi.

1.7.1 File di POM

POM sta per Project Object Model e ogni singolo progetto è descritto attraverso il file di configurazione pom.xml, senza il quale Maven non

può fare nulla. Il POM è un file XML e guida l'esecuzione in Maven definendo in modo chiaro l'identità e la struttura di un progetto in ogni suo aspetto. Tutto è descritto nel POM: informazioni generali del progetto, dipendenze, processo di compilazione e fasi secondarie come la generazione di documentazione.

Un POM ai minimi termini è composto da un tag root <project> che conterrà i tag:

- <modelVersion> che dichiara a quale versione di POM questo progetto è conforme
- <groupId> ID del gruppo del progetto
- <artifactId> ID dell'artefatto (del progetto)
- <version> cioè la versione del progetto
- <packaging> che è il tipo di archivio che vogliamo esportare (jar, war o ear)

I parametri **groupId**, **artifactId**, **version** e **packaging** identificheranno univocamente un progetto. Se packaging non è specificato nel POM, assumerà “jar” a valore di default.

Ecco un esempio di pom.xml minimale con packaging non specificato:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

Per ogni dipendenza è possibile anche definire uno scope:

- **compile** (default) – le dipendenze sono disponibili in tutti i classpath del progetto

- **provided** – è simile a compile, ma prevede che a runtime le dipendenze siano rese disponibili dall’ambiente di esecuzione (per esempio le JavaEE APIs per un’applicazione enterprise)
- **runtime** – le dipendenze sono richieste solo in esecuzione
- **test** – le dipendenze sono richieste solo per la compilazione e l’esecuzione dei test
- **system** – la dipendenza non viene recuperata tramite repository, ma ne viene esplicitamente dichiarata la posizione locale

Ad esempio:

```
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.13</version>
  <scope>test</scope>
</dependency>
```

1.7.2 Plugin e Goal

Un **goal** è una singola funzione che può essere eseguita sul progetto, l’equivalente Maven dei task Ant. I Goal possono essere sia specifici per il progetto dove sono inclusi, sia riusabili. I **Plugin** sono goal riutilizzabili in tutti i progetti.

Maven ha una serie di plugin built-in disponibili, i principali sono i seguenti:

- **clean**: che permette di cancellare i compilati dal progetto;
- **compiler**: che permette di compilare i file sorgenti;
- **deploy**: che permette di depositare il pacchetto generato nel repository remoto;
- **install**: che permette di depositare il pacchetto generato nel repository locale;

- **site**: che permette di generare la documentazione del progetto;
- **archetype**: che permette di generare la struttura di un progetto a partire da un template.

Ciascun plugin mette a disposizione dei goal specifici o più di un goal (per esempio, il plugin di compilazione, **compiler**, prevede un goal per la compilazione del codice sorgente e un altro per la compilazione dei casi di test). Ciascun goal, a sua volta, riceve in ingresso specifici parametri, facoltativi o obbligatori.

Altri plugin possono essere realizzati qualora sia necessario estendere ulteriormente le capacità di Maven a causa di particolari esigenze.

1.7.3 L'archetipo

Una delle caratteristiche particolarmente apprezzate di Maven è quella di fornire un insieme di standard che rendono possibile l'applicazione di tutta una serie di "buone pratiche". Uno di questi standard è costituito dalla struttura della directory del progetto, denominata archetype (archetipo). Coerentemente con l'impostazione di Maven, applicare tale standard non è obbligatorio, sebbene sia consigliabile: Maven fornisce una serie di strumenti che permettono di utilizzare strutture diverse e, in casi molto particolari, è anche possibile non conformarsi agli standard. Occorre comunque ricordare che deviare dalle impostazioni standard tende a invalidare, o comunque a ridurre, la portata di tutta una serie di vantaggi, quali quelli tipici derivanti dal fatto che si tratti di uno "standard":

i vari servizi di Maven sono predisposti per funzionare con questa struttura e quindi nessuna ulteriore operazione è richiesta per farli funzionare;

1. tale standard permette di uniformare i vari progetti presenti all'interno di un'organizzazione;
2. tale struttura semplifica l'apprendimento dei progetti e quindi favorisce lo spostamento del personale;
3. si evitano inutili perdite di tempo legate a discussioni relative alla struttura del progetto ("dove posizioniamo il file X ?"), alla necessità di creare nuove directory per memorizzare determinati file.

Non va dimenticato poi che ci sono vantaggi legati al fatto che l'archetipo è il risultato di un attento e prolungato studio. Quindi, invece di cercare di inventarsi l'ennesima struttura, con il rischio peraltro di ottenere risultati modesti o addirittura problematici, è decisamente più facile e conveniente beneficiare di un disegno derivante da una lunga esperienza che include anche l'uso di plug-in e di svariati tool. Pertanto, l'utilizzo della struttura standard elimina alla radice il rischio di utilizzare un'organizzazione delle cose poco adatta all'uso d'importanti tool di sviluppo/plug-in.

1.7.4 Ereditarietà

Un'altra caratteristica molto potente e unica di Maven è la possibilità di relazionare tra loro i progetti con **legami di ereditarietà**, un po' come avviene con le classi Java. In Maven l'ereditarietà permette di creare nuovi **POM** file che ereditano le relazioni definite in un altro POM genitore detto **super POM**. I progetti genitori e quelli esclusivamente aggregabili devono specificare il valore pom all'interno del campo packaging. Questo campo serve in generale per indicare la natura del prodotto finale. Gli elementi ereditabili dai discendenti che si possono specificare in un POM genitore sono:

- dipendenze

- lista degli sviluppatori e contributori
- esecuzioni dei plugin con id corrispondenti
- configurazione dei plugin
- risorse

1.8 Maven ed Eclipse

E' possibile utilizzare Maven in Eclipse senza ricorrere all'installazione di Maven sul computer. Esiste infatti un plugin chiamato m2e che permette di sfruttare la potenza di Maven in Eclipse. E' possibile trovare ed installare il plugin m2e sul marketplace.

CAPITOLO 2 : GESTIONE DI GRUPPI DI MACCHINE VIRTUALI

In questo capitolo viene illustrato l'utilizzo di Vagrant nell'ambito del lavoro di tesi. Saranno spiegati anche Chef e Puppet, linguaggi di provisioning utilizzati per configurare agevolmente ed efficacemente delle Vagrant Box.

2.1 Premessa

Nell'ambito del lavoro di tesi, un obiettivo da raggiungere era quello di riuscire a simulare, in un ambiente virtualizzato, le tecnologie affrontate nel corso di Architetture Software. Attualmente vengono messi in commercio personal computer con un quantitativo di Ram sempre maggiore. Il mercato attuale, ad un costo non troppo elevato, offre computer dai 4GB agli 8GB di RAM. Oltre che per eseguire le moderne applicazioni, un tale quantitativo di RAM, rende l'utilizzo di Virtual Machine su personal computer molto più facile e fattibile. Questo ha aperto alcune opzioni interessanti nel ciclo di sviluppo di qualsiasi applicazione o servizio in quanto ora è possibile avere una macchina di test/sviluppo su ogni personal computer. Un amministratore di sistema vorrebbe che queste macchine virtuali siano il più simile possibile all'ambiente di produzione e **Vagrant** rappresenta un buon modo per gestire e distribuire macchine virtuali agli sviluppatori. Vagrant inoltre è compatibile con Amazon AWS e, pertanto, permette con estrema facilità di creare ambienti virtuali su Amazon Web Services EC2. E' possibile rilasciare macchine virtuali nell'ambiente di Amazon con la stessa facilità con la quale si creano virtual machine sul proprio PC. Tale modo di operare risulta particolarmente efficace quando non si dispone di personal computer particolarmente performanti. Vista la grande versatilità di Vagrant, nonché il suo crescente utilizzo nella comunità

degli sviluppatori, si è ritenuto opportuno utilizzare tale strumento nell’ambito del lavoro di questa tesi.

2.2 Cos’è Vagrant



VAGRANT

Vagrant è un software open-source per la creazione e la configurazione di ambienti di sviluppo virtuali, fornisce un ambiente di lavoro facile da configurare, riproducibile, portatile, costruito con tecnologie standard del settore e controllato da un unico flusso di lavoro coerente per aiutare a massimizzare la produttività e la flessibilità di un singolo sviluppatore o di un team di programmati.

Per ottemperare ai suoi scopi, “*Vagrant sta sulle spalle di giganti*”. Le macchine sono gestite mediante VirtualBox, VMware, AWS, o un qualsiasi altro provider. Strumenti standard di provisioning e configurazione come script in shell, Chef o Puppet, possono essere utilizzati per installare e configurare il software sulla macchina automaticamente.

In sintesi Vagrant è un sistema per riprodurre e rendere portabili degli ambienti di sviluppo. Viene scaricata una macchina virtuale e, grazie al software di virtualizzazione (per esempio Virtual Box), si ha un server di testing privato e configurato secondo le specifiche, magari uguale a quello di produzione. La cosa formidabile è che grazie a **Vagrant** questo ambiente può essere riprodotto facilmente sulle macchine del team di sviluppo.

2.3 Perché Vagrant

Utilizzare Vagrant permette di fornire a tutti gli sviluppatori del team una macchina virtuale con la configurazione sempre allineata, in quanto creata a partire da file che sono sotto controllo di versione assieme al progetto in fase di sviluppo. Gli stessi file possono essere utilizzati anche per il server di produzione, garantendo quindi un perfetto allineamento tra i due ambienti.

Vagrant comporta ripercussioni positive per lo sviluppatore, designer e per l'ingegnere di sistema.

Lo **sviluppatore** non si dovrà più preoccupare di errori del software derivanti da differenti ambienti di esecuzione. Vagrant permette di isolare le dipendenze dalla loro configurazione in un unico ambiente coerente senza porre limitazioni e vincoli agli strumenti normalmente usati da un programmatore. Una volta che la configurazione dell'ambiente in cui si intende rilasciare l'applicazione è stata definita in un file, chiamato Vagrantfile, è completata, è sufficiente avviare Vagrant per avere una macchina virtuale che ha tutti gli strumenti settati ed installati pronti per essere utilizzati. E' possibile condividere il Vagrantfile tra gli sviluppatori dello stesso team per assicurare che ognuno lavori sulla stessa tipologia di macchina virtuale. Ogni membro del team può continuare ad usare il suo sistema operativo e strumenti di produzione preferiti creando comunque un codice che funzioni per la macchina di produzione che dovrà ospitare l'applicazione finale.

Vagrant, ad un **ingegnere di sistema**, garantisce un ambiente coerente sul quale rilasciare applicazioni o script per la gestione dell'infrastruttura. E' possibile configurare una macchina virtuale usando shell script, oppure mediante Chef cookbooks o moduli Puppet

sfruttando la virtualizzazione offerta da strumenti altamente performanti quali VirtualBox o VMware.

Vagrant comporta benefici anche per i **designer**. Tale strumento imposterà automaticamente tutte le risorse necessarie per avviare una webapp ed il designer si potrà occupare solo del design. Una volta che uno sviluppatore ha configurato Vagrant, non è più necessario preoccuparsi di come mandare in esecuzione un'applicazione. Sarà Vagrant stesso che, una volta avviato, provvederà a mandare in esecuzione l'applicazione. Questo permette ad un designer di modificare la parte grafica di un sistema senza dover continuamente interagire con gli sviluppatori. Il designer può quindi modificare la parte di codice di sua competenza e vederne direttamente gli effetti con una interazione minima, se non nulla, con gli altri componenti del team di sviluppo.

2.4 Requisiti - Hypervisor

Per iniziare ad usare Vagrant bisogna, per prima cosa, installare un hypervisor. Quelli supportati sono molti ma, il predefinito, ed è quello che è stato usato nella tesi, è **VirtualBox**. Successivamente è necessario installare anche **Vagrant**. A questo punto è tutto praticamente pronto per avvivare una prima macchina virtuale.

2.5 Concetti base

2.5.1 Vagrant Box

Il primo concetto fondamentale da capire per quanto riguarda **Vagrant** è quello di Box. Una **Box** è un vero e proprio contenitore che contiene il sistema operativo nonché i software di base ed ogni loro configurazione. Una box è l'immagine di base utilizzata per creare un ambiente virtuale con Vagrant. E' pensata per essere un file portatile

che può essere utilizzato da altri, su qualsiasi piattaforma su cui Vagrant possa funzionare, per far partire un ambiente virtuale. Esistono già diverse box con delle configurazioni di base e sono rese disponibili dai server di Vagrant. Per aggiungere a Vagrant una specifica box il comando è il seguente.

```
$ vagrant box add hashicorp/precise32
```

In questo modo dal Vagrant Cloud verrà scaricata una box chiamata “hashicorp/precise32”. Tale box è quella che, prevalentemente, è stata usata nella tesi ed è una semplice macchina virtuale a 32 bit con Ubuntu 12.04 con il supporto a Chef e Puppet pre-installato. I box forniscono solo l’immagine di base per Vagrant. Nel momento in cui si esegue **Vagrant up**, comando per avviare la creazione ed il boot di una macchina virtuale, il box viene copiato in modo che possa essere poi modificato per quella specifica macchina virtuale. Pertanto, è possibile rimuovere o aggiornare il box dopo che una macchina virtuale è stata creata.

2.5.2 Vagrantfile

Il concetto più importante di vagrant è il **Vagrantfile** perché permette di descrivere il tipo di macchina richiesta per il progetto nonché come configurare ed effettuare il provision di tale macchina. Questo file è nominato in questo modo perché Vagrantfile è proprio il nome che il file, contenente una configurazione Vagrant, deve avere. Tale file è scritto mediante Ruby ma non è necessario conoscere in modo approfondito tale linguaggio di programmazione per configurare una o più macchine virtuali. Di seguito, un esempio minimale di configurazione mediante Vagrantfile.

```

# -*- mode: ruby -*-
# vi: set ft=ruby :

# Vagrantfile API/syntax version. Don't touch unless you know what you're doing!
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # Every Vagrant virtual environment requires a box to build off of.
  config.vm.box = "hashicorp/precise32"
end

```

Escludendo le righe che iniziano con #, rappresentanti dei commenti, il frammento di codice precedente permette di configurare una macchina Ubuntu con supporto per Chef e Puppet. Il “2” nella prima riga effettiva di codice specifica la versione dell’oggetto **config** mediante il quale è possibile dettare le regole di configurazione della macchina all’interno della sezione **do – end**.

Config.vm.box specifica quale box deve essere avviata.

A questo punto il comando da lanciare da riga di comando per avviare tale macchina virtuale è **vagrant up**. Dopo aver lanciato tale comando Vagrant provvederà a fare il boot della macchina con le caratteristiche specificate. Terminato il boot, per essere operativi all’interno della macchina, è sufficiente effettuare un **vagrant ssh**.

2.5.3 Cartelle sincronizzate

Questo concetto è molto utile quando si lavora con le box. Le cartelle sincronizzate consentono a Vagrant di sincronizzare una cartella sul computer host sulla macchina guest, permettendo di continuare a lavorare sul file del progetto sul computer host, ma utilizzando le risorse nella macchina guest per compilare o eseguire il progetto. Per impostazione predefinita, Vagrant condividerà la directory del progetto (la directory con il Vagrantfile) nella directory /vagrant.

```
config.vm.synced_folder "src/", "/srv/website"
```

Il comando appena riportato permette di sincronizzare una cartella. Il primo parametro è il path di una cartella sulla macchina host. Il secondo parametro è un path assoluto ed indica in qualche cartella sulla macchina virtuale deve essere sincronizzata la cartella della macchina host. Se tale cartella non esiste nella macchina virtuale verrà creata.

Le cartelle sincronizzate, ovviamente, sono state molto utilizzate nell'ambito del lavoro di tesi. Un programma, distribuito sotto forma di runnable jar file, può essere collocato nella cartella della macchina host che è sincronizzata, come da configurazione, con una directory della macchina guest. Operando in questo modo i file saranno immediatamente disponibili anche nella macchina virtuale pronti per essere usati.

2.6 Provisioning

L'avvio di una macchina virtuale vuota non è molto utile, per questo motivo Vagrant supporta il provisioning delle macchine virtuali attraverso l'uso di provisioners. Vagrant infatti non è solo un sistema per il rapido avvio di macchine virtuali ma, come già detto, permette di controllarne le impostazioni nel tempo insieme al software installato. E' possibile gestire la configurazione automatica attraverso una struttura di provisioning abbastanza flessibile che supporta normali shell script (Bash), Puppet, Chef e Ansible. Al termine delle operazioni di configurazione, indipendentemente dallo strumento usato, il risultato sarà quello di una macchina virtuale con tutto lo stack software che è di interesse avere in quello specifico ambiente. Nel lavoro di testi è stato usato con grande frequenza Chef ma talvolta è stato fatto ricorso a Puppet.

2.6.1 Chef



Come riporta la documentazione ufficiale, Chef di Opscode è un sistema che permette di trasformare un server, sia esso fisico o virtuale, in codice. Chef permette di automatizzare il modo di creare, gestire e rilasciare l'infrastruttura. L'infrastruttura stessa diventa pertanto testabile, ripetibile e persino soggetta a controllo di versione proprio come accade con il codice di programmazione.

Chef si basa su definizioni riutilizzabili, dette recipes (ricette), per automatizzare le attività di configurazione dell'infrastruttura. Esempi di recipes sono le istruzioni per configurare un server web, database o un meccanismo di load balancing. Le ricette descrivono in cosa consiste l'infrastruttura e come ogni parte della stessa debba essere rilasciata, configurata e gestita. Le ricette a loro volta sono costituite da risorse (resources). Ogni risorsa descrive un frammento dell'infrastruttura come un file, un template o un package che deve essere installato.

Spesso Chef è utilizzato in modalità **client server**, dove un client (chiamato nodo) accede ad un repository (server) per recuperare le "ricette" da eseguire sulla macchina. Nel nostro caso useremo **Chef-Solo** che ci esonera dal dover avere un repository remoto, ma di contro dobbiamo avere tutte le "ricette" in una cartella. La cartella che deve contenere tali ricette è chiamata **cookbooks**. Per usare Chef è quindi necessario dare la seguente struttura alla cartella.

```
// tree -L 2
.
├── cookbooks
│   ├── apt
│   ├── build-essential
│   ├── java
│   ├── nodejs
│   ├── openssl
│   ├── postgresql
│   ├── rvm
│   └── tomee
└── Vagrantfile
└── www
```

La cartella di un progetto Vagrant avrà quindi, oltre al `Vagrantfile`, una directory chiamata `cookbooks`. All'interno di `cookbooks` avremo i vari cookbook come ad esempio quello di `java` oppure di `glassfish`. Ogni cookbook può essere composto da diverse recipes. In `glassfish` ad esempio esistono 4 tipi di recipes: **default** (effettua il download e l'estrazione del file binario di `glassfish` e crea un utente ed un gruppo), **attribute_driven_domain** (configura 0 o più domini di `glassfish` usando l'attributo `glassfish/domains`), **attribute_driven_mq** (configura 0 o più broker GlassFish OpenMQ usando l'attributo `openmq/instances`) e **search_driven_domain** (configura 0 o più domini di GlassFish usando search per generare la configurazione). Ogni recipe è personalizzabile mediante le **resources**.

Vista la struttura di una configurazione Chef è necessario capire come possa essere usata in coabitazione con Vagrant.

```

def define_node(config, name, glassfish_config)
  config.vm.define name do |client|
    client.vm.hostname = "#{name}-vm"

    client.vm.provision :chef_solo do |chef|
      chef.add_recipe 'apt'
      chef.add_recipe 'java::default'
      chef.add_recipe 'glassfish::attribute_driven_domain'

      chef.json = {
        'java' => {
          'install_flavor' => 'oracle',
          'jdk_version' => 7,
          'oracle' => {
            'accept_oracle_download_terms' => true
          }
        },
        'glassfish' => glassfish_config
      }
    end
  end
end

```

Per prima cosa è necessario indicare che come strumento di provisioning si usa Chef e successivamente specificare le recipes che si intende adottare per la configurazione.

Il comando **client.vm.provision :chef_solo** sta ad indicare che si usa Chef-Solo mentre **java::default** indica che all'interno del cookbook java, presente all'interno della cartella cookbooks, va utilizzata la recipe default. L'elenco di tutti i recipe è chiamato “**run-list**”. Lo stesso ragionamento si applica al caso di glassfish. Come detto, è anche possibile andare a personalizzare ogni recipes e questo è possibile farlo mediante JSON.

2.6.2 Puppet



Come Chef, anche Puppet è uno strumento per automatizzare la gestione delle infrastrutture. Il suo scopo è quello di automatizzare le attività ripetitive per quanto riguarda il setup di ambienti di

produzione e distribuzione di applicazioni mediante la definizione di un flusso di attività e di regole che devono essere applicate per ottenere la configurazione desiderata. Puppet è uno strumento molto efficace ma, rispetto a Chef, è più complicato. Nell’ambito del lavoro di tesi, Puppet è stato usato per configurare una vagrant box con Glassfish.

Puppet funziona con un proprio linguaggio di configurazione che è scritto in file chiamati “**manifests**” che hanno estensione .pp . Per mantenere il tutto nel modo più ordinato possibile è opportuno creare una cartella chiamata **puppet** nella directory principale del progetto in questione. All’interno della cartella puppet è necessario creare una directory chiamata **manifests** dove andremo a posizionare i file manifest. All’interno dei file di manifest sono definite delle classi che specificano delle risorse.

```
Exec { path => [ "/bin/", "/sbin/" , "/usr/bin//", "/usr/sbin/" ] }

class system-update {
  exec { 'apt-get update':
    command => 'apt-get update',
  }

  $sysPackages = [ "build-essential" ]
  package { $sysPackages:
    ensure => "installed",
    require => Exec['apt-get update'],
  }
}

class apache {
  package { "apache2":
    ensure => present,
    require => Class["system-update"],
  }

  service { "apache2":
    ensure => "running",
    require => Package["apache2"],
  }
}

include apache
include system-update
```

Nel frammento di codice precedente sono definite due classi: system-update ed apache. E’ possibile anche dichiarare delle variabili, si veda ad esempio \$syspackage che, in questo caso, è un array di package che necessitano di essere installati. L’ordine con il quale sono definite le classi non ha nessuna influenza sul modo con il quale puppet esegue i

comandi. Le dipendenze possono far variare l'ordine di esecuzione dei task in modo dinamico.

Per utilizzare Vagrant, nel Vagrantfile, è necessario inserire un frammento di configurazione simile al seguente.

```
config.vm.provision "puppet" do |puppet|
  puppet.manifests_path = 'puppet/manifests'
  puppet.manifest_file = 'site.pp'
  puppet.module_path = 'puppet/modules'
end
```

Mediante tali semplici righe di configurazione indicheremo che verrà usato Puppet quale strumento di provisioning. Al fine di migliorare l'attività di provisioning, è consigliato distribuire la configurazione in più file, chiamati **Puppet Modules**. Per tale scopo è necessario creare, all'interno della cartella puppet, una directory di nome **modules** che dovrà contenere i vari moduli. Per modulo s'intendono directory e file organizzati secondo una specifica struttura. I file di manifest presenti all'interno di un modulo devono rispettare alcuni vincoli sul loro nome.

Nel Vagrantfile è necessario specificare che tutti i moduli necessari al provisioning del sistema si trovano nel path “puppet/modules”. Tale configurazione è specificata mediante il comando:

```
puppet.module_path = "puppet/modules"
```

I moduli devono inoltre rispettare alcuni vincoli:

- Ogni modulo deve essere contenuto in una directory ed il nome della directory è il nome del modulo.
- Ogni directory deve avere una sotto-directory di nome manifest che contiene tutti i file con estensione .pp

- Il file manifest deve avere nome init.pp, deve essere contenuto in una directory di nome manifests, ed al suo interno ci deve essere la definizione di una singola classe che deve avere lo stesso nome del modulo.

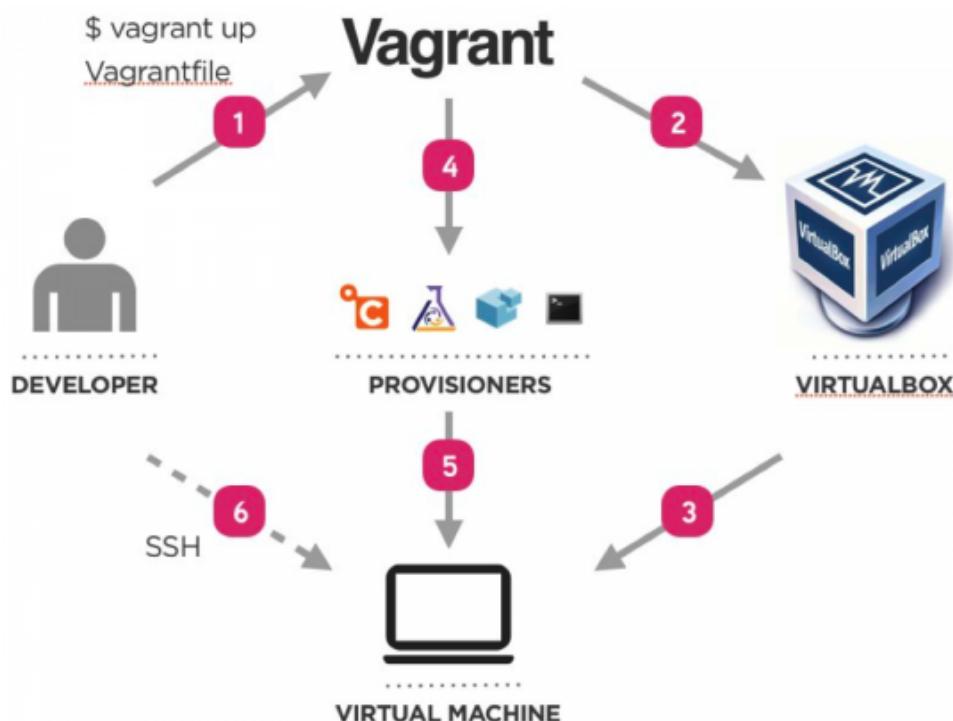
Si supponga di avere un progetto Vagrant che fa uso di Puppet con la configurazione suddivisa in due moduli: system-update e apache. Un esempio di strutturazione di directory per un tale progetto potrebbe essere il seguente.

- Project Root (./)
 - puppet
 - manifests
 - site.pp
 - modules
 - apache
 - manifests
 - init.pp
 - system-update
 - manifests
 - init.pp
 - Vagrantfile
 - (...)

Come indicato nella configurazione di Vagrant i vari moduli di Puppet si trovano all'interno della directory puppet/modules mentre il file di manifest principale, dal nome site.pp, è contenuto in puppet/manifest.

2.7 Ciclo di sviluppo

A questo punto, avendo passato brevemente in rassegna tutti gli strumenti che possono essere coinvolti nell'utilizzo di Vagrant, è utile fornire un quadro d'insieme per capire qual è il flusso di lavoro in un progetto che fa uso di tale tecnologia per gestire ambienti virtualizzati.



Lo sviluppatore interagisce con Vagrant utilizzando alcuni semplici comandi. Il principale comando usato è “vagrant up” (1) che ha lo scopo di effettuare il booting della macchina virtuale ed equivale al pulsante di accensione di una macchina fisica. Lo sviluppatore deve provvedere a fornire un file denominato “Vagrantfile” che contiene alcune direttive di base per creare una macchina virtuale con le caratteristiche necessarie ad ospitare l'applicazione che si intende sviluppare. A questo punto Vagrant interagirà con un hypervisor (solitamente VirtualBox) dandogli le istruzioni per la creazione di una macchina virtuale (2-3). (4) Vagrant può anche interagire con strumenti di provisioning (come Chef o Puppet) al fine di installare tutto il software richiesto per rendere eseguibile il

progetto all'interno della macchina virtuale (5). Una volta che la macchina virtuale è stata creata, avviata e configurata, lo sviluppatore può accedervi (6) attraverso SSH. Questo metodo d'accesso ricorda quello verso un server remoto. Nel caso di Vagrant il server remoto risiede però in una macchina virtuale posizionata all'interno del personal computer dello sviluppatore.

CAPITOLO 3: ANALISI ARCHITETTURALE SPRING FRAMEWORK

In questo capitolo viene effettuata l’analisi architetturale di Spring, celebre framework Java lighweight. Nel lavoro di tesi, Spring è stato usato in maniera intensa per la realizzazione dell’interfaccia web.

3.1 Premessa

Nell’ambito di questa tesi è stato proposto l’utilizzo di Spring Framework. Spring è la principale tecnologia Java per la costruzione di un’architettura a componenti elastica, mantenibile ed indipendente dal contesto in un qualunque software Java. Spring è il framework opensource più ampliamente utilizzato nel mondo Java per realizzare codice semplice e caratterizzato da un’architettura elegante, potente, stabile e manutenibile nel tempo.

3.2 Cos’è Spring

Spring è un framework open-source creato da Rod Johnson e descritto nel libro *Expert One-on-One: J2EE Design and Development*. Spring è stato creato per affrontare lo sviluppo di applicazioni enterprise mediante plain-old Java objects (POJOs) senza la complessità degli Enterprise Java Beans (EJB). Tale framework può essere usato con successo per qualsiasi applicazione Java e non solo per sviluppare sistemi lato-server. Milioni di sviluppatori nel mondo usano il Framework Spring per creare codice avente le seguenti caratteristiche: alte performance, facilmente testabile e riusabile. Per semplificare lo sviluppo di sistemi in Java, Spring utilizza quattro principi fondamentali:

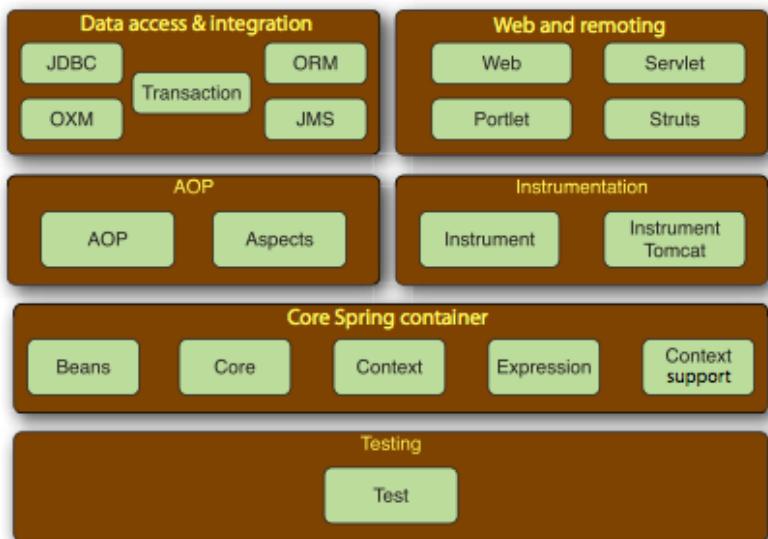
- Utilizzo di Plain Old Java Objects (POJO) che rendono l’attività di sviluppo leggera e minimale
- Basso accoppiamento attraverso un uso intenso di interfacce e iniezione delle dipendenze

- Programmazione dichiarativa attraverso programmazione orientata agli aspetti
- Uso di templates per evitare di riscrivere sempre stesse porzioni di codice.

Spring Framework è composto da diversi moduli. Scaricando e decomprimendo una distribuzione di Spring si trovano 20 file JAR. Ogni JAR appartiene ad una delle sei seguenti categorie:

- Testing;
- Core Spring Container;
- AOP;
- Instrumentation;
- Data Access & Integration;
- Web and Remoting.

La seguente figura schematizza l'architettura fondamentale di Spring.



3.2.1 Core Spring Container

Il “Core Spring Container” è il cuore di Spring. E’ un contenitore che gestisce il ciclo di vita di oggetti di qualsiasi natura che, in Spring,

vengono detti beans. Il suo compito è quindi quello di creare, configurare e gestire tali beans. All'interno di questo modulo si trova la “Spring Bean Factory” che è la porzione di Spring che permette l'iniezione delle dipendenze. Sfruttando il lavoro di questa Factory è possibile effettuare l'iniezione delle dipendenze e la configurazione di Spring in diversi modi. Più in dettaglio **i moduli di Core e Beans** sono responsabili delle funzionalità di *Inversion Of Control (IoC)* e *Dependency Injection*, **Il modulo context estende i servizi basilari del Core** aggiungendo le funzionalità tipiche di un moderno framework. Tra queste troviamo JNDI, EJB, JMX, internazionalizzazione(I18N) e supporto agli eventi. Infine il **modulo Expression** fornisce un potente linguaggio per interrogare e modificare oggetti a runtime.

3.2.2 AOP

Il modulo denominato AOP fornisce il supporto alla programmazione orientata agli aspetti. Come l'iniezione delle dipendenze, l'AOP permette di avere un basso accoppiamento tra gli oggetti che costituiscono l'applicazione. In Spring l'utilizzo di AOP offre il meglio di sé nella **gestione delle transazioni**, permettendo di evitare l'utilizzo degli EJB per tale scopo.

3.2.3 Data Access & Integration

Questo modulo si occupa di due aspetti fondamentali: accesso ai dati e integrazione. Tale porzione dell'ecosistema Spring fornisce un livello di astrazione per l'accesso ai dati mediante tecnologie eterogenee tra loro come ad esempio JDBC, Hibernate o JDO. Questo modulo tende a nascondere la complessità delle API di accesso ai dati, semplificando ed uniformando quelle che sono le problematiche legate alla gestione delle connessioni, delle transazioni e delle

eccezioni. Notevole attenzione è stata posta sull'integrazione del framework con i principali ORM in circolazione compresi JPA, JDO, Hibernate, e iBatis. Il modulo Integration fornisce un'astrazione per usare in modo più efficiente la tecnologia JMS – Java Message Service per l'integrazione asincrona con altre applicazioni. Spring offre anche svariate implementazioni di Object/XML Mapper come JAXB, Castor e XMLBean. Più in dettaglio:

- **JDBC**: e' un modulo che fornisce un livello di astrazione a JDBC che elimina la necessità di implementare a mano tutto il codice necessario a JDBC.
- **ORM**: modulo che fornisce un livello di integrazione con alcune popolari API che implementano il mapping tra oggetti Java e tabelle di un database relazionali. Tali API includono il supporto a JPA, JDO, Hibernat ed iBatis.
- **OXM**: modulo che fornisce un livello di astrazione che supporta il mapping tra oggetti Java e documenti XML, fornisce implementazioni di JAXB, Castor, XMLBeans, JiBX ed XStream.
- **JMS**: modulo che fornisce il Java Messaging Service, contiene funzionalità per produrre e consumare messaggi.
- **Transaction**: modulo che fornisce un supporto per la gestione delle transazioni dichiarative per classi che implementano interfacce speciali e per tutti i POJO.

3.2.4 Web and Remoting

Essendo il Model-View-Controller (MVC) un paradigma comunemente usato per sviluppare applicazioni web, Spring è basato su tale modello. Il mondo Java è ricco di framework MVC. Apache Struts, JSF, WebWork e Tapestry sono i più famosi. Il punto di forza

di Spring è che permette anche l'integrazione con questi framework.

Entrando nel dettaglio:

- **Web**: modulo che fornisce le funzioni di integrazione base orientate al web, come ad esempio la funzione multipart file-upload e l'inizializzazione dello IoC container facente uso di servlet listeners ed un application context orientato al web.
- **Web-Servlet**: modulo che contiene l'implementazione del pattern Model-View-Controller (MVC) per sviluppare Web Application.
- **Web-Struts**: modulo che contiene le classi di supporto per integrare un'applicazione Struts all'interno di un'applicazione Spring.
- **Web-Portlet**: modulo che fornisce un'implementazione MVC da usare nello sviluppo di una web application basata su portlet. Tale modulo ripropone in tale modello di sviluppo le funzionalità del modulo Web-Servlet.

Tale porzione dell'ecosistema Spring, oltre alle funzionalità per costruire applicazioni web, fornisce diverse opzioni per la comunicazione remota come Remote Method Invocation (RMI), Hessian, Burlap e JAX-WS.

3.2.5 Testing

Riconoscendo l'importanza del testing nello sviluppo del software, Spring mette a disposizione un modulo dedicato il cui compito è quello di testare le applicazioni sviluppate con Spring. Questo livello mette a disposizione un ambiente molto potente per il test di componenti Spring, grazie anche alla sua integrazione con JUnit e TestNG e alla presenza di Mock objects per il testing del codice in isolamento.

Quanto appena illustrato costituisce solo una minima parte delle potenzialità offerte da Spring. In realtà esistono altri framework e librerie che fanno parte del mondo di Spring che ne incrementano il suo valore. Di seguito una rapida rassegna di altri moduli che estendono le funzionalità basilari di Spring.

Spring web Flow: sfrutta le funzionalità di Spring MVC per permettere la realizzazione di applicazioni web flow-based

Spring Web Services: permette di sviluppare web services in modo più rapido ed efficiente

Spring Security: utilizzando l'AOP permette di definire, in modo dichiarativo, aspetti relativi alla sicurezza di una applicazione

Spring Integration: offre l'implementazione di diversi patterns di integrazione

Spring Batch: permette di eseguire operazioni di movimentazione massiva di dati in modo automatico, schedulato e fail-safe

Spring Social: permette di sviluppare e integrare applicazioni all'interno di social network quali Facebook e Twitter

Spring Mobile: sostiene e facilita lo sviluppo di applicazioni web mobili

Spring DM - Dynamic Modules: utilizzando Spring DM è possibile costruire applicazioni che sono formate da diversi moduli, debolmente accoppiati e fortemente coesi che espongono e consumano servizi all'interno del framework OSGi

Spring LDAP: fornisce un modello di accesso a LDAP

Spring Rich Client: toolkit che permette di sviluppare applicazioni Swing

Spring.NET: offre le caratteristiche di basso accoppiamento attraverso iniezione delle dipendenze e programmazione orientata agli aspetti sulla piattaforma .NET

Spring-Flex: permette di far interagire applicazioni Flex e AIR con i beans di Spring mediante BlazeDS.

Spring Roo: fornisce degli strumenti interattivi per permettere uno sviluppo rapido di applicazioni Java EE appoggiate allo Spring Framework

Spring Extensions: altre estensioni fornite da sviluppatori di terze parti che offrono nuove funzionalità.

3.3 Iniezione delle dipendenze

Il cuore di Spring è basato sul principio dell'iniezione delle dipendenze (Dependency Injection - DI) o inversione di controllo (Inversion of Control - IoC). Usando l'iniezione delle dipendenze il codice che ne scaturirà sarà semplice, facile da capire e testare. Qualsiasi applicazione non banale è costituita da due o più classi che collaborano l'una con l'altra per realizzare una logica di business. Tradizionalmente ogni oggetto ha il compito di ottenere i riferimenti degli oggetti con cui collabora. Questo comporta un codice fortemente accoppiato e difficile da testare.

Prendiamo in esame la seguente classe

```
public class Cavaliere {  
    private Spada spada;  
    public Cavaliere(){  
        spada = new Spada();  
    }  
}
```

Ogni qualvolta sarà istanziato un Cavaliere questo creerà la sua spada. In questo modo è stata creata una dipendenza tra la classe Cavaliere e Spada. Utilizzando l'iniezione delle dipendenze il codice del precedente esempio può essere così modificato:

```
public class Cavaliere {  
    private Spada spada;  
    public Cavaliere(Spada spada){  
        this.spada = spada;  
    }  
}
```

In questo secondo caso la classe Cavaliere non si deve preoccupare della creazione della Spada. La classe Spada verrà creata da una factory ed implementata in maniera totalmente indipendente e verrà fornita alla classe Cavaliere nel momento in cui essa verrà istanziata. L'intera procedura verrà mediata dal Framework Spring. Nell'esempio appena visto è stato totalmente rimosso il controllo dalla classe Cavaliere ed è stato riposto altrove (ad esempio in un file di configurazione XML, in una classe di configurazione di Spring o mediante annotazioni all'interno del codice). La dipendenza (nell'esempio la classe Spada) sarà creata da una factory (come tutti i bean facenti parte di un'applicazione Spring) e sarà iniettata all'interno della classe che la usa (ovvero Cavaliere) tramite uno strumento chiamato Class Constructor. In questo modo il controllo del flusso è stato invertito dalla Dependency Injection poiché la gestione delle dipendenze è stata delegata ad un sistema esterno. Ora la classe Cavaliere non è accoppiata a nessuna implementazione di Spada. Nel secondo frammento di codice mostrato è molto più facile usare metodi di testing. Come si può capire dall'esempio precedente i vantaggi nell'uso dell'iniezione delle dipendenze sono molteplici e possono essere riassunti così:

- **Codice Collante Ridotto:** la DI permette di scrivere meno codice per

far colloquiare i diversi componenti di una applicazione. In alcuni casi il codice è banale e consiste, come nell'esempio visto, semplicemente nella creazione di una nuova istanza. In altri casi le righe di codice da scrivere possono essere piuttosto complesse. Le maggiori criticità si possono verificare quando si devono effettuare delle ricerche in repository JNDI oppure quando si devono invocare risorse remote. In tali casi l'adozione della DI semplifica il codice dal momento che può fornire un servizio automatico di lookup del JNDI ed il proxy automatico di risorse remote.

- **Configurazione dell'applicazione semplificata:** Adottando l'iniezione delle dipendenze il processo di configurazione di una applicazione è semplificato. E' infatti possibile usare le annotazioni oppure file XML per specificare quali risorse devono essere iniettate in una classe specifica. Inoltre in alcuni casi l'uso della DI permette di cambiare, in modo semplice ed efficace, l'implementazione di una dipendenza. Per esempio si consideri il caso in cui un DAO (Data Access Object) esegue delle operazioni su un database PostgreSQL. Se, per qualche motivo, il database deve cambiare è sufficiente cambiare l'iniezione della dipendenza senza apportare modifiche dirette al codice.
- **Gestire dipendenze comuni in un singolo repository.** Usando la DI tutte le informazioni sulle dipendenze sono localizzate in un unico repository (file XML o apposita classe Java) rendendo il processo sulla gestione delle dipendenze più semplice e meno soggetto ad errori.
- **Attività di testing più efficiente.** Progettando le classi per l'iniezione delle dipendenze è molto facile, come visto, cambiare le dipendenze. Ciò è particolarmente utile nell'attività di test. Si consideri un oggetto

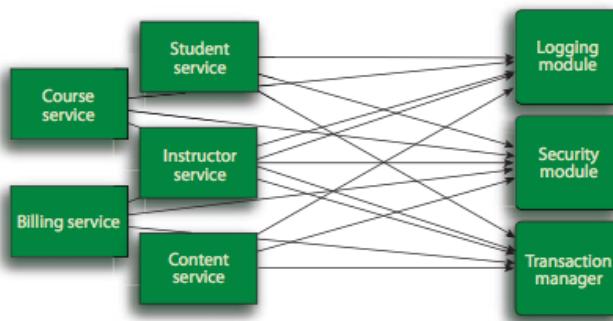
di business che effettua alcune elaborazioni complesse. Per eseguire tali compiti usa un DAO per accedere ai dati memorizzati in un database relazionale. In fase di test non interessa verificare le azioni svolte dal DAO ma è di interesse andare a controllare il comportamento dell'oggetto di business fornendogli diversi insieme di dati. In un approccio tradizionale, dove un oggetto di business è responsabile di ottenere una istanza del DAO, si hanno dei problemi in fase di testing perché non si riesce a sostituire l'implementazione del DAO con un mock che restituisca l'insieme dei dati di test. Usando l'iniezione delle dipendenze è possibile creare un mock del DAO che restituisca l'insieme dei dati di test da passare all'oggetto di business per i controlli del caso. Questo meccanismo può essere esteso per testare ogni livello dell'applicazione e questo è particolarmente utile per verificare i componenti web in cui è possibile creare mock di HttpServletRequest e HttpServletResponse. [SIAW]

3.4 Programmazione Orientata agli Aspetti

Sebbene l'iniezione delle dipendenze permetta di ottenere un basso accoppiamento, la programmazione orientata agli aspetti (AOP) permette di catturare funzionalità che sono utilizzate in tutta l'applicazione in componenti riusabili. L'AOP è definita come una tecnica che favorisce la separazione degli interessi all'interno di un sistema software. Un generico sistema è formato da diversi componenti, ciascuno dei quali è responsabile di una specifica funzionalità. Caratteristiche di un sistema come logging, gestione delle transazioni e sicurezza spesso vengono affidate a componenti le cui responsabilità fondamentali sono altre. Tali caratteristiche, siccome sono dislocate in più componenti, sono chiamate cross-cutting concerns. Allocando tali funzioni su più componenti si introducono due livelli di complessità nel codice:

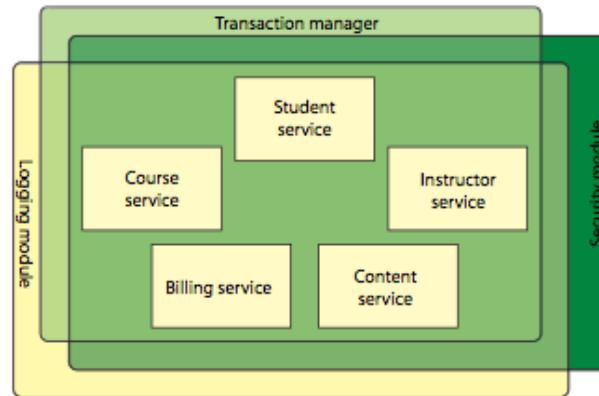
- Il codice che implementa tali funzionalità è sparso su più componenti. Questo significa che se bisogna effettuare un cambiamento in una delle funzionalità è possibile che le modifiche abbiano ripercussioni su più componenti. Anche se una singola funzionalità è stata allocata in un modulo separato resta il problema delle chiamate di metodo che sono duplicate in più punti.
- I componenti sono pieni di codice che non implementa le funzionalità di base per le quali il componente è stato pensato. Un metodo per aggiungere una voce di rubrica dovrebbe preoccuparsi solo di svolgere questo compito e non occuparsi di aspetti come sicurezza o transazionalità.

La seguente figura mostra tale complessità.



Gli oggetti di business sulla sinistra dell'immagine sono troppo legati con i servizi di sistema. Ogni componente, come mostra la figura, deve svolgere i compiti di base per i quali è stato progettato e deve occuparsi del logging, della sicurezza e di essere in un contesto transazionale. AOP consente di modularizzare tali servizi per renderli fruibili, in modo dichiarativo, ai componenti che ne fanno richiesta. Questo

permette pertanto di avere dei componenti più coesi che si occupano solo di svolgere compiti per i quali sono stati progettati.



Come si vede dalla figura soprastante, servizi come la sicurezza, logging e transazioni sono utili a tutti i componenti dell'applicazione e, usando una metafora, possono essere paragonati ad una coperta che deve avvolgere tutta l'applicazione. I componenti interni all'applicazione come Student Service, Course Service si devono occupare solo della logica di business. L'applicazione tuttavia ha bisogno di servizi quali sicurezza, logging e transazioni e tali servizi, grazie all'AOP fornita da Spring, possono essere usati in modo flessibile e trasparente. [SIAW]

3.5 Semplificare l'integrazione con JEE

Negli ultimi anni i framework che fanno uso dell'iniezione delle dipendenze come Spring hanno guadagnato un largo consenso all'interno della comunità degli sviluppatori. Sempre un maggior numero di programmatore sceglie di usare framework come Spring ignorando un approccio basato su EJB. Per facilitare il lavoro degli sviluppatori sono state introdotte le versioni 3.0, 3.1 e 3.2 degli EJB che hanno semplificato l'API della specifica degli Enterprise Java Beans abbracciando inoltre molti dei concetti dell'iniezione delle

dipendenze. Tuttavia, tutte le applicazioni che sono state costruite usando gli EJB o per sistemi realizzati con Spring per i quali si ha la necessità di rilasciarli su container JEE per utilizzare servizi tipici di piattaforme enterprise come JTA Transaction Manager, data source connection pooling, JMS connection factories Spring fornisce un supporto semplificato. Per gli EJB, Spring mette a disposizione un metodo per eseguire il lookup nel registry JNDI ed iniettare il riferimento dell'EJB all'interno dei beans di Spring. È possibile usare Spring per fare l'azione opposta ovvero iniettare all'interno dell'EJB i beans di Spring.

Il “**Java Naming and Directory Interface**” (JNDI) è una API Java che consente di ricercare in una directory degli oggetti in base al nome. JNDI è generalmente usato nelle applicazioni JAVA EE per effettuare operazioni come:

- Transazioni (UserTransaction & TransactionManager)
- Pool di connessioni transazionali ai database (XADataSource)
- Connettersi a JMS e utilizzare queue e/o topic
- Connettersi ad un EJB
- Sicurezza

Per tutte le risorse referenziate da un registry JNDI, Spring elimina la necessità di scrivere del codice complesso per effettuare il lookup. Come conseguenza avremmo che l'applicazione sarà disaccoppiata dal JNDI e questo permette di avere del codice riusabile. Usando l'API convenzionale di JNDI il codice che solitamente è scritto assomiglia al seguente:

```

InitialContext ctx = null;
try{
    ctx = new InitialContext();
    DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/SpitterDatasource");

} catch(NamingException ne){
    //handle naming exception ...
} finally{
    if(ctx != null){
        try{
            ctx.close();
        } catch(NamingException ne) {}
    }
}

```

Dal listato appena mostrato è possibile fare le seguenti osservazioni:

- Per recuperare un DataSource è necessario creare, e poi chiudere, un contesto iniziale (InitialContext). Dal punto di vista del codice non è un'aggiunta di notevoli dimensioni ma tuttavia rappresenta del codice che non è in linea con l'obiettivo di recuperare un date source.
- Bisogna gestire un'eccezione del tipo javax.naming.NamingException.
- Il frammento di codice precedente è strettamente accoppiato con un JNDI Lookup. In generale si dovrebbe recuperare un DataSource cercando di non essere dipendenti da come questo DataSource venga recuperato.
- Il codice dipende anche da uno specifico nome di JNDI (in questo caso java:comp/enc/jdbc/SpitterDatasource – che è quello usato per la ricerca nel registry JNDI). Il nome potrebbe essere esternalizzato in un file ma, in questo modo, nel listato precedente bisognerebbe inserire il codice per eseguire la ricerca nel registry JNDI a partire dalle informazioni presenti nel file di configurazione.

La seguente immagine riassume i punti appena trattati mostrando come un generico elemento DAO sia dipendente ed accoppiato al JNDI.



Per eliminare tutte le problematiche appena descritte è possibile usare Spring ed in particolare il meccanismo dell'iniezione delle dipendenze. Spring mette a disposizione il namespace **jee** dove è definito l'elemento **<jee:jndi-lookup>** che permette di eseguire in modo semplice ed efficace l'uso di un oggetto registrato nel JNDI all'interno di Spring.

Ad esempio per iniettare in Spring un EJB sono sufficienti le seguenti linee di codice all'interno del file di configurazione di Spring.

```

<jee:jndi-lookup id="sqrtBean" jndi-name="SqrtSession"/>

<bean id="sqrtService" class="client.SqrtBean">
    <property name="service" ref="sqrtBean">
        </property>

```

Il codice appena mostrato inietta nel bean client.SqrtBean l'EJB di nome SqrtSession. In questo modo nel bean di Spring chiamato sqrtService non c'è nessuna dipendenza specifica dall'EJB. Se in futuro si dovesse usare un altro bean è sufficiente cambiare l'elemento **<jee:jndi-lookup>**. [JSKM]

3.6 Alternative a Spring

Sul mercato, trovare un framework con le stesse funzionalità di Spring è complesso. Google-Guice è una libreria Java che sfrutta tutti i benefici dell’iniezione delle dipendenze. Per quanto riguarda la parte di sviluppo web i concorrenti a Spring sono molteplici. I più noti sono Struts, Tapestry, JavaServer Faces (JSF), Apache Wicket, VRaptor, SiteMesh, Xwork, WebWork.

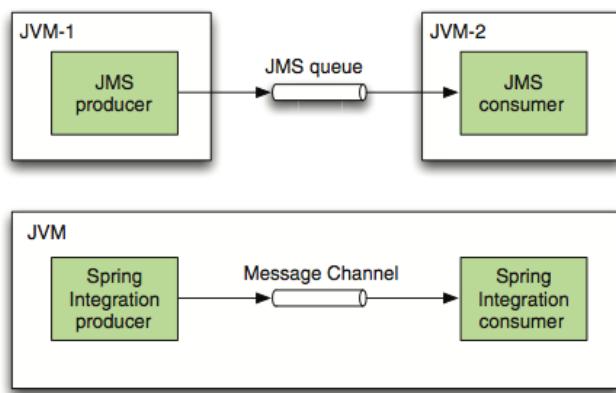
3.7 JMS

Molti sviluppatori Java, quando sentono parlare di “messaging”, la prima cosa che pensano è **Java Message Service (JMS)**. JMS è l’API predominante, appartenente a Java EE, che consente ad applicazioni Java presenti in rete di scambiarsi messaggi tra loro. La specifica JMS è stata progettata per fornire un’astrazione generale sul middleware orientato ai messaggi (MOM). Esistono un gran numero di implementazioni open source della specifica JMS una delle quali è ActiveMQ. Tale prodotto, estremamente facile da configurare, è stato utilizzato negli esempi proposti nella tesi per spiegare come sia possibile fruire del messaging in Spring.

3.7.1 La relazione tra JMS e Spring Integration

Spring permette di usare in modo semplice il paradigma di comunicazione basato sul “Messaging”. Per ottemperare a tale obiettivo viene in ausilio il modulo Spring Integration. Tale parte dell’ecosistema di Spring fornisce un modello coerente per il messaging intraprocesso e interprocesso. Quando si discute sul ruolo di JMS spesso si pensa all’ integrazione di sistemi diversi ma in realtà può essere vantaggioso usare JMS anche all’interno della stessa applicazione per avere benefici quali persistenza, transazioni, bilanciamento di carico e failover. Spring Integration permette di usare facilmente JMS sia per la comunicazione tra processi dislocati

su macchine diverse sia per far colloquiare processi sullo stesso calcolatore. Il ruolo primario di un channel adapter e di un messaging gateways è di connettere una destinazione locale a dei sistemi esterni senza incidere sul codice dei componenti consumatori o produttori. Un altro vantaggio che gli adattatori forniscono è la separazione tra gli aspetti relativi al messaging dai protocolli di trasporto. Quest'ultimi consentono di abilitare un messaging di tipo document-style se la particolare implementazione dell'adattatore invia le richieste su http, interagendo con un filesystem o la mappatura verso un'altra API di messaging. Gli adattatori ed i gateways basati su JMS cadono in quest'ultima categoria e pertanto sono delle ottime scelte quando è richiesta l'integrazione con un sistema esterno. Siccome JMS e Spring seguono lo stesso paradigma di messaging è possibile schematizzare la comunicazione intraprocesso e interprocesso usando un modello simile.



La prima parte della figura mostra la comunicazione interprocesso usando JMS mentre la seconda parte della figura mostra l'integrazione intraprocesso usando Spring Integration. Quale tipo di integrazione è più appropriato dipende dal particolare tipo di architettura dell'applicazione da realizzare. Indipendentemente dal tipo di comunicazione che si vuole utilizzare, Spring Integration fornisce una serie di qualità:

- **Load Balancing:** lo scenario è quello in cui più consumatori sono registrati presso una destinazione condivisa. Il carico può essere distribuito sulla base delle capacità degli stessi consumatori. Ad esempio può accadere che alcuni processi siano in esecuzione su macchine lente o che il processamento di alcuni messaggi necessiti di risorse ingenti e pertanto possono essere gli stessi consumatori a richiedere messaggi quando possono effettivamente elaborarli senza costringere il dispatcher a prendere decisioni.
- **Scalabilità:** I produttori potrebbero inviare tantissimi messaggi ad un unico consumatore. Per evitare un accumulo ed un ritardo nel trattamento dei messaggi, è possibile aggiungere dei processi che consumino tali messaggi nei vincoli di tempo previsti.
- **Disponibilità:** L'intero sistema deve restare operativo anche nel caso di fallimento di uno o più processi consumatori. Nel caso estremo di fallimento di tutti i processi consumatori, il broker deve essere in grado di memorizzare tutti i messaggi fino a quel momento inviati sino al ripristino di uno dei consumatori. Allo stesso modo i processi produttori possono essere allocati o deallocati senza procurare interferenze con i processi consumatori. Tale beneficio è la conseguenza del basso accoppiamento fornito da un sistema di messaggistica. Questa caratteristica può essere usata in sistemi che non devono avere un downtime. Se, per esempio, è necessario aggiornare i processi consumatori questi possono essere rimossi uno alla volta senza inficiare sulle caratteristiche di disponibilità del sistema.
- **Transazionalità:** è possibile fornire un supporto transazionale ai messaggi. Se un componente consumatore tratta con un successo un messaggio può effettuare il commit della transazione altrimenti deve fare il rollback. Quando questa caratteristica è attivata si può

avere bilanciamento di carico ed inoltre, se un processo fallisce, avviene il rollback ed il messaggio può essere inviato ad un altro processo consumatore.

I benefici appena elencati sono ottenuti grazie all'uso combinato di JMS e Spring Integration. Ad esempio Spring Integration 2.1 ha il supporto per RabbitMQ, che implementa il protocollo AMQP.

3.8 Spring e Web Service

Spring Web Service (Spring-WS) è un modulo di Spring che consente di facilitare la creazione di Web Services. In particolar modo tale libreria aiuta nella creazione di Web Services di tipo Soap con approccio “Contract-First”. Usando Spring lo sviluppatore non dovrà codificare esplicitamente il WSDL che sarà invece generato dal framework a partire dall'XML della response e della request definite per il contratto del servizio. In sostanza, creare un web service, richiede la definizione di un file XML che rappresenta il contratto del servizio, la scrittura dei classici file di configurazione di Spring e la codifica delle classi di implementazione dell'endpoint e del servizio.

I vantaggi nell'uso di Spring-WS possono essere così ricapitolati:

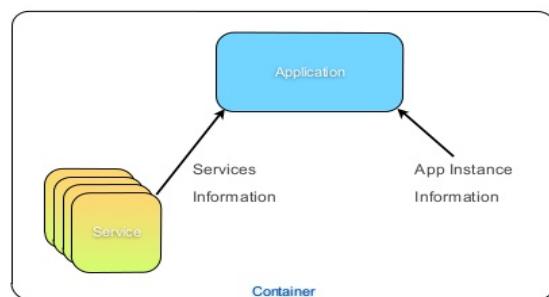
- Velocizza l'utilizzo di Best Practise quali sviluppo mediante un approccio “contract-first”, uso del profilo WS-I, basso accoppiamento tra contratto e implementazione;
- Potente mapping: è possibile distribuire ogni richiesta in entrata a qualsiasi componente in base al payload del messaggio, SOAP Header o espressioni XPath.
- Supporto per le API di XML: i messaggi XML in ingresso possono essere gestiti con JAXP API come DOM, SAX e STAX ma anche con JDOM, dom4j e XOM;

- Marshalling flessibile: è possibile usare JAXB 1 e 2, Castor, XMLBeans, JiBX e XStream;
- Spring-WS usa gli stessi principi alla base del framework Spring pertanto tutte le configurazioni possono essere definite nell'ambito dell'application context e questo riduce il tempo di sviluppo complessivo. L'architettura di Spring-WS è molto simile a quella di Spring-MVC;
- Supporto alla sicurezza: sono previste funzionalità per firmare i messaggi SOAP, cifrarli, decifrarli e autenticarli.

Per il rilascio del web service è sufficiente un lightweight container come Tomcat o Jetty.

3.9 Spring Cloud

Spring offre molti benefici anche in ambito Cloud. Una delle caratteristiche di Spring è la portabilità e negli scenari attuali tale qualità è sempre più importante. Per sfruttare i benefici del Cloud, all'interno dell'ecosistema Spring, è presente il modulo Spring Cloud che permette di creare rapidamente applicazioni per il Cloud che sfruttano alcuni pattern comuni nei sistemi distribuiti come gestione delle configurazioni, sessioni distribuite, stato del cluster, leadership election e service discovery. Tali applicazioni possono connettersi a vari servizi di cloud scoprendo tutte le informazioni necessarie a runtime



Spring funziona in modo affidabile e coerente in tutte le PaaS (Platform as a service) dei più importanti vendor come Cloud Foundry, Beanstalk di Amazon Web Services, e Google App Engine. Una tipica applicazione fatta con Spring non richiederà modifiche per essere rilasciata, ad esempio, su Cloud Foundry e pertanto potrà usare servizi come PostgreSQL, MySQL, MongoDB, Redis e RabbitMQ. La maggior parte delle offerte PaaS variano nelle tecnologie supportate ma un lightweight application server come Tomcat o Jetty è sempre presente. Sviluppando applicazioni con Spring è pertanto possibile sfruttare la modularità e l'agilità che le offerte PaaS forniscono. Due sono i concetti introdotti per permettere tale flessibilità: Cloud Connector e Service Connector. Un Cloud Connector è un'interfaccia che un fornitore di cloud implementa per permettere al resto della libreria di lavorare con la piattaforma di cloud. Un Service Connector è un oggetto che rappresenta la connessione a un servizio. E' possibile definire Cloud Connector e Service Connector personalizzati per supportare altre piattaforme e servizi cloud, rilasciarli come file Jar ed aggiungerli al classpath.

3.10 Spring Vs JEE

Come ampiamente detto, Spring Framework è stato introdotto con l'obiettivo di essere una semplice ed efficace alternativa agli EJB. Da quando Spring è stato proposto alla comunità degli sviluppatori sia il framework che gli stessi EJB si sono evoluti ed influenzati l'uno con l'altro. Nel Maggio 2013 gli EJB hanno raggiunto la versione 3.2 che associa alla semplicità di utilizzo un ricco set di funzionalità. Spring è alla versione 4.1.2 e da quando è uscito per la prima volta tantissime sono state le funzionalità che gli ideatori hanno aggiunto. Sebbene

attualmente realizzare applicazioni con Spring o JEE comporti un tempo di sviluppo più limitato rispetto al passato e numerosi vantaggi, sul web esistono infinite diatribe tra sostenitori di Spring e di JEE. Per decidere quale stack tecnologico usare andrebbero presi in considerazione i seguenti punti:

- Requisiti non funzionali;
- Funzionalità offerte in accordo con le necessità del cliente;
- Competenze del team;
- Conseguenze che può provare la scelta di una determinata tecnologia rispetto ad un'altra.

Come si evince da discussioni e confronti presenti sul web i punti appena elencati, nella scelta tra un approccio basato su Spring o JEE, non vengono minimamente presi in considerazione. A tal proposito è utile effettuare un confronto tra Spring e JEE per analizzare eventuali punti di forza o criticità delle due tecnologie.

3.10.1 Comparazione

Feature	EJB 3.x	Spring Framework
Distributed Computing	<ul style="list-style-type: none"> • Permettono l'invocazione remota con RMI. Possibilità di accedere ai beans attraverso @Remote interfaces • Possibilità di effettuare un lookup remoto su JNDI • Accesso attraverso web services: JAX-WS, JAX-RS (non fanno parte degli EJB, ma EJBs possono essere acceduti usando tali servizi) 	<ul style="list-style-type: none"> • Comunicazione remota con RMI, JAX-RPC, JAX-WS, JAX-RS • moduli: spring-ws, spring-webmvc
Beans	<ul style="list-style-type: none"> • @Stateless – fornisce logica di business • @Singleton – ne esiste solo uno per ogni applicazione • @Statefull – fornisce logica di business e può mantenere traccia sullo stato del client • @MessageDriven – permette di gestire messaggi 	<ul style="list-style-type: none"> • Singleton – solo uno per IoC container; creato all'avvio dell'applicazione • Prototype – creato ogni volta che si effettua una iniezione • Request – creato per una singola richiesta • Session – mantiene lo stato di una sessione http con un client • Global Session – mantiene lo stato in una sessione http globale
Gestione delle Dipendenze	<ul style="list-style-type: none"> • JNDI registry permette di ottenere beans usando interfacce di tipo local, remote o no-interface views • @EJB annotation permettono una iniezione delle dipendenze automatica • @Resource permette di iniettare risorse come EJBContext, TimerService, SecurityContext • possibilità di gestire dipendenze attraverso descrittori di deployment 	<ul style="list-style-type: none"> • permette l'iniezione attraverso metodi setter, costruttori oppure mediante annotazioni del tipo @Autowired, @Resource, @Inject • iniezione attraverso file di configurazione (chiamati application context descriptor) • permette il lookup in JNDI registry
Gestione della Concorrenza	<ul style="list-style-type: none"> • per i beans di sessione stateless e stateful session il container gestisce la concorrenza • il container gestisce un pool di beans accessibili • anche per bean singleton (dalla versione 3.1) è stato introdotto il supporto alla concorrenza • invocazione asincrona usando @Asynchronous 	<ul style="list-style-type: none"> • tutti i beans sono singleton, beans con compiti di sessione sono creati per ogni client • la concorrenza è gestita dai bean • invocazione asincrona usando @Async

Gestione delle Transazioni	<ul style="list-style-type: none"> il container gestisce le transazioni usando JTA la gestione delle transazioni è permessa da container attraverso annotazioni o file di configurazione nei bean @Statefull la gestione delle transazioni non deve essere esplicita 	<ul style="list-style-type: none"> gestisce le transazioni attraverso differenti API come JTA, JDBC, Hibernate, JPA, JDO Supporto per una gestione dichiarativa delle transazioni
Messaging	<ul style="list-style-type: none"> message driven come ascoltatori di messaggi supporto per code (point2point) e topic (publish-subscribe) concorrenza gestita dal container 	<ul style="list-style-type: none"> con spring-jms è possibile creare dei consumatori e produttori di messaggi richieste un MQ esterno come ActiveMQ
Aspect Oriented Programming	<ul style="list-style-type: none"> Interceptors 1.1 (since 3.1) 	<ul style="list-style-type: none"> Modulo spring-aop e AspectJ, CGLIB
Scheduling	<ul style="list-style-type: none"> scheduling jobs usando TimerService @Scheduled 	<ul style="list-style-type: none"> Spring TaskScheduler Quartz può essere aggiunto a Spring
Security Management	<ul style="list-style-type: none"> Integrated support for declarative and programmatic security through JAAS 	<ul style="list-style-type: none"> modulo spring-security ricco di funzionalità relative alla sicurezza
Integration Testing	<ul style="list-style-type: none"> integration testing con Arquillian integration testing con embedded container 	<ul style="list-style-type: none"> modulo dedicato al testing ricco di funzionalità
Deployment	<ul style="list-style-type: none"> enterprise archive *.ear –può consistere di molti moduli web ed ejb singolo modulo web *.war configurazione in application.xml (*.ear) e ejb-jar.xml (*.jar, *.war) richiede application server per EJB come JBoss, WebLogic, Apache Tomee/+ 	<ul style="list-style-type: none"> packaged in *.jar or *.war files configurazione in *.xml files oppure mediante annotazioni rilascio in lightweight containers come Tomcat, Jetty

3.10.2 Cosa usare?

Capire quando è meglio usare Spring o JEE è cosa assai ardua specialmente oggigiorno. Entrambe le soluzioni forniscono mezzi per raggiungere gli stessi obiettivi pertanto per decretare quale tecnologia usare nella realizzazione di un sistema i veri punti da prendere in considerazione sono i seguenti:

- environment (standalone o distribuito);
- esigenze funzionali del cliente;
- qualità non funzionali (scalabilità, fail-over);
- conoscenza generale del modello di programmazione per entrambe le soluzioni;
- competenze del team di sviluppo.

Adottare un approccio basato sugli EJB attualmente non è più complesso come poteva essere agli albori di tale tecnologia. Realizzare un'applicazione che richiede diversi moduli di Spring può portare ad avere un risultato finale che è formato da diverse centinaia di megabyte di codice.

CAPITOLO 4: GESTIONE DEI CAMBIAMENTI, IL CONTROLLO DI VERSIONE

Lo scopo di questo capitolo è quello di motivare l'uso di software per il controllo di versione, descrivendo le modalità di applicazione.

4.1 Problematica

Quando si realizza del software, vi sono sempre dei cambiamenti e poiché tali cambiamenti si verificano, occorre gestirli in maniera efficace. La *gestione dei cambiamenti* (change management), talvolta detta anche configurazione del software (software configuration management), è un insieme di attività progettate con lo scopo di tenere sotto controllo i cambiamenti tramite l'identificazione dei prodotti che possono cambiare, la definizione delle relazioni fra di essi, la definizione dei meccanismi per la gestione delle differenti versioni di questi prodotti, il controllo delle modifiche imposte e producendo versioni e report sulle modifiche eseguite. All'interno di un corso universitario di Architetture Software, in cui gli studenti sono chiamati a realizzare progetti che utilizzano tecnologie di frontiera e dove spesso tali tecnologie devono essere integrate le une con le altre, è possibile pensare che lo sviluppo di un elaborato passi attraverso diverse versioni. Spesso l'attività di progettazione non è eseguita da un singolo studente ma, per iniziare ad abituare il discente ad un contesto industriale, in team. Per tali ragioni è bene indurre gli studenti ad utilizzare un sistema per il controllo delle versioni in modo che la loro attenzione sia rivolta maggiormente sull'aspetto sperimentale piuttosto che su quello organizzativo. Come strumento viene proposto GitHub, popolare piattaforma di social-coding.

4.2 Gestione dei cambiamenti

I cambiamenti sono inevitabili nella realizzazione di software ed accrescono la confusione in chi prende parte ad un progetto. La confusione nasce quando i cambiamenti non vengono analizzati preliminarmente, registrati prima di essere effettuati, riferiti a tutti gli interessati e controllati in modo da migliorare la qualità e ridurre gli errori. A questo proposito W.A. **Babich** nel testo “*Software Configuration Management*” edito da Addison-Wesley ha affermato

*“L’arte di coordinare lo sviluppo del software al fine di minimizzare [...] la confusione si dice **gestione delle configurazioni**. La gestione delle configurazioni è l’arte di individuare, organizzare e controllare le modifiche al software in corso di realizzazione da parte di un team. Lo scopo è quello di massimizzare la produttività minimizzando gli errori.”*

La gestione delle configurazioni software (SCM, Software Configuration Management o CM, Change Management) è un’attività ombrello, che copre tutto il processo software. Un cambiamento, in generale, può avvenire in qualsiasi momento, quindi si sviluppano attività di SCM per

- riconoscere i cambiamenti;
- controllarli;
- garantire che siano opportunamente implementati;
- riferire a tutti gli interessati l’avvenuto cambiamento.

Uno degli obiettivi principali dell’ingegneria del software è quello di accrescere la facilità con cui trattare i cambiamenti e ridurre il carico di lavoro necessario in occasione dei cambiamenti stessi. [SCMB]

4.3 Importanza della gestione dei cambiamenti

“*Se non controllate i cambiamenti, saranno i cambiamenti a controllare voi.*” Tale frase, molto utilizzata nei libri di testo che trattano di ingegneria del software, sintetizza con estrema efficacia quale sia l’importanza della gestione dei cambiamenti. E’ molto facile che un insieme di modifiche incontrollate trasformi un progetto software ben realizzato in un caos. Per questo motivo, un appropriato sistema per la gestione delle configurazioni del software è una parte essenziale delle tecniche di buona gestione dei progetti e di solida ingegnerizzazione del software.

4.4 Gestione delle configurazioni software

Il risultato del processo software è una serie di informazioni che si possono suddividere in tre grandi categorie:

- Programmi (nelle forme sorgente ed eseguibile)
- Elaborati che descrivono i programmi (sia per i tecnici, sia per gli utenti)
- Strutture dati (contenute nei programmi o esterne).

Gli elementi che compongono l’informazione prodotta in ogni sua parte dal processo software sono globalmente chiamati *configurazione software*. Se ciascun elemento della configurazione si limitasse a generare altri elementi, non si creerebbe molta confusione. Ma, sfortunatamente, nel processo entra in gioco un’altra variabile: le modifiche.

La **gestione delle configurazioni** software è un insieme di attività ideate per gestire le modifiche per tutto il ciclo di vita del software; la si può anche considerare una attività di gestione della qualità del software, applicata in tutte le fasi del processo software. [PISP]

4.5 Il processo di gestione delle configurazioni

Il processo di gestione della configurazione del software definisce una serie di task che hanno quattro obiettivi principali:

- identificare tutti gli elementi che collettivamente definiscono la configurazione del software;
- gestire le modifiche ad uno o più di questi elementi;
- facilitare la costruzione di versioni differenti di un'applicazione
- garantire che venga mantenuta la qualità del software a mano a mano che la configurazione evolve nel corso del tempo.

All'interno di tale processo, un ruolo importante è ricoperto dai meccanismi di controllo delle versioni.

4.6 Controllo delle versioni

Il controllo delle versioni combina procedure e strumenti al fine di gestire versioni diverse degli oggetti della configurazione, prodotti nel corso del processo software.

Un sistema di controllo delle versioni implementa o viene integrato direttamente con quattro funzionalità principali:

- un database del progetto (repository) che memorizza tutti gli oggetti rilevanti della configurazione;
- una *funzionalità di gestione della versione* che memorizza tutte le versioni di un oggetto di configurazione (o che consente di costruire una qualsiasi versione utilizzando le differenze rispetto alle versioni precedenti);
- una *funzionalità make* che consente di raccogliere tutti gli oggetti di configurazione rilevanti e costruire una specifica versione del software.

Inoltre, i sistemi di controllo delle versioni e delle modifiche spesso implementano un monitoraggio dei problemi (chiamato anche monitoraggio dei bug) che consente al team di registrare e monitorare lo stato di tutti i problemi associati a ciascun oggetto della configurazione.

Molti sistemi di controllo della versione definiscono il concetto d'*insieme di modifiche* (*change set*), una raccolta di tutte le modifiche (rispetto ad una qualche configurazione stabilizzata) necessarie per creare una specifica versione del software.

Per un'applicazione od un sistema possono essere identificati vari insiemi di modifiche. Questo consente di costruire una versione del software specificando gli insiemi di modifiche (usandone il nome) che devono essere applicate alla configurazione stabilizzata. Per ottenere ciò è applicato un approccio a modellazione del sistema. Il modello del sistema contiene:

- uno schema (template) che include una gerarchia di componenti ed un “ordine di costruzione” che descrive il modo in cui il sistema deve essere costruito
- le regole di costruzione
- le regole di verifica (spesso è possibile interrogare il modello del sistema per valutare quale impatto avrà un cambiamento in un componente sugli altri componenti)

Negli ultimi venti anni sono stati proposti diversi strumenti automatici di supporto al controllo di versione. La differenza fra le soluzioni sta soprattutto nel grado di sofisticazione degli attributi utilizzati per costruire le diverse varianti e versioni di un sistema e nel meccanismo del processo di costruzione. [PISP]

4.7 GitHub



GitHub è un social-code, uno speciale social network dedicato ai programmatori. La piattaforma social dedicata alla programmazione è intersecata al controllo di versione dei file chiamato Git. Per capire cosa sia GitHub e come funziona, è necessario comprendere innanzitutto cos'è Git, cuore pulsante della piattaforma web. Creato dal Linus Torvalds, Git è un **software di controllo di versione**: ciò vuol dire che controlla e gestisce gli aggiornamenti di un progetto senza sovrascrivere nessuna parte del progetto stesso. Venne creato da Torvalds e dai suoi collaboratori nel corso dello sviluppo del kernel Linux: nel caso in cui qualche aggiornamento non avesse dato gli effetti sperati, si poteva sempre tornare indietro e recuperare la versione funzionante senza troppi problemi. In GitHub questi concetti e queste pratiche sono state portate all'estremo, applicandole su un campione molto più ampio e in ambiti più disparati. Grazie a Git, gli utenti della piattaforma social creata da Tom Preston-Werner, Chris Wanstrath e PJ Hyett possono lavorare contemporaneamente sulla medesima versione dello stesso progetto senza timore di apportare modifiche sostanziali. Tutte le vecchie versioni saranno conservate nel proprio *repository*, così da poterle recuperare in caso di necessità. Inoltre, per ogni utente al lavoro verrà creata una differente versione del progetto, così da non creare fastidiose sovrapposizioni o sovrascritture.

Grazie a Github è inoltre possibile gestire i progetti online, mantenere una copia sul server e visionarla online senza scaricarla in ssh. In dettaglio le caratteristiche di github sono:

- Mandare messaggi privati;
- Sistema follow/unfollow dei utenti (stile twitter) e dei progetti;
- Possibilità di gestire una wiki (per ogni repository);
- Possibilità di gestire un sistema di issue (per ogni repository);
- Possibilità di usare la maggior parte dei comandi del protocollo git tramite un pannello;

GitHub facilita pertanto la comunicazione tra sviluppatori di uno stesso team. Facilità di utilizzo e apprendimento immediato, nonché grande disponibilità di documentazione e tutorial, sono altre caratteristiche fondamentali di tale piattaforma che ne hanno consentito un grande successo. Per tali motivi, GitHub, ben si presta ad essere usato per la gestione del controllo delle versioni nell'ambito di progetti di un corso universitario quale Architetture Software.

4.8 Cooperative Learning

L’idea di suddividere gli studenti in piccoli team di sviluppo, e l’utilizzo di strumenti quali GitHub che facilitano la comunicazione e la condivisione di codice tra membri del gruppo, ha ripercussioni positive sull’apprendimento. Per spiegare brevemente tali vantaggi è possibile fare riferimento al “*Cooperative Learning*”.

Con **Cooperative Learning** (CL) si intende un insieme di principi, tecniche e metodi di conduzione della classe, in base ai quali gli studenti affrontano l’apprendimento delle discipline curriculari lavorando in piccoli gruppi in modo interattivo, responsabile, collaborativo, solidale e ricevendo valutazioni sulla base dei risultati ottenuti.

L'idea di lavorare insieme non è certo nuova nella storia dell'umanità, ma mai come in questo periodo l'arte della collaborazione appare indispensabile e difficile.

Lev Semenovic Vygotskij (1896 - 1934), brillante psicologo sovietico che si è particolarmente impegnato nell'ambito educativo, ha sottolineato come un apprendimento basato su collaborazioni in gruppo abbia ripercussioni positive sull'alunno. Per Vygotskij le conoscenze di un umano sono divise in due parti:

- **zona di sviluppo attuale**, che riguarda le abilità già acquisite
- **zona di sviluppo potenziale**, dove si trovano le abilità non ancora presenti nella persona perché con ogni probabilità verranno acquisite in futuro.

Mediante un approccio di apprendimento basato sul lavoro di gruppo, grazie alla condivisione di esperienze, alla diversificazione delle conoscenze tipiche di un team, alla responsabilità personale di ogni membro del gruppo, un alunno è stimolato a svolgere al meglio il proprio lavoro. Se qualche conoscenza richiesta per portare a termine un task del lavoro assegnato al gruppo non è presente in modo maturo all'interno della zona di sviluppo attuale di un discente, grazie alla collaborazione, lo studente in questione può arrivare ad avere una conoscenza migliore e piena di tale competenza.

Tale frase, in estrema sintesi, riassume il pensiero di Vygotskij.

“Ciò che l'alunno riesce a fare in cooperazione oggi, potrà farlo da solo domani. Pertanto, l'unica buona forma di istruzione è quella che anticipa lo sviluppo e lo conduce; essa non dovrebbe essere indirizzata tanto alle forme mature quanto a quelle che stanno maturando.”

CAPITOLO 5: LO STUDIO DI CASO

Lo scopo di questo capitolo è quello di illustrare lo studio di caso utilizzato per la sperimentazione. L'applicazione realizzata è basata su uno scenario realmente esistente. Dopo aver introdotto le caratteristiche generali dello studio di caso verranno descritte le specifiche, i requisiti funzionali e di qualità, le modalità di accesso al sistema.

5.1 Premessa

Per ottemperare agli obiettivi della tesi è stato scelto un esempio pratico di sistema e realmente esistente. E' stato individuato un dominio che potesse essere scomposto in sotto-sistemi ognuno dei quali ha un proprio insieme di requisiti, scopi e attori condividendo, tuttavia, un insieme di operazioni comuni. L'esempio applicativo è stato scelto in modo che possa essere facilmente compreso e gestito per consentire di focalizzare l'attenzione sugli aspetti architetturali e tecnologici.

5.2 Applicazione di car-sharing

5.2.1 Generalità dell'applicazione

L'applicazione di riferimento prende spunto da servizi di car-sharing come Car2Go (<https://www.car2go.com>) ed Enjoy (<http://enjoy.eni.com>) popolari servizi di car sharing a flotta libera.

Il **car sharing** è un servizio che permette di utilizzare un'automobile su prenotazione, prelevandola e riportandola in un parcheggio, e pagando in ragione dell'utilizzo fatto. Questo servizio è utilizzato all'interno di politiche di mobilità sostenibile, per favorire il passaggio dal possesso

del mezzo all'uso dello stesso (cioè all'accesso al servizio di mobilità), in modo da consentire di rinunciare all'automobile privata ma non alla flessibilità delle proprie esigenze di mobilità. L'auto, in questo modo, passa dall'ambito dei beni di consumo a quello dei servizi. La novità dei servizi di car sharing attuali è stata nell'introduzione del free floating, ossia nel fatto che non ci fossero punti di sosta convenzionati: le auto si parcheggiano per la strada, esattamente come si fa con la propria vettura. Quando un abbonato ne ha bisogno, cerca la vettura più vicina alla posizione in cui si trova, consultando il sito internet, utilizzando l'app, oppure chiamando il call center. C'è infine la possibilità di salire direttamente su un'auto parcheggiata per strada, controllando che non sia già prenotata (appare la dicitura sul parabrezza). Raggiunta l'auto designata il noleggiatore dovrà, mediante il computer di bordo della macchina, inserire alcuni dati sullo stato attuale della vettura. Tali informazioni riguardano lo stato di pulizia interna dell'abitacolo nonché la presenza di anomalie della carrozzeria della macchina. Alla fine dell'utilizzo si parcheggia, anche sulle aree di sosta riservate ai residenti. L'utente paga un abbonamento, e poi una cifra al chilometro comprensiva di carburante.

5.2.2 Lo Scenario

L'applicazione realizzata come lavoro di questa tesi ha cercato di ripercorrere il più fedelmente possibile quanto avviene nei reali servizi di car-sharing. Per evidenziare alcune peculiarità delle tecnologie utilizzate, alcuni aspetti delle reali applicazioni per la gestione del noleggio delle auto sono stati enfatizzati mentre altri sono stati rilassati.

Un **utente**, mediante un sito web, ha la possibilità di vedere all'interno del Grande Raccordo Anulare di Roma, quali sono le vetture che può noleggiare. Di ogni vettura l'utente può osservarne la posizione sulla mappa geografica ed avere alcune indicazioni sullo stato della macchina stessa. Tali informazioni sono inerenti alla benzina attualmente disponibile nel serbatoio nonché allo stato di pulizia interno della macchina e alle condizioni esterne della carrozzeria.

Un utente, se interessato a noleggiare un'auto, può registrarsi e quindi autenticarsi al sistema. Una volta effettuato il login, un utente può decidere di effettuare la prenotazione di un'auto. In tal modo la vettura risulterà prenotata e nessun altro utente può noleggiarla. L'utente che ha effettuato la prenotazione la può disdire, rendendo l'auto nuovamente libera, oppure la può confermare decidendo così di iniziare un viaggio. L'utente prima di mettersi alla guida, deve compilare un form che riassume lo stato delle condizioni esterne ed interne dell'auto con eventuali note aggiuntive. Settate tali informazioni è possibile iniziare una guida. Il sistema realizzato, in realtà, effettuerà una simulazione di guida assegnando una nuova posizione dell'auto all'interno di Roma e calcolando distanza e tempi relativi allo spostamento. Tali dati sono stati ottenuti in diversi modi, quello più interessante fa uso dei WebService messi a disposizione da Google Maps per calcolare percorso e tempo di percorrenza in base alle condizioni attuali del traffico. Un utente, inoltre, può anche visionare lo storico dei noleggi effettuati ed avere un riepilogo dei costi.

Un **amministratore** di sistema ha il duplice compito di gestire le utenze ed il parco auto. Per gestione di utenze s'indica la possibilità di assegnare il ruolo di amministratore ad un utente registrato all'applicazione. L'amministratore ha, come anticipato, il compito di

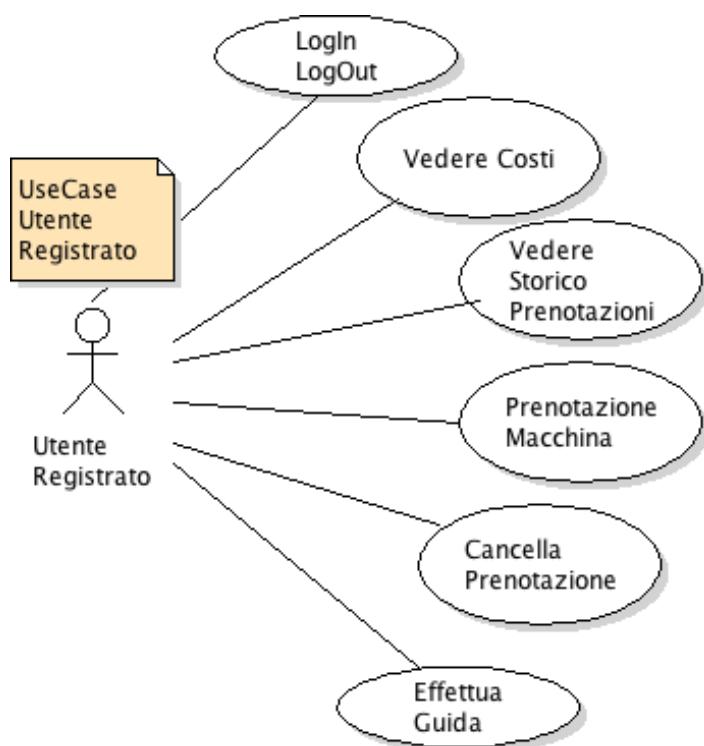
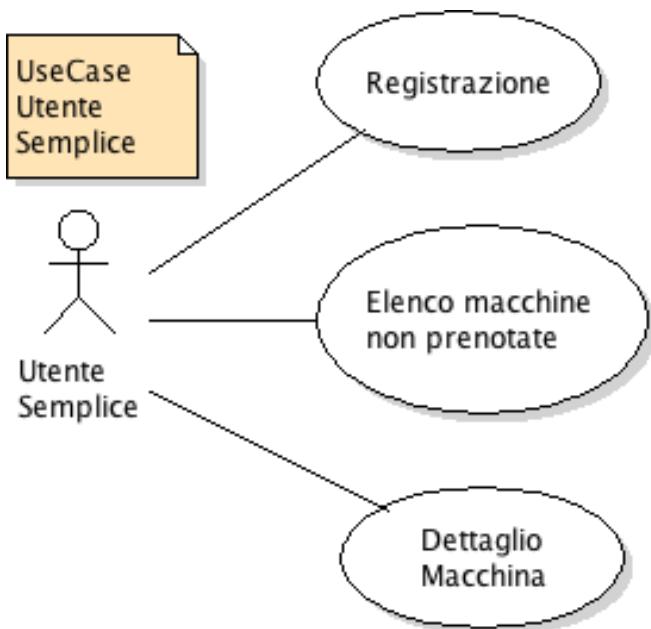
dover gestire i mezzi ossia creare una nuova vettura, effettuare i rifornimenti e far pulire o riparare le auto.

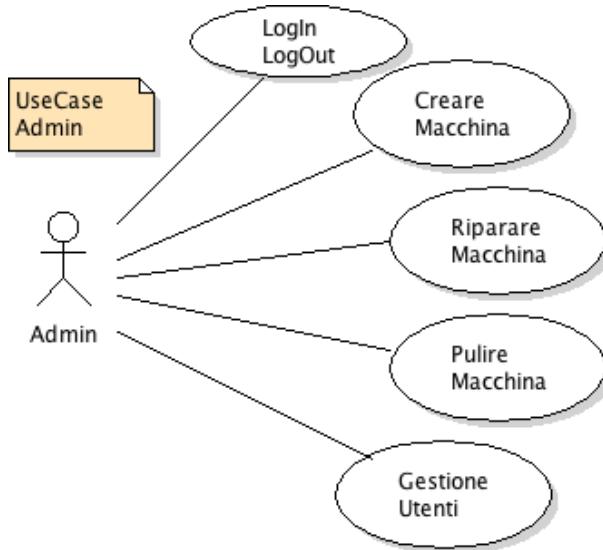
Una **vettura**, come si evince da quanto scritto precedentemente, può essere noleggiata ed ha associate delle informazioni che riguardano il suo stato interno/esterno e quantità di carburante. Di una vettura è di interesse sapere la sua posizione attuale in termine di indirizzo e coordinate quali latitudine e longitudine. Una vettura può anche inviare all'amministratore di sistema l'eventuale tipologia di guasto che l'ha colpita.

Una **prenotazione** riassume dei dati sul noleggio effettuato da un utente nei confronti di una vettura. Per tali motivi è necessario tenere traccia del quantitativo di chilometri percorsi e del tempo necessario a coprire tale distanza nonché della data in cui è stato effettuato il tragitto. Tali dati saranno necessari per effettuare la tariffazione.

5.2.3 Requisiti funzionali

Le funzionalità dell'applicazione di car-sharing possono essere così riassunte:





- **Gestione delle utenze:** Il sistema deve permettere la registrazione di utenze e meccanismi di autorizzazione e autenticazione. Sono consentiti tre tipi di utente: utente non registrato, utente standard registrato e amministratore. L'utente standard può interagire con il sistema noleggiando le auto e controllare lo storico dei noleggi e dei relativi costi. L'amministratore può creare auto e gestire il sistema attraverso l'invocazione di comandi per riparare, pulire e rifornire di carburante le auto. Per una migliore gestione delle utenze è stato utilizzato un Roled Based Access Control
- **Gestione dei noleggi:** l'utente standard registrato al sistema può prenotare e noleggiare le auto dopo essersi autenticato al sistema. Di ogni noleggio sarà d'interesse l'ora in cui è avvenuto, la distanza percorsa e la durata dello stesso. Queste informazioni saranno utili per conteggiare i costi.
- **Gestione del parco auto:** ogni automobile della flotta ha la costante necessità di avere carburante e non problematiche interne o esterne. Per problematiche interne si intende una poca pulizia

dell’abitacolo mentre le problematiche esterne riguardano la presenza di eventuali danneggiamenti della carrozzeria o eventuali guasti. Se il carburante scende sotto una soglia stabilita oppure se la vettura presenta problematiche interne/esterne sarà compito del sistema effettuare le notifiche opportune. In caso di problematica interna, la vettura comunicherà il tipo di guasto riscontrato. E’ di interesse per il sistema tenere traccia delle prenotazioni effettuate dagli utenti nei confronti di una vettura. Se una vettura è prenotata, non può essere richiesta da un altro utente.

5.2.4 Gestione dell’accesso ai client

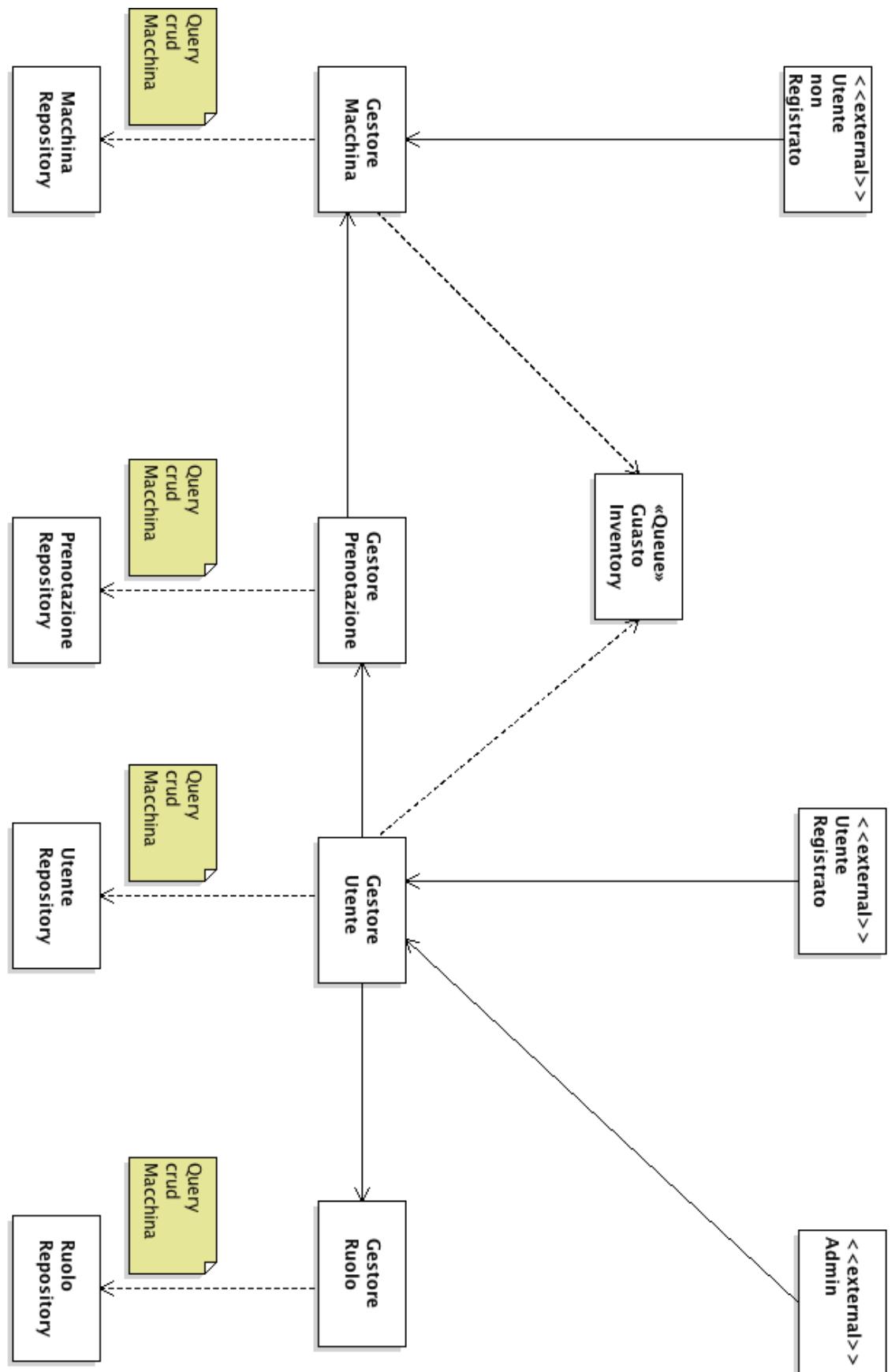
Il sistema deve essere accessibile tramite interfaccia web: a tale scopo è stata prevista la realizzazione di una web application che permetta l’accesso alle funzionalità del sistema. Tali funzionalità sono state successivamente realizzate utilizzando tecnologie diverse. Per usufruire di tali tecnologie un utente utilizza sempre le medesime pagine web. Attraverso una architettura idonea, le varie tecnologie con le quali sono state implementate le funzionalità, sono del tutto trasparenti rispetto alla logica di presentazione e quindi all’utente finale. E’ stato realizzato un client Swing per richiedere dei servizi esposti come Web Server Rest.

5.2.5 Vista Funzionale

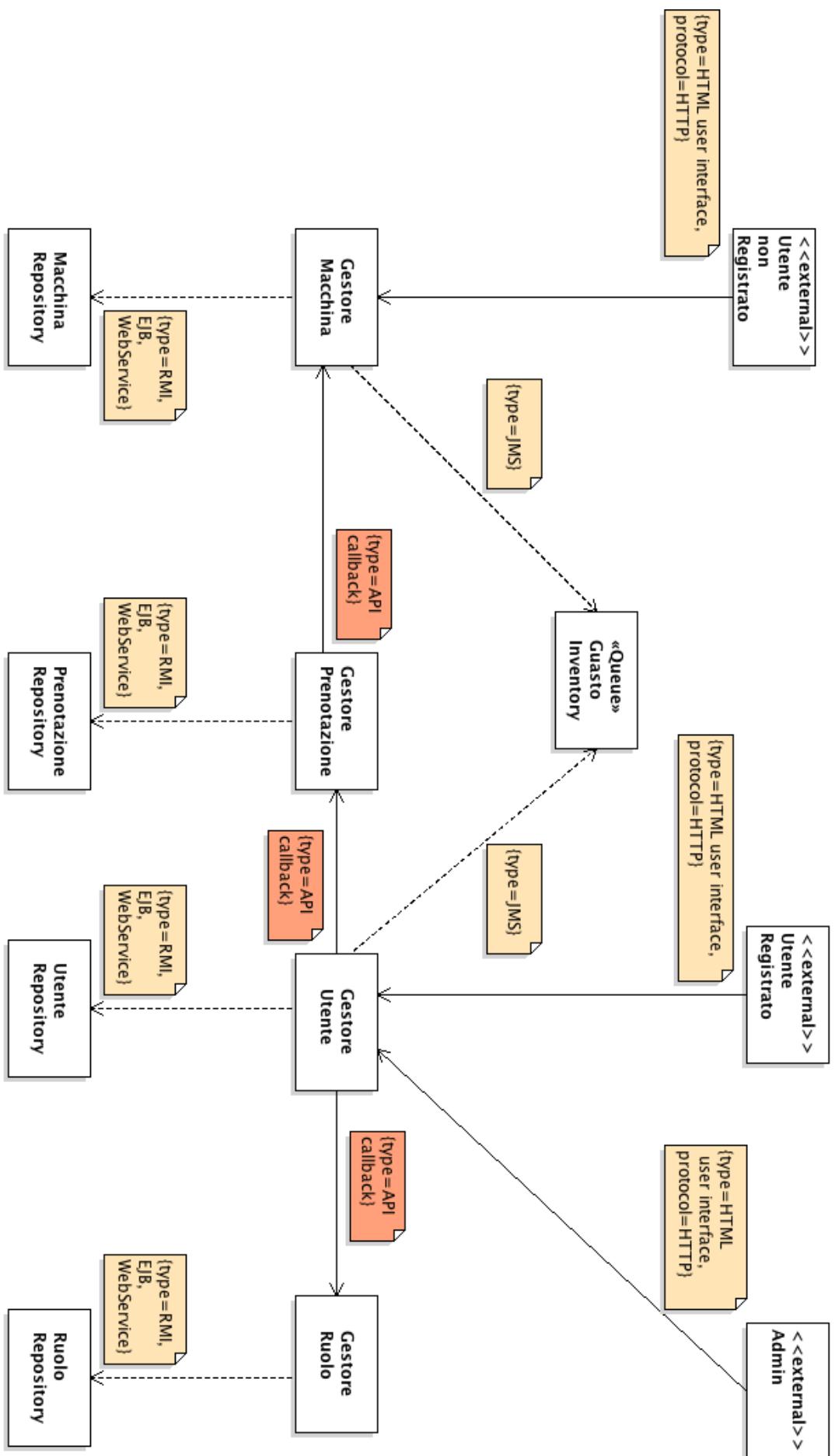
L'immagine seguente mostra una vista funzionale dell'applicazione in cui sono mostrati:

- I tre attori esterni che interagiscono con l'applicazione:
 - Utente non registrato
 - Utente registrato
 - Admin
- Quattro componenti funzionali principali
 - Gestore Macchina
 - Gestore Utente
 - Gestore Ruolo
 - Gestore Prenotazione
- Cinque repository
 - Macchina Inventory
 - Utente Inventory
 - Ruolo Inventory
 - Prenotazione Inventory
 - Guasto Inventory
- Il tipo di comunicazione tra i componenti

Applicazione di Car-Sharing – Vista Funzionale



Applicazione di Car-Sharing – Vista Funzionale



Dell'applicazione sono state realizzate diverse versioni. Più in dettaglio sono state create versioni che implementano tutte le funzionalità mediante RMI, EJB, Web Service e come semplice applicazione Web. È stata anche realizzata una versione che ha unito tutte le tecnologie affrontate nel corso di Architetture Software. In tale versione, quindi, le funzionalità complessive sono state realizzate mediante l'uso congiunto di tutte le tecnologie studiate nel corso pensando ad una possibile assegnazione come homework. Più dettaglio, mediante RMI, è stata implementata la parte esterna dell'applicazione, ossia quella che può vedere un utente che non è registrato al sistema. Tale porzione dell'applicazione comprende funzionalità quali: elenco delle macchine non prenotate, dettaglio di una macchina, registrazione al sistema e login. Una volta che l'utente ha effettuato il login, a seconda delle credenziali immesse, può accedere alla pagina principale dell'utente registrato oppure a quella dell'admin. Seguendo tale percorso, le funzionalità relative all'utente registrato e quelle relative all'admin sono state realizzate secondo tecnologie differenti. Funzioni come la prenotazione di un'auto, riepilogo di costi e storico dei noleggi sono state realizzate mediante EJB. Le funzionalità tipiche dell'amministratore come abilitazione di utenti, creazione di macchine e pulizia di vetture sono state implementate mediante Web Service di tipo Rest. Con JMS è stato realizzato uno specifico caso d'uso che permette la gestione e la riparazione di guasti che possono essere presenti nelle vetture. Per quanto concerne i Web Service di tipo Rest, questi sono usati per catturare informazioni geografiche e relative alle distanze sui viaggi effettuati da un utente. È stato anche realizzato un client che fa uso di Swing per mostrare l'elenco di macchine noleggibili ed un dettaglio su ciascuna vettura. Tale client Swing, per ottenere le informazioni d'interesse, utilizza un Web Service di tipo Rest.

5.2.6 Requisiti di qualità

Lo studio di caso selezionato permette la trattazione e l'approfondimento di problematiche relative a diversi ambiti, quali:

- **Interoperabilità:** impegno richiesto per far interagire il programma con altri programmi
- **Riusabilità:** il grado con cui un programma (od alcune sue parti) può essere utilizzato di nuovo in altre applicazioni; è correlato al modo in cui il programma è organizzato ed allo scope delle funzioni svolte dal programma;
- **Modificabilità:** facilità con cui un sistema può accomodare cambiamenti [ASW]
- **Disponibilità:** la capacità di un sistema di essere completamente o parzialmente funzionante come e quando richiesto, anche a fronte di guasti di componenti del sistema [ASW]
- **Scalabilità:** la capacità del sistema di rispondere a variazioni nel suo carico applicativo [ASW]
- **Portabilità:** la facilità con la quale il software può essere trasferito da un ambiente ad un altro come indicato dai seguenti sottoattribuiti: adattabilità, facilità di installazione, conformità e sostituibilità in particolare;
- **Testabilità:** l'impegno necessario per stabilire, tramite testing, se un programma svolge la funzione prevista.

Qualità come interoperabilità, riusabilità, modificabilità, portabilità sono state ottenute grazie un uso congiunto dello stile architettonicale “Ports And Adapters (Architettura Esagonale)” di Alistar Cockburn e della metodologia Domain Driven Development. Particolare attenzione è stata quindi rivolta alla strutturazione dell'applicazione cercando inoltre di definire un modello di dominio ispirato al Model-Driven Design.

CAPITOLO 6: ARCHITETTURA ESAGONALE

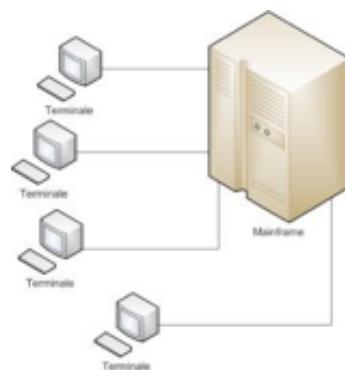
In questo capitolo è illustrata l'architettura esagonale. Tale tipo di architettura è usata nel definire la struttura della tesi.

6.1 Premessa

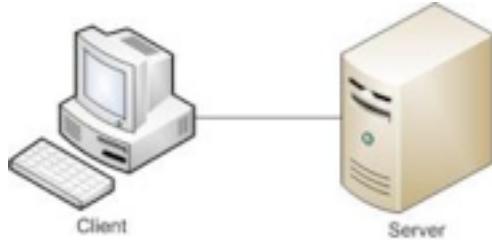
La tesi realizzata prevede che uno stesso servizio possa essere fruito in diversi modi. Affinché una diversa implementazione di un servizio non abbia ripercussioni sullo strato di presentazione o sulla logica di base, è stata utilizzata l'Architettura Esagonale. Tale stile architettonicale ha l'obiettivo di separare la logica di base dell'applicazione dai servizi utilizzati. Questo consente che servizi di natura diversa siano collegati all'applicazione e che questa possa funzionare utilizzando tali funzionalità.

6.2 Evoluzione Architettura Enterprise

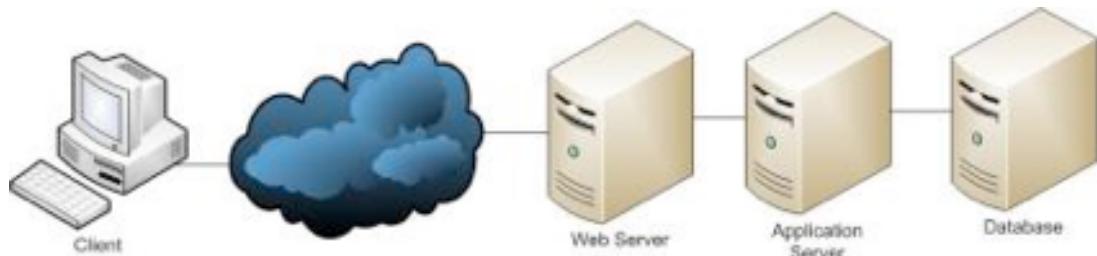
Nel corso degli anni l'architettura software ha subito delle continue innovazioni sia per motivi commerciali sia a seguito di evoluzioni tecnologiche. Focalizzandosi sulle evoluzioni tecnologiche si è passati da architetture centralizzate ad architetture distribuite. Inizialmente le architetture erano basate su Mainframe e l'accesso avveniva esclusivamente tramite terminali, tutto il software era residente su un singolo server, il Mainframe.



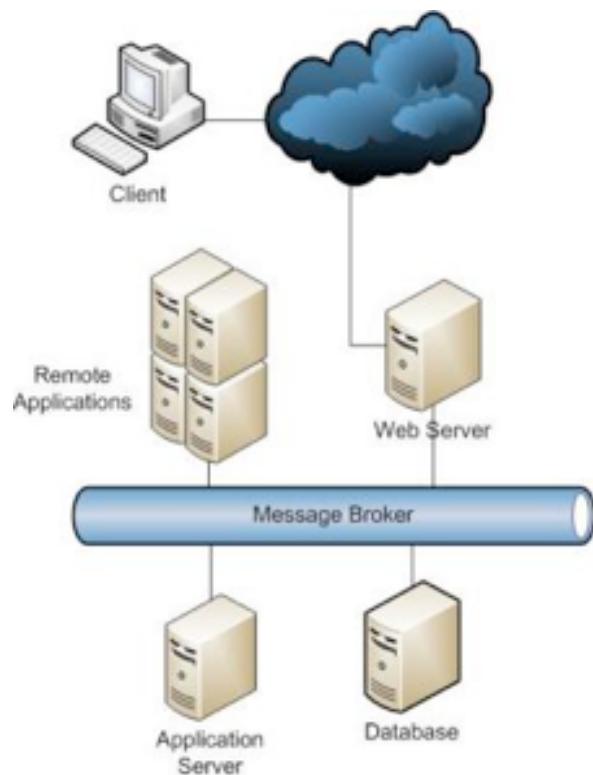
Negli anni '80 si è passati ad architetture di tipo Client-Server in cui il Client o Fat-Client si occupava di gestire le logiche di visualizzazione mentre il Server gestiva le logiche di business e la gestione dei dati.



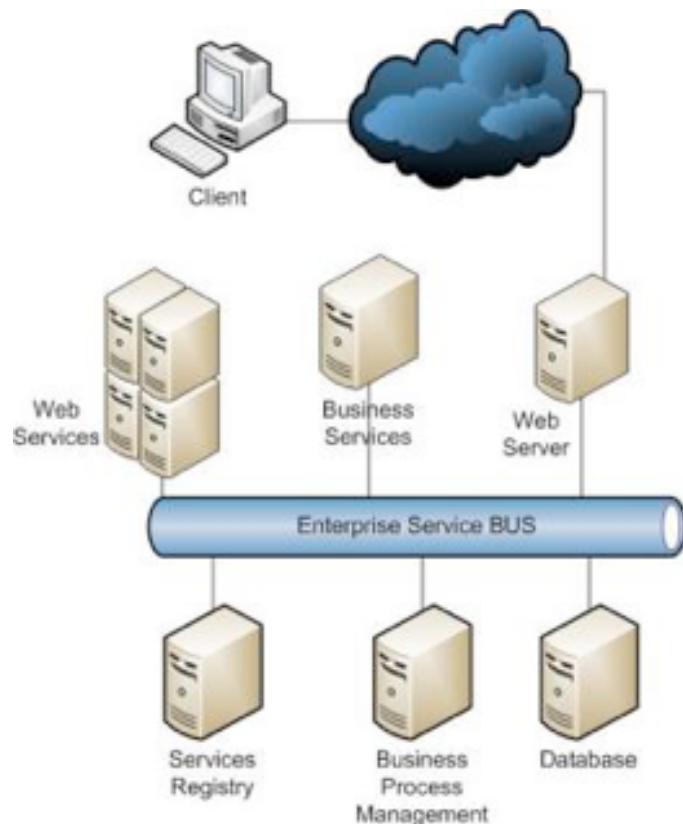
Con l'avvento di internet si è passati ad architetture di tipo 3-tier dove il Client o Thin-Client era rappresentato da semplici programmi che tramite la rete accedevano al Server che presentava una suddivisione delle competenze ripartite in presentation, business e data layer. Il presentation layer si occupava delle logiche di presentazione, il business layer delle logiche di elaborazione ed il data layer della gestione dei dati. Tali compiti erano delegati rispettivamente a Web Server, Application Server ed al Database.



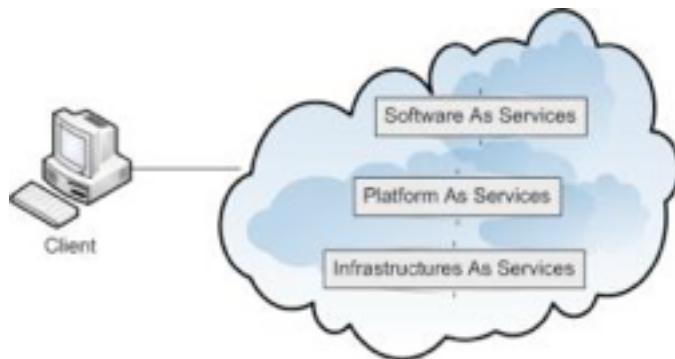
Dando uno sguardo alle modalità di intercomunicazione si è passati da una architettura di tipo EAI, ad una SOA per arrivare ai nostri giorni al Cloud. Un'architettura EAI (Enterprise Application Integration) si basa sulla presenza di un Hub, un Canale o Bus di comunicazione, che consente di mediare e centralizzare la comunicazione tra i sistemi coinvolti, evitando di configurare le connessioni secondo una modalità point-to-point e generare un sorta di "spaghetti connection". Le comunicazioni tra sistemi eterogenei sono rese possibili dall'ausilio di Adapter che "adattano" cioè traducono i messaggi provenienti dai diversi linguaggi in un formato riconoscibile.



Con l'introduzione delle architetture SOA (Service Oriented Architecture) si è passati ad introdurre ulteriori componenti: ESB, BPM, Registry, Web Services ecc; l'architettura è orientata intorno al concetto di servizio e di composizione di servizio.



L'avvento del Cloud Computing ha introdotto un grado di astrazione all'architettura sottostante che viene ridimensionata in base alle esigenze e ciò consente di ridurre quel gap tra il business e l'IT consentendo all'impresa di guadagnare quel Time-to-Market ossia essere veloce nel rispondere alle esigenze del mercato. Tutto diventa un servizio vendibile: il software, la piattaforma e l'infrastruttura.



6.3 Conseguenze evolutive

L'evoluzione dell'architettura ha generato una frammentazione delle competenze, basti guardare alla differenza tra un'architettura basata su mainframe rispetto ad una architettura SOA. Sicuramente questo consente di gestire in modo sempre più preciso una serie di requisiti non funzionali: sicurezza, performance, manutenibilità, scalabilità ecc.

Questo ha comportato che l'applicazione intensificasse notevolmente la sua rete di comunicazione con una fitta serie di componenti:

- *sistemi esterni*: vari device come cellulari, palmari, pc, tablet ecc che comunicano in formati diversi
- *sistemi interni* automatici: come sistemi di monitoraggio, test automatizzati, gestori di eventi

- *sistemi interni inter-applicativi*: componenti utilizzati dall'applicazione: Identity Access Manager, Datasource, Sistemi legacy.

Questa fitta rete di comunicazione e d'interscambio dati, se da un lato consente di distribuire le competenze, produce un disorientamento in termini di sviluppo applicativo e di esecuzione di test.

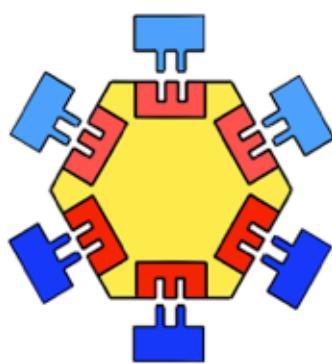
- *In termini di sviluppo*: il software non è più centralizzato in un unico posto ma è distribuito su diversi server, oltretutto replicati, pertanto occorre effettuare degli specifici interventi, per esempio modificare la presentazione (JSP, Tag file, JSF, HTML, Javascript, CSS), la logica (classi, file di configurazione ecc). Lo sviluppo pertanto è diventato distribuito, non esiste più un concetto di “centro”. Il centro è semplicemente ciò che ci interessa in quel momento.
- *In termini di test*: la loro esecuzione è sempre più spostata verso i test d'integrazione più che verso i test unitari. Di tale fatto è possibile accorgersi quando bisogna testare delle funzionalità: solo quando tutti i sistemi sono avviati è possibile fare una semplice prova. A volte anche testare un nuovo form html diventa un'impresa: configurare LDAP per l'autenticazione, il database per l'accesso ai dati, il CICS per i dati legacy, i Web Services per i servizi di profilazione ecc. Ovviamente a seconda del caso occorre configurare i sistemi di riferimento.

Tutto ciò sposta l'attenzione e richiede delle competenze trasversali.

6.4 L'architettura esagonale

Un approccio interessante a questo problema è presentato dall'**Architettura Esagonale**, anche conosciuta come pattern “**Ports and**

Adapters" di Alistar Cockburn, che si propone di centralizzare l'attenzione sull'applicazione e decentralizzare tutti i sistemi esterni. La logica di business di un'applicazione è costituita dagli algoritmi che sono essenziali per raggiungere gli scopi dell'applicazione stessa. Gli algoritmi quindi implementano i casi d'uso che sono il cuore dell'applicazione. Quando vengono apportate modifiche a tale codice, cambia l'essenza dell'applicazione. I servizi o sistemi esterni non sono essenziali. Un servizio infatti può essere sostituito con un altro senza cambiare lo scopo dell'applicazione. In senso stretto questo tipo di architettura considera l'applicazione come un quadro formato da un insieme di servizi ma il core business dell'applicazione deve essere indipendente da tali servizi. I vantaggi di tale tipo di architettura sono evidenti perché la logica di business può essere testata indipendente dai servizi/sistemi esterni. E' facile sostituire un servizio con un altro che meglio si adatta alle nuove esigenze. Tale sostituzione non comporta nessun tipo di ripercussione sul resto dell'applicazione. Un'efficace rappresentazione di tale architettura può essere la seguente:



Nel disegno, la parte in giallo rappresenta la logica di business mentre gli elementi in varie tonalità di rosso e blu rappresentano i servizi che si connettono all'applicazione mediante delle porte. Da tale raffigurazione si

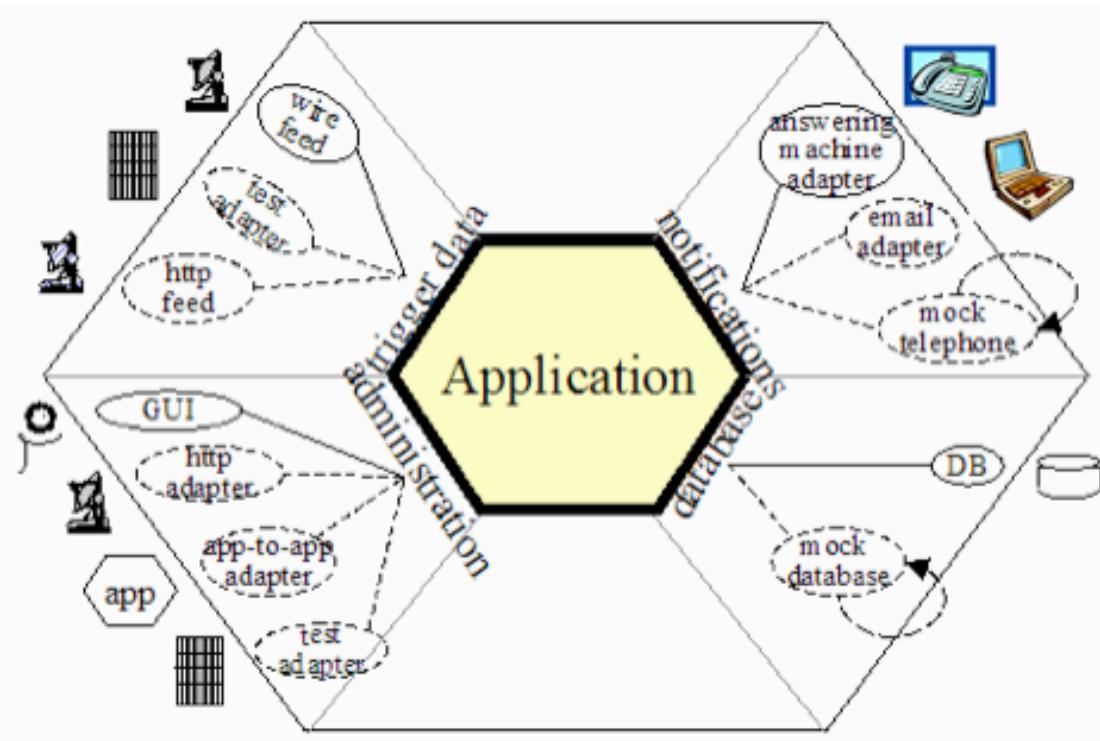
capisce perché l'architettura esagonale è anche conosciuta con il nome di “Ports And Adapter”.

L'applicazione dialoga con i servizi/sistemi esterni tramite delle porte le quali utilizzano un adapter per dialogare con diversi sistemi. A loro volta i sistemi esterni, passando per una porta applicativa, vengono filtrati da un Adapter che si occupa di tradurre il messaggio in un formato conosciuto per permettere il dialogo. Pertanto sia l'applicazione sia i sistemi esterni non conoscendo la natura del sistema contattato sono interessati solo al messaggio. Il sistema prevede una comunicazione I/O, di input/output, entrata/uscita, da/verso i sistemi esterni.

Ecco le parole dell'ideatore di tale architettura:

Create your application to work without either a UI or a database so you can run automated regression-tests against the application, work when the database becomes unavailable, and link applications together without any user involvement.

Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases. (Alistar Cockburn)



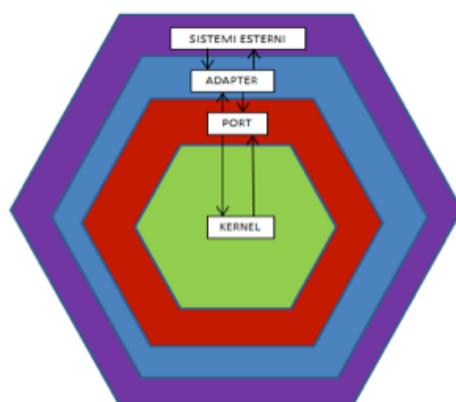
Questa architettura ha molti punti in comune con la “Onion Architecture” cioè “Architettura a cipolla” proposta da Jeffrey Palermo.

Le principali caratteristiche di questa architettura e le sue differenze rispetto ad una architettura n-tier sono:

- In una classica architettura n-tier, la chiave di lettura è da sinistra a destra, oppure dall’alto al basso; in questo caso la chiave di lettura è dal centro verso l’esterno, ossia dall’applicazione ai sistemi esterni.
- Questa architettura pone sullo stesso piano tutti i sistemi esterni all’applicazione trattando in modo simmetrico tutti gli attori. In un’architettura n-tier, esiste invece una asimmetria tra utenti e sistemi contattati, gli utenti sono gli attori principali, ossia coloro che guidano l’applicazione mentre i sistemi contattati sono gli attori secondari, ossia coloro che sono guidati dall’applicazione.
- per ogni Port è possibile associare diversi Adapter che hanno l’obiettivo di comprendere le diverse implementazioni dello stesso servizio. Per

esempio una Port può essere utilizzata per l'accesso al database che a sua volta può essere Oracle, MySql, un database in-memory, un mock.

- le Port possono essere utilizzate dai vari attori: utenza web, utenza applicative, utenze amministrative
- la rappresentazione in forma esagonale consente di visualizzare l'interazione dei sistemi esterni con l'applicazione passando per una dimensione, si riceve l'input dell'utente e si interroga il database per estrarre i dati. L'applicazione spesso interagisce con molti sistemi e questo comportamento può essere rappresentato in modo più efficace e comprensibile utilizzando una rappresentazione esagonale in cui si evidenziano le multi dimensioni
- l'applicazione non deve essere vista come una sequenza di passi che vede ad un estremo l'utente che interagisce con l'applicazione e dall'altro estremo il database; l'applicazione invece deve essere vista come un corpo centrale indipendente dai sistemi esterni che può vivere autonomamente senza avere la necessità di interagire con essi.
- l'architettura applicativa può essere paragonata all'architettura di un sistema operativo, l'applicazione è paragonabile al kernel, gli adapter sono paragonabili ai moduli del kernel, i sistemi esterni sono i device, mentre i messaggi sono delle system call.



Riassumendo i concetti principali di tale architettura sono:

- **Porta**: una porta è un entry point fornito dalla core logic. Definisce un insieme di funzioni.
- **Porte Primarie**: sono l'API principale dell'applicazione. Sono chiamate dagli adattatori primari che rappresentano il lato dell'applicazione in cui risiede l'utente. Esempi di porte primarie sono funzioni che permettono di modificare oggetti, attributi e relazioni nella logica di business.
- **Porte Secondarie**: sono le interfacce per gli adattatori secondari. Vengono chiamate dalla logica di business. Un esempio di porta secondaria è l'interfaccia per memorizzare un singolo oggetto. Questa interfaccia specifica semplicemente i metodi per creare, recuperare, aggiornare e cancellare un oggetto. Non è necessaria nessuna informazione sul modo in cui l'oggetto viene memorizzato.
- Un **Adattatore** è un ponte tra l'applicazione ed un servizio necessario all'applicazione stessa.
- Un **Adattatore Primario** è un pezzo di codice tra l'utente e la logica di business. Un adattatore potrebbe essere una funzione di test unitario per la logica di business. Un altro tipo di adattatore potrebbe essere un controller che interagisce sia con l'interfaccia grafica dell'utente che con la logica di business. L'adattatore primario chiama le API della logica di business.
- Un **Adattatore Secondario** è una implementazione di una porta secondaria (che è una interfaccia). Per esempio, può essere una semplice classe che converte le richieste d'interazione di una applicazione verso un database e restituisce i risultati provenienti dal database in un formato richiesto dalla porta secondaria. La logica di business invoca le funzioni dell'adattatore secondario.

6.5 Come funziona

Sfruttando l'architettura esagonale è possibile creare più versioni di un'applicazione. Ogni applicazione si differenzierà dalle altre per i diversi servizi che la caratterizzano nonostante mantenga un nucleo funzionale comune. I passi da compiere per realizzare un simile tipo di applicazione possono essere i seguenti:

- Viene creata un'istanza dell'applicazione e tutti gli adattatori necessari;
- Gli adattatori secondari sono passati alla logica di business (iniezione delle dipendenze);
- Gli adattatori primari vengono collegati alla logica di business e pertanto possono iniziare a guidare l'applicazione;
- L'input dell'utente viene elaborato da uno o più adattatori primari per poi passare alla logica di business;
- La logica di business interagisce solo con gli adattatori secondari;
- L'output prodotto dalla logica di business viene restituito agli adattatori primari che lo restituiscono all'utente. [MATV]

6.6 Tecniche di implementazione comuni

Per realizzare un'architettura esagonale possono essere usate alcune tecniche di implementazioni piuttosto comuni.

- Una tecnica molto usata è l'**iniezione delle dipendenze**. Tale metodo può essere usato per iniettare gli adattatori secondari alla logica di business.
- Le **interfacce** possono essere usate per definire le porte secondarie. Gli adattatori secondari implementano tali interfacce.
- E' possibile creare delle **Factory** per adattatori di uno specifico servizio.

CAPITOLO 7: PERSISTENZA

Questo capitolo è focalizzato sul data modeling. A tal proposito si intende illustrare il processo utilizzato per creare un modello concettuale del sistema di storage attraverso l'identificazione e la definizione delle entità, nonché delle loro relazioni, utili perché il sistema svolga i suoi compiti correttamente.

7.1 Problematica

Il sistema deve garantire la persistenza dei dati ma allo stesso tempo le particolari modalità per l'accesso alla base dei dati devono essere trasparenti al resto del sistema. Ottemperando a questo obiettivo sarà quindi possibile ridurre al minimo l'accoppiamento presente tra codice applicativo e livello di persistenza.

Per soddisfare tali requisiti il sistema, oltre ad un'attenta progettazione, utilizza il framework Hibernate.

7.2 Il modello

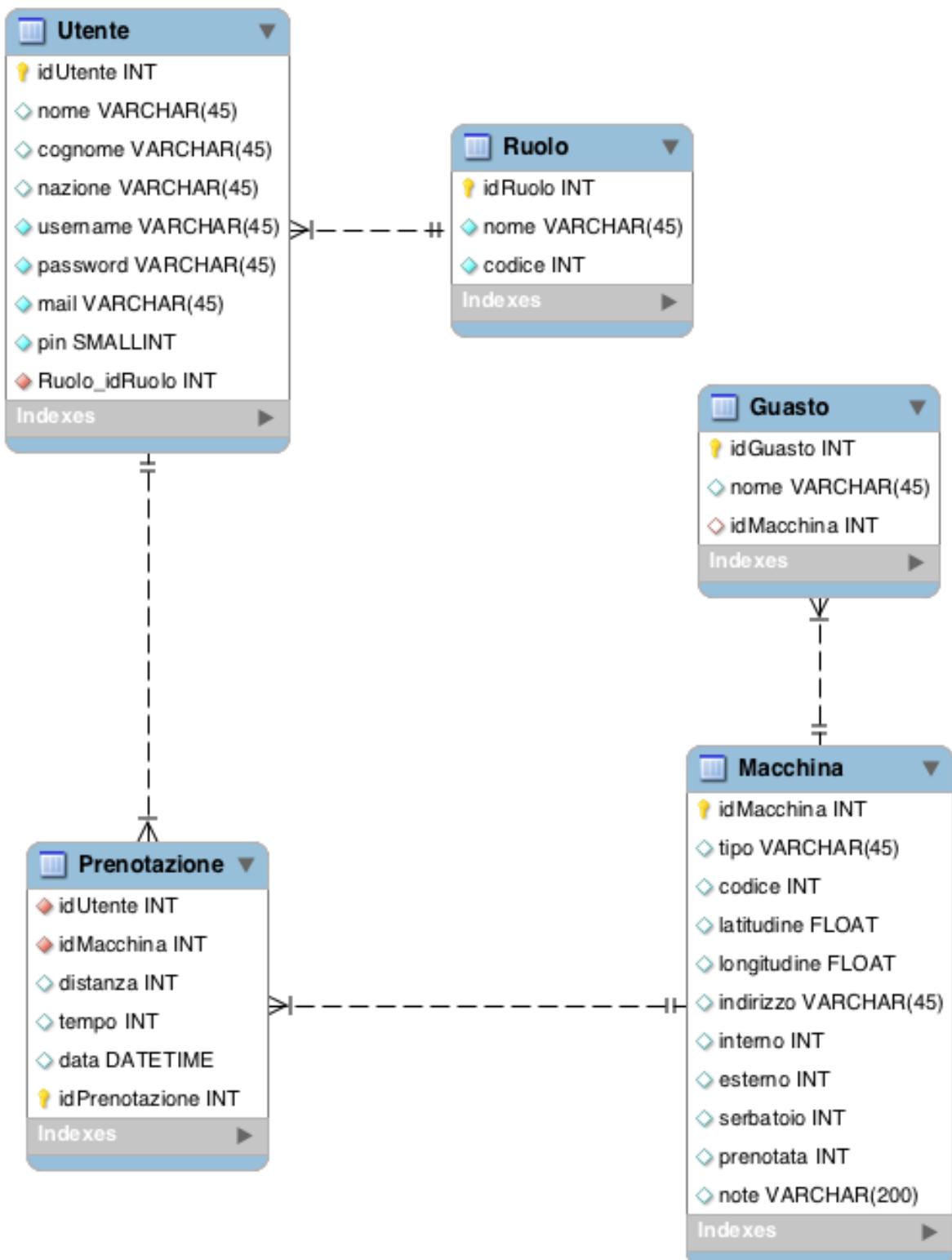
A seguito dell'analisi dei requisiti dell'applicazione, tenendo presenti le linee guida del pattern Model Driven Design, sono state individuate le seguenti entità:

- **Ruolo:** rappresenta i possibili ruoli che un utente può ricoprire. Esistono due tipi di ruoli: Admin e User. Di un ruolo è necessario conoscere il nome ed il codice.
- **Utente:** rappresenta un utente registrato al sistema. E' d'interesse conoscere il nome, cognome, nazione, username, password, mail, pin ed il Ruolo a cui appartiene.

- **Macchina:** è una vettura che il sistema può gestire. Di una macchina è necessario tenere traccia del tipo, codice, coordinate geografiche quali latitudine, longitudine, indirizzo, situazione dello stato interno ed esterno, Infine è necessario sapere se una vettura è prenotata.
- **Prenotazione:** rappresenta l'entità che deve gestire il concetto di noleggio e prenotazione. A tal proposito è necessario che tale entità gestisca informazioni inerenti all'utente che intende effettuare una prenotazione e qual è la vettura che vorrebbe usare. Per gestire i costi di un noleggio è necessario memorizzare la distanza ed il tempo di durata del viaggio. Il sistema tiene anche traccia della data in cui il noleggio è stato effettuato.
- **Guasto:** tale entità rappresenta un possibile malfunzionamento che colpisce una vettura. Di un guasto è d'interesse conoscere il nome e la vettura su cui il malfunzionamento ricade.

La figura seguente mostra il diagramma Entità Relazioni dell'applicazione di car-sharing realizzata nell'ambito del lavoro di tesi. Le classi che rappresentano il modello sono state codificate nel package car2go-model.

Modello ER applicazione
Francesco Paris
Matricola 453908



7.3 Strutturazione dei Data Access Object

Per gestire differenti tipi di accesso al database, per ogni entità, è stata creata un'interfaccia che ha la funzione di gestire l'accesso ai repository. Tali interfacce sono specificate nel progetto car2go-persistence-common, all'interno del package `it.car2go.persistence.common`. In tali interfacce ci sono i metodi per realizzare operazioni CRUD (Create, Read, Update e Delete). Le concretizzazioni di tali interfacce sono realizzate in dei componenti DAO che implementano tali interfacce. Un DAO è un oggetto che permette di stratificare ed isolare l'accesso ad una tabella del database mediante query creando quindi un maggior livello di astrazione. Seguendo le linee guida dell'Architettura Esagonale, le interfacce servono per mettere a fattor comune le operazioni che i vari DAO dovranno implementare. In questo modo se bisogna fruire del database con una tecnologia diversa sarà sufficiente realizzare dei nuovi DAO, che implementano le interfacce presenti nel package `it.car2go.persistence.common`, ma questo non avrà ripercussioni sul resto del sistema grazie all'astrazione offerta dalle interfacce. Nell'applicazione che è stata usata nel lavoro di tesi è stato usato il framework Hibernate. Quando sono stati usati gli Enterprise Java Beans (EJB), per mettere in maggior evidenza il ciclo di vita di tali componenti, è stato usato JDBC con query in Sql.

7.4 Hibernate

Hibernate è una piattaforma middleware open source per lo sviluppo di applicazioni Java che fornisce un servizio di Object Relational Mapping (ORM). L'ORM è il sistema di persistenza automatico e trasparente di oggetti in tavole di un database relazionale. Ogni oggetto viene reso persistente nel database tramite l'inserimento di nuovi record i cui campi contengono i valori degli attributi dell'oggetto. Di solito ad ogni oggetto corrisponde un record di una particolare tabella associata alla classe

dell'oggetto. L'associazione tra la classe e la tabella è ottenuta tramite l'utilizzo di file di descrizione (file di mapping), in cui si specificano le modalità di mapping tra gli attributi dell'oggetto e i campi della tabella. Ogni interrogazione è effettuata utilizzando un linguaggio simil-SQL che permette di scrivere query utilizzando il nome delle classi e degli attributi. Le interrogazioni sono convertite dal tool ORM in istruzioni SQL da eseguire sul database relazionale sottostante. I resultset delle interrogazioni sono convertiti nei corrispondenti oggetti in maniera del tutto trasparente. Dal punto di vista dello sviluppatore uno strumento ORM permette di effettuare delle richieste di istanze di particolari classi, con filtri basati sulle proprietà delle classi, i cui risultati sono liste di oggetti. Tutti i dettagli sottostanti al livello object-oriented sono praticamente nascosti.

In definitiva una soluzione ORM consiste in:

- API per eseguire le operazioni di base (CRUD) sugli oggetti delle classi persistenti
- Un linguaggio per la costruzione di query sulle classi e le proprietà delle classi
- Un sistema per specificare il mapping tramite metadati
- Un sistema interno per l'interazione con oggetti transazionali, per il dirty checking, il fetching delle associazioni lazy.

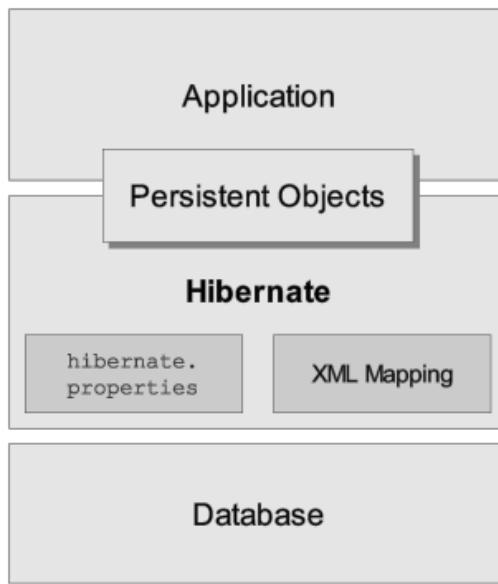
I benefici che si hanno nell'utilizzo di una soluzione ORM sono svariati:

- **Produttività:** il codice che si occupa della persistenza dei dati è la parte forse più tediosa di un'applicazione. Un tool ORM elimina molto di questo codice, semplificandolo e automatizzandolo al massimo
- **Manutenibilità:** dovendo scrivere meno codice è più facile manutenere l'applicazione. Inoltre in soluzioni fatte “a mano”, che trasformano il modello ad oggetti in query relazionali e record di database in oggetti, è più difficile far variare il modello di dominio insieme al modello relazionale. Con un tool ORM si mantengono separati i due modelli

tramite l'uso di uno strato intermedio di mapping, che spesso minimizza anche la propagazione di variazione tra i due modelli.

- **Performance:** anche se a prima vista sembra che lo strato aggiuntivo di mapping introduca dei peggioramenti alla performance, ci sono talmente tante opzioni di ottimizzazioni (utilizzo di cache di primo e secondo livello, utilizzo di batch query ecc) che in realtà, in applicazioni con un numero elevato di accessi in lettura, è possibile ottenere delle performance estremamente più efficienti di una semplice soluzione basata su SQL.
- **Indipendenza dal tipo di database:** questa è una delle caratteristiche più interessanti di uno strumento di ORM, il quale astrae l'applicazione dal sottostante database SQL e dal suo dialetto. ORM introduce un linguaggio proprietario per le interrogazioni e le operazioni CRUD. Questo linguaggio viene poi convertito automaticamente nell'SQL del database sottostante. Cambiando un parametro nella configurazione dell'ORM, è possibile portare l'applicazione su altri database senza praticamente toccare una riga di codice. Inoltre l'ORM conosce molto bene (spesso molto meglio del programmatore) il dialetto del particolare database, e ottimizza le query utilizzando quanto più possibile funzioni SQL proprietarie.

Hibernate cerca di essere una soluzione completa al problema della gestione di dati persistenti in Java. Si pone nel mezzo tra l'applicazione e un database relazionale, lasciando lo sviluppatore libero di concentrarsi sugli aspetti di business dell'applicazione. Hibernate non è una soluzione intrusiva e si adatta bene ad applicazioni nuove ed esistenti, e non richiede modifiche distruttive al resto dell'applicazione. [BHLI]



Per rendere persistente una classe è necessario specificare un mapping tra la classe ed il database. Nella tesi, per ottemperare a tale scopo, sono stati usati dei file XML per definire questa associazione. Ogni classe ha il suo file di mapping associato. Di solito il file XML ha lo stesso nome della classe con estensione “.hbm.xml”. Il file di mapping per Ruolo ha quindi nome Ruolo.hbm.xml. Si seguito il file di mapping per l’entità Ruolo.

```

<hibernate-mapping package="it.car2go.model">
    <class name="Ruolo" table="Ruolo">
        <id name="idRuolo" column="idRuolo">
            <generator class="native"/>
        </id>
        <property name="nome" column="nome" type="string"/>
        <property name="codice" column="codice" type="int"/>
    </class>
</hibernate-mapping>

```

Come è possibile notare, il mapping viene definito tramite:

- Seguono poi le definizioni delle proprietà tramite l’elemento `<property>`, ognuna delle quali è mappata con una colonna della tabella.
- Per utilizzare Hibernate deve essere costruita innanzitutto una **SessionFactory**. La SessionFactory è la factory che permette la

creazione di una Session di Hibernate, la quale può essere vista in maniera semplicistica come la cache delle istanze di oggetti Java persistenti e permette l’interazione con il database. La SessionFactory deve essere creata una sola volta, è molto dispendiosa a livello di risorse e contiene tutti gli elementi <class> in cui è specificato il nome della classe Java e la tabella corrispondente al database;

- All’interno di <class> si trova la definizione dell’id dell’oggetto tramite l’elemento <id>, in cui si definisce quale sia la politica di generazione di tale id. Nell’esempio è definito un id con una politica che delega al database sottostante il compito di creare l’id (ad esempio un campo auto-increment) tramite l’elemento <generator class=”native”/>

aspetti di configurazione per l’accesso ad un database. La SessionFactory viene configurata attraverso un file XML di nome hibernate.cfg.xml.

```
<hibernate-configuration>
<session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/car2go</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.hbm2ddl.auto">update</property>
    <property name="hibernate.current_session_context_class">thread</property>
    <property name="hibernate.transaction.manager_lookup_class">org.hibernate.transaction.SunONETransactionManagerLookup</property>
    <property name="hibernate.transaction.auto_close_session">true</property>
    <property name="hibernate.transaction.flush_before_completion">true</property>
    <mapping resource="it/car2go/model/Ruolo.hbm.xml"/>
    <mapping resource="it/car2go/model/Ruolo/Utente.hbm.xml"/>
    <mapping resource="it/car2go/model/Ruolo/Macchina.hbm.xml"/>
    <mapping resource="it/car2go/model/Ruolo/Prenotazione.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

Il file specifica:

- Il driver JDBC di connessione al database;
- Il tipo di dialetto per il particolare database da usare; cambiando dialetto e driver è possibile usare un qualsiasi altro database senza modificare assolutamente l’applicazione;
- I parametri di connessione al database
- Il gestore per le transazioni
- I file di mapping (hbm.xml) da usare per definire le classi persistenti

CAPITOLO 8: LOGICA DI BUSINESS

In questo capitolo sono descritte e motivate le decisioni assunte nell'implementazione della logica di business.

8.1 Realizzazione della logica di Business

Lo scopo della logica di business è definire dei compiti che il sistema deve soddisfare. Un requisito del lavoro di tesi è che tali compiti possano essere realizzati mediante diverse tecnologie. Per realizzare tale obiettivo sono state seguite le linee guida dettate dal pattern “Architettura Esagonale”. A tal proposito sono state specificate, all’interno del package `it.car2go.service` del progetto `car2go-service-common`, delle interfacce. Tali interfacce dichiarano, per ogni tipo di entità, delle operazioni di business. L’interfaccia `MacchinaService`, ad esempio, ha la definizione di operazioni per recuperare le macchine non prenotate, per avere l’elenco di vetture che necessitano di rifornimento o di riparazione. Tutte queste interfacce presenti nel package `it.car2go.service` costituiscono una sorta di Application Programming Interface (API) rispetto allo strato di presentazione. In questo modo lo strato di presentazione è indipendente dalla particolare tecnologia utilizzata per realizzare tali funzionalità.

Nel dettaglio le interfacce presenti sono:

- **RuoloService**: comprende operazioni per salvare, aggiornare, recuperare un ruolo.
- **UtenteService**: in tale interfaccia ci sono tutte le operazioni per la gestione delle utenze. Sono previsti metodi per verificare la presenza di un utente, per recuperare un utente o tutti gli utenti registrati al sistema.
- **MacchinaService**: permette di gestire le funzionalità relative ad un macchina. In tale interfaccia ci sono operazioni per creare, aggiornare,

recuperare e cancellare un'auto. Sono presenti anche operazioni per recuperare le macchine che necessitano di riparazione oppure che sono sporche internamente o hanno bisogno di rifornimento;

- **PrenotazioneService:** in tale interfaccia sono compresi tutti i metodi per gestire i noleggi di un'auto pertanto ci sono operazioni per aggiungere, recuperare o aggiornare una prenotazione.

Le interfacce MacchinaService, PrenotazioneService, RuoloService e UtenteService sono state implementate in vari modi usando diverse tecnologie come RMI, EJB, WebService (Soap e Rest). Un particolare caso d'uso è stato realizzato mediante JMS.

CAPITOLO 9 : INTERFACCIA WEB

In questo capitolo sono illustrate le scelte effettuate per l'implementazione dell'interfaccia web, a livello di decisioni progettuali e di tecnologie utilizzate.

Per la realizzazione di tale porzione dell'applicazione è stato usato Spring MVC, framework utilizzato per realizzare web-app basate sul modello MVC che sfrutta i punti di forza offerti dal framework Spring come l'**inversion of control** (tramite dependency injection) e la **aspect oriented programming**. Spring MVC si occupa di mappare metodi e classi Java con determinati url, di gestire differenti tipologie di “viste” restituite al client, di realizzare applicazioni internazionalizzate e di gestire i cosiddetti temi per personalizzare al massimo l’esperienza utente.

9.1 Il pattern MVC

Per comprendere al meglio il framework è necessario introdurre il pattern architettonale che implementa ovvero il modello MVC. MVC rappresenta un acronimo per Model View Controller ovvero le tre componenti principali di un'applicazione web.

- Il **Model** si occupa di accedere ai dati necessari alla logica di business implementata nell'applicazione ed è indipendente dall'interfaccia utente;
- le **View** si occupano di creare l'interfaccia utilizzabile dall'utente e che espone i dati da esso richiesti;
- i **Controller** si occupano di implementare la vera logica di business dell'applicazione integrando le due componenti precedenti, ricevendo gli input dell'utente, gestendo i modelli per la ricerca dei dati e la creazione di viste da restituire all'utente.

9.2 MVC in Spring

Spring MVC implementa perfettamente questo approccio mantenendo sia i concetti che la nomenclatura del pattern. All'interno di un'applicazione Spring MVC avremo quindi:

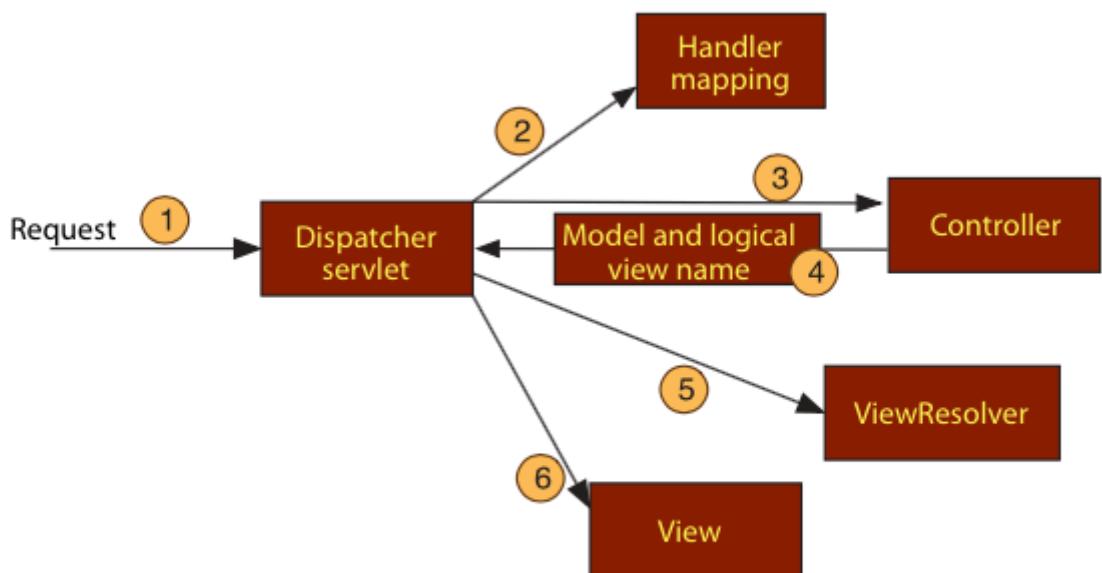
- il Model, rappresentato dalle classi che a loro volta rappresentano gli oggetti gestiti e le classi di accesso al database;
- le View, rappresentate dai vari file JSP (che vengono compilati in HTML) e da eventuali classi per l'esportazione in formati diversi da HTML (PDF, XLS, CSV...);
- i Controller, rappresentati da classi (chiamate appositamente Controller) che rimangono “in ascolto” su un determinato URL e, grazie al Model e alle View, si occupano di gestire la richiesta dell'utente.

Secondo la documentazione ufficiale Spring MVC presenta molti altri vantaggi oltre alla netta separazione tra le funzionalità:

- è adattabile, flessibile e non intrusivo grazie alla presenza di comode e chiare Java Annotations;
- permette di scrivere codice riusabile;
- possibilità di essere esteso tramite adattatori e validatori scritti ad hoc per le specifiche esigenze;
- url dinamici, SEO-friendly e personalizzabili;
- gestione integrata dell'internazionalizzazione e dei temi;
- libreria JSP sviluppata ad hoc per facilitare alcune operazioni ripetitive;
- nuovi scope per i bean (request e session) che permettono di adattare i container base di Spring anche al mondo web.

9.2.1 Gestione di una richiesta in Spring MVC

Nell'utilizzo di Spring MVC, una delle prime cose che devono essere studiate attentamente è il processo di una gestione delle richieste. Una generica richiesta nasce quando un utente, attraverso il suo browser, clicca un link oppure completa ed invia un form. Prima che al browser dell'utente giunga una risposta, la richiesta attraversa diversi step e viene processata più volte. La seguente figura riassume tutte le trasformazioni che una richiesta deve subire.



Come si evince dal punto 1, la richiesta inizialmente viene presa in carico dalla **Dispatcher servlet**. Come molti framework Java, anche Spring MVC gestisce le richieste attraverso un'unica servlet che ricopre il ruolo di **front controller**. Un front controller è un comune pattern usato nelle applicazioni web dove ad una singola servlet è affidato il compito di smistare la richiesta al componente che eseguirà l'elaborazione vera e propria. Il compito della Dispatcher servlet è quello di inviare la richiesta ad un controller. Un **controller** è un componente in grado di processare la richiesta. Una tipica applicazione è composta da

diversi controller e la Dispatcher servlet necessita di maggiori informazioni per decretare quale controller deve prendere in carico la richiesta. Per tale motivo la Dispatcher servlet consulta uno o più **Handler mapping** per capire quale sarà il prossimo componente che dovrà analizzare la richiesta. L'Handler mapping presta particolare attenzione all'URL della richiesta per aiutare la Dispatcher servlet nel suo processo decisionale. A questo punto, in base alle informazioni in suo possesso, la Dispatcher servlet può inviare la richiesta al **Controller** opportuno. Il Controller ricaverà dalla richiesta il payload (le informazioni realmente inviate dall'utente) che userà per effettuare il compito di elaborazione per il quale è stato implementato. Solitamente un Controller ben progettato compie piccole elaborazioni. Elaborazioni più complesse sono affidate alla logica di business. Il risultato dell'elaborazione del controller è, genericamente, un insieme di informazioni che devono essere trasmesse all'utente che ha inviato la richiesta. Inviare queste informazioni direttamente al client non è il modo più consone di operare. Tali dati devono essere formattati in un formato user-friendly, tipicamente in HTML. Per tale motivo è necessario trasferire tali informazioni ad una **view**. Un altro compito affidato al controller è di impacchettare il modello con il nome della vista che dovrà formare l'output. Pertanto, come mostrato in figura dalla freccia n.4, il Controller invia alla Dispatcher Servlet il modello ed il nome logico della vista. Il Controller, in questo modo, non è accoppiato con nessuna vista. Il nome logico, infatti, non identifica direttamente nessuna JSP. Alla Dispatcher Servlet viene comunicato un nome che servirà, in seguito, ad identificare quale vista sarà designata a rappresentare le informazioni. Pertanto la Dispatcher Servlet utilizzerà il **View Resolver** per identificare una specifica implementazione di una vista a partire dal nome logico. A questo punto la Dispatcher Servlet

conosce quale vista rappresenterà i dati e la gestione della richiesta iniziale dell’utente può essere considerata completata.

9.2.2 Settare Spring MVC

Come evidenziato poc’anzi, la DispatcherServlet rappresenta un elemento importante all’interno di Spring MVC. Essendo una servlet va configurata all’interno del file web.xml che compare all’interno di ogni applicazione web realizzata in Java.

```
<!-- Processes application requests -->
<servlet>
    <servlet-name>car2go</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Nel frammento del file web.xml riportato in foto è stata specificata la classe DispatcherServlet che avrà il ruolo di front controller dell’applicazione. Particolare importanza ha anche il tag servlet-name al cui interno è stato specificato il valore “car2go”. Per default, quando la DispatcherServlet è caricata, questa carica anche l’application context di Spring a partire dal file XML il cui nome è basato sul contenuto del tag servlet-name. In questo caso, poiché la servlet è chiamata car2go, la DispatcherServlet proverà a caricare l’application-context da un file chiamato car2go-servlet.xml che deve essere presente all’interno della cartella WEB-INF.

```
<!-- The definition of the Root Spring Container shared by all Servlets and Filters -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/root-context.xml</param-value>
</context-param>
```

All’interno del tag context-param è possibile definire altri file di configurazione di Spring. In questo modo è possibile applicare i principi della modularità anche ai file di configurazione di Spring. Si potrebbe infatti usare un file di configurazione per ogni scopo: data source, persistenza e sicurezza.

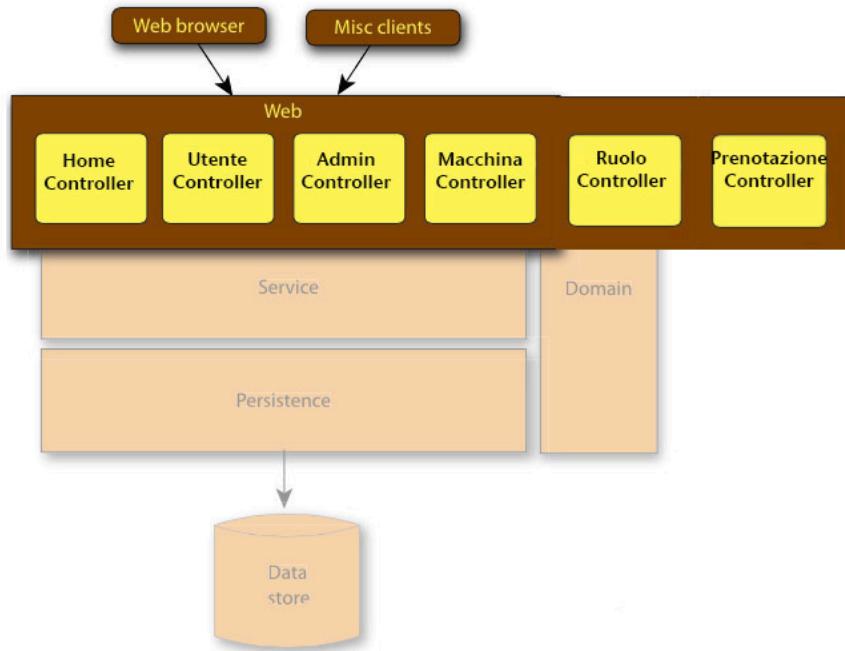
9.3 Controller

Settato l'ambiente per utilizzare Spring MVC, è necessario stabilire quanti e quali controller realizzare. Il principio di base seguito è stato quello di implementare un singolo controller per ogni tipo elementare di risorsa che l'applicazione deve gestire.

Nell'applicazione sono stati quindi realizzati i seguenti controller:

- **UtenteController**
- **MacchinaController**
- **RuoloController**
- **PrenotazioneController**

Ai seguenti controller, viste le numerose operazioni che deve compiere un utente amministratore, è stato aggiunto un apposito controller chiamato **AdminController** proprio per gestire in modo più efficiente le funzionalità riservate all'Admin.



Ai controller appena citati va aggiunto l'**HomeController** il cui compito è quello di svolgere le operazioni necessarie a mostrare le informazioni sulla

prima pagina dell'applicazione. E' stato realizzato un HomeController perché i compiti che deve realizzare non sono associati direttamente a nessun concetto di base dell'applicazione.

L'HomeController è il primo controller che deve essere scritto.

```
@Controller
public class HomeController {

    @Autowired
    private MacchinaService macchinaService;

    @Inject
    public HomeController(MacchinaService macchinaService){
        this.macchinaService = macchinaService;
    }

    @RequestMapping({" "/" , "/home"})
    public String showHomePage(Map<String, Object> model){

        List<Macchina> lista = macchinaService.getMacchineNonPrenotate();
        int size = lista.size();

        model.put("macchine", lista);
        model.put("totaleMacchine", size);

        return "home";
    }
}
```

Come si evince dal codice mostrato, è possibile e consigliato usare le annotazioni per facilitare il lavoro di implementazione. Mediante l'annotazione `@Controller` s'indica che la classe HomeController è un controller. L'annotazione `@Controller` è una specializzazione dell'annotazione `@Component` ed in virtù di questa relazione Spring scopre e regista HomeController direttamente come un bean senza nessuna configurazione esplicita da parte del programmatore nel file dell'application-context. Per raggiungere gli scopi per i quali l'HomeController è stato creato, quest'ultimo necessita di recuperare

l’elenco delle macchine non prenotate. Tale lista sarà poi passata alla vista che si occuperà di comporre il codice HTML da mandare in output. Per recuperare la lista di auto non prenotate viene fatto uso di MacchinaService. Tale interfaccia specifica delle operazioni di business la cui reale implementazione dipende dalla particolare tecnologia usata. Nel caso di car2go-simple-web esisterà un’apposita classe che realizzerà l’implementazione di MacchinaService. Nella realizzazione dell’applicazione mediante RMI, MacchinaService è l’interfaccia remota del servizio. La reale implementazione sarà affidata a degli elementi Servant lato Server. Non implementando la logica di business direttamente nel Controller ed usando il meccanismo delle interfacce per la realizzazione di Porte ed Adattatori è stato raggiunto lo scopo di ottenere l’indipendenza dalla tecnologia con la quale sono implementate le operazioni di business. Il compito di un controller è pertanto quello di catturare le richieste ed i relativi dati provenienti dall’utente ed, in base a questi, attivare l’opportuna funzionalità di business. I dati ricavati dai metodi di business saranno poi passati alla vista che si occuperà di mostrarli all’utente finale. Attraverso il costruttore di HomeController e mediante le annotazioni `@Autowired` ed `@Inject` viene iniettato in MacchinaService un riferimento ad un oggetto che implementa l’interfaccia MacchinaService. `showHomePage` è il metodo che si occupa di recuperare l’elenco delle macchine e passarlo ad una vista. Tale metodo è annotato con `@RequestMapping({"/", "/home"})` per indicare che ogni richiesta il cui path è formato da / oppure /home sarà gestita dalla funzione `showHomePage`. Tale metodo, come parametro di input prevede una Map di Stringa-Oggetto. Tale Map rappresenta il modello. Dopo aver recuperato la lista delle macchine non prenotate tale elenco viene aggiunto al modello in modo da essere mostrato dalla vista. Per concludere l’analisi di tale controller è necessario discutere del valore di ritorno. Come si nota dal codice precedente il metodo restituisce un

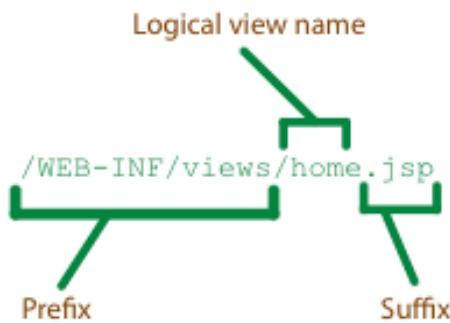
oggetto di tipo String. Tale stringa è il nome logico della vista che dovrà mostrare i dati. Un controller, come precedentemente spiegato, non deve prendere parte direttamente al processo di individuazione della reale vista che dovrà formattare in modo opportuno i dati ma dovrà solo identificare un nome logico di una vista. Una volta che il controller ha terminato il suo lavoro la DispatcherServlet, mediante l'aiuto del ViewResolver, userà tale nome logico per trovare la vera vista che si occuperà di presentare i dati all'utente.

Il ViewResolver va configurato all'interno dell'application-context principale ossia, nel caso dell'applicazione, car2go-servlet.xml.

Di seguito si riporta la configurazione usata.

```
<!-- Resolves views selected for rendering by @Controllers to .jsp resources in the /WEB-INF/views director-->
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
</bean>
```

Il ViewResolver usato è InternalResourceViewResolver. Il comportamento di tale resolver è molto semplice e può essere esemplificato dalla figura seguente.



InternalResourceViewResolver aggiunge al nome logico il suffisso .jsp mentre il prefisso è /WEB-INF/views/.

Spring mette a disposizione diversi resolver. E' possibile scegliere il resolver più appropriato in base alle esigenze dell'applicazione che si intende realizzare.

Di seguito una tabella che mostra nome del View Resolver e una breve descrizione.

View resolver	Description
BeanNameViewResolver	Finds an implementation of View that's registered as a <bean> whose ID is the same as the logical view name.
ContentNegotiatingViewResolver	Delegates to one or more other view resolvers, the choice of which is based on the content type being requested. (We'll talk more about this view resolver in chapter 11.)
FreeMarkerViewResolver	Finds a FreeMarker-based template whose path is determined by prefixing and suffixing the logical view name.
InternalResourceViewResolver	Finds a view template contained within the web application's WAR file. The path to the view template is derived by prefixing and suffixing the logical view name.
JasperReportsViewResolver	Finds a view defined as a Jasper Reports report file whose path is derived by prefixing and suffixing the logical view name.
ResourceBundleViewResolver	Looks up View implementations from a properties file.
TilesViewResolver	Looks up a view that is defined as a Tiles template. The name of the template is the same as the logical view name.
UrlBasedViewResolver	This is the base class for some of the other view resolvers, such as InternalResourceViewResolver. It can be used on its own, but it's not as powerful as its subclasses. For example, UrlBasedViewResolver is unable to resolve views based on the current locale.
VelocityLayoutViewResolver	This is a subclass of VelocityViewResolver that supports page composition via Spring's VelocityLayoutView (a view implementation that emulates Velocity's VelocityLayoutServlet).
VelocityViewResolver	Resolves a Velocity-based view where the path of a Velocity template is derived by prefixing and suffixing the logical view name.
XmlViewResolver	Finds an implementation of View that's declared as a <bean> in an XML file (/WEB-INF/views.xml). This view resolver is a lot like BeanNameViewResolver except that the view <bean>s are declared separately from those for the application's Spring context.
XsltViewResolver	Resolves an XSLT-based view where the path of the XSLT stylesheet is derived by prefixing and suffixing the logical view name.

9.4 Applicazione Web

Attraverso l'interfaccia web, un **utente non registrato** può accedere a diverse funzionalità, tra cui:

- Visualizzare l’elenco delle auto disponibili;
- Visualizzare per ogni auto il dettaglio su posizione, pulizia interna, stato esterno e benzina;
- Registrazione al sistema;
- Autenticazione;

Un **utente che si è registrato** al sistema, dopo essersi autenticato può:

- Visualizzare il riepilogo dei proprio noleggi;
- Visualizzare il riepilogo dei costi;
- Prenotare un’auto, previa autenticazione;
- Annullare la richiesta di prenotazione di un’auto;
- Effettuare la simulazione di guid;
- Effettuare il logout.

Un utente denotato come **Amministratore**, una volta autenticato, può:

- Abilitare al ruolo di Amministratore utenti semplici;
- Creare nuove macchine;
- Effettuare il rifornimento alle auto;
- Pulire internamente le auto;
- Riparare le auto;
- Effettuare il logout;

Queste operazioni sono offerte all’interfaccia web da tutti gli elementi che implementano le interfacce presenti in car2go-service-common. Le interfacce presenti nel package it.car2go.service, ossia MacchinaService, PrenotazioneService, RuoloService, UtenteService, rappresentano una API per la versione web dell’applicazione verso la logica di business. Gli oggetti che implementano tali interfacce, sebbene realizzati con tecnologie differenti, si devono occupare di accedere ai dati per realizzare le operazioni di business.

Di seguito alcune immagini che raffigurano l'applicazione in esecuzione. La seguente immagine mostra l'home page dove è possibile, a partire dal menu, scegliere di effettuare il login oppure registrarsi. E' possibile, attraverso l'elenco presente nella sezione "Le Macchine", selezionare una vettura per vederne tutti i dettagli.

Richiedendo di vedere il dettaglio di una macchina si ottiene una schermata come la seguente:



Car-Sharing

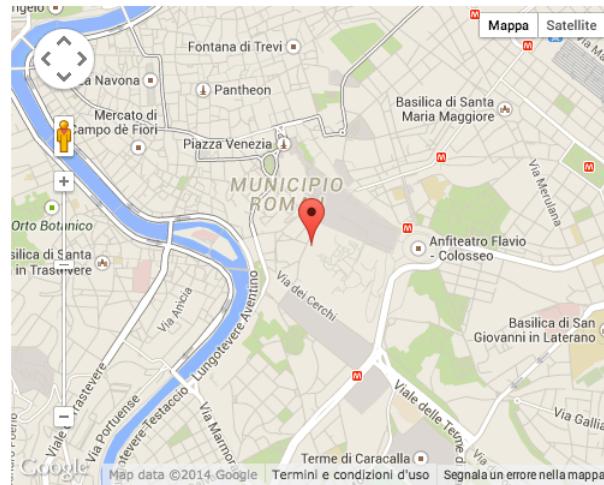
Prenota la tua auto

CarSharing

Progetto per la Tesi di
Laurea Magistrale in
Ingegneria Informatica.
Francesco Paris.
Matricola: 453908

Menu':[Home](#)
[Login](#)
[Registrati](#)**Dettaglio Macchina Doble**Le informazioni sulla macchina **Doble**.

- **Modello:** Doble
- **Codice:** 4789
- **Indirizzo:** Via di San Teodoro, 7, 00186 Roma, Italy
- **Benzina:**
- **Stato Interno:**
- **Stato Esterno:**
- **Note:** ok nuova botta esterno ok Macchina ok



Le icone di benzina e stato interno/esterno cambiano colore a seconda dello stato della macchina. I valori di latitudine e longitudine sono usati per rappresentare (attraverso il marker rosso) la posizione della macchina sulla Mappa di Google. La schermata per permettere la registrazione o l'autenticazione di un utente è un semplice form. Di seguito si riporta la schermata per registrare un nuovo utente.



Una volta che un nuovo utente si sarà registrato, per default, gli sarà assegnato il ruolo User. Solo l'Amministratore ha la possibilità di eleggere Admin un utente semplice.

L'immagine seguente mostra la pagina che sarà visualizzata quando un utente con ruolo User si autenticherà al sistema. Come si evince dal menù è possibile consultare lo storico dei noleggi effettuati e vedere i costi sinora sostenuti.



Ecco la schermata raggiungibile selezionando la voce “Costo Totale” dal menù presente a sinistra della precedente immagine.



[Home](#) [Prenota](#) [Logout](#)

Car-Sharing

Prenota la tua auto

CarSharing

Progetto per la Tesi di
Laurea Magistrale in
Ingegneria Informatica.
Francesco Paris.
Matricola: 453908

Menu:

[Home](#)
[Prenota Auto](#)
[Storico Prenotazioni](#)
[Costo Totale](#)
[Logout](#)

Riepilogo Costi



Numero Noleggi Effettuati: 5
Distanza Totale: 36 km
Costo Totale: 18.84999999999998
Euro

Copyright © 2014, Francesco Paris - Matricola 453908.

[Home](#) [Prenota](#) [Logout](#)

La funzionalità più complessa è quella relativa a Prenota Auto. Per noleggiare un'auto, la prima cosa che l'utente dovrà fare, sarà la scelta di una vettura tra quelle al momento disponibili. Tale azione si può compiere scegliendo, mediante clic sul pulsante grigio, una fra le vetture disponibili.

Car-Sharing

Prenota la tua auto

CarSharing

Progetto per la Tesi di
Laurea Magistrale in
Ingegneria Informatica.
Francesco Paris.
Matricola: 453908

Menu:

[Home](#)
[Prenota Auto](#)
[Storico Prenotazioni](#)
[Costo Totale](#)
[Logout](#)

Prenota Auto

Elenco di macchine libere che e' possibile prenotare

- Macchina: Smart
[Prenota Smart](#)
- Macchina: Mercedes
[Prenota Mercedes](#)
- Macchina: Doble
[Prenota Doble](#)

Copyright © 2014, Francesco Paris - Matricola 453908.

[Home](#) [Prenota](#) [Logout](#)

Scelta la vettura l'utente si troverà proiettato in un'altra schermata come la seguente:

119

CarSharing

Progetto per la Tesi di Laurea Magistrale in Ingegneria Informatica.
Francesco Paris.
Matricola: 453908

Guida Auto

Prenotazione effettuata

L'utente Luca Toni ha effettuato la prenotazione di una macchina.

- Macchina: Smart
- Benzina: 100 %

Stato pulizia Interna della macchina:
Ottimo ▾

Stato Esterno della macchina:
Ottimo ▾

Inserire note particolari sullo stato della macchina:
ok

Invia i dati e guida **Guida**

Annulla Prenotazione!

E' possibile annullare la guida e la prenotazione.
Vuoi annullare la prenotazione?

Si, Elimina Prenotazione Macchina Smart

Come si nota dall'immagine, tale pagina è suddivisa in due parti. La parte inferiore permette ad un utente di annullare la prenotazione. Cliccando sul pulsante grigio, la vettura appena prenotata tornerà ad essere libera pronta per accettare altre prenotazioni. La parte superiore della pagina presenta alcune informazioni sull'utente che ha effettuato la prenotazione e sull'auto oggetto del noleggio. Prima di poter confermare l'intenzione di guidare il mezzo, proprio come accade all'interno di una vettura in un reale sistema di car sharing, l'utente è chiamato a dare un giudizio sullo stato di pulizia interno della vettura, sulle condizioni esterne ed, eventualmente, a specificare alcune note aggiuntive. Cliccando su Guida, verrà simulato un percorso all'interno del Grande Raccordo Anulare di Roma. Il sistema genererà delle nuove coordinate di latitudine e longitudine e calcolerà la distanza dalla destinazione randomica appena creata ed il punto di partenza. Come già spiegato per i dati geografici il sistema usa il servizio messo a disposizione da Google Maps. L'applicazione, in base alla distanza, stima

tempi e costi del viaggio ed infine abbassa il livello di carburante. Al termine di questa elaborazione l'utente verrà riportato nella sua home page dove potrà vedere i dettagli del viaggio appena effettuato nello storico relativo ai noleggi ed ai costi.

L'altra tipologia di utente che può interagire con l'applicazione è l'Admin.



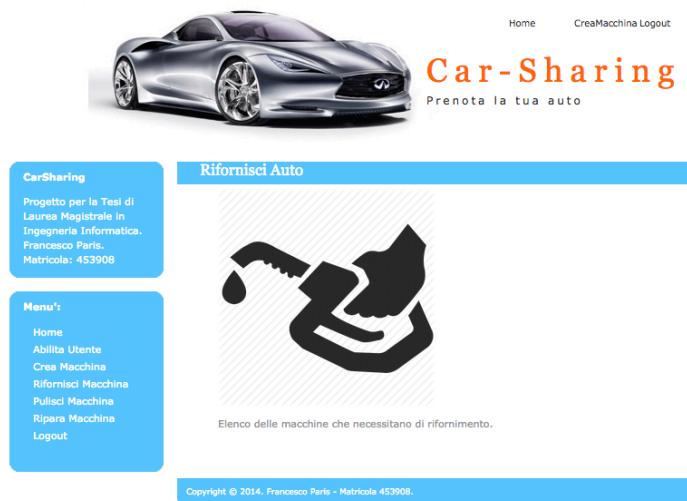
Un Admin, come mostra la schermata precedente ottenuta dopo il login, può abilitare un utente oppure creare e gestire le automobili. Nell'immagine seguente si mostra la schermata mediante la quale un amministratore può abilitare un utente.



La creazione di una nuova macchina verrà effettuata mediante un semplice form dove sarà cura dell'Amministratore inserire tipo e codice della

vettura. I valori relativi alla posizione geografica saranno completati in modo randomico direttamente dall'applicazione.

Le schermate per rifornire, pulire e riparare una vettura sono fondamentalmente tutte uguali.



Tali pagine sono costituite da un elenco di macchine che necessitano di essere rifornite/pulite/riparate. L'amministratore può completare l'azione cliccando sul pulsante grigio. Terminato tale compito l'amministratore verrà riportato sulla sua home page. Un utente che si è autenticato al sistema, di quale tipo esso sia, può effettuare il logout. Un'apposita pagina confermerà che l'azione di logout è stata completata con successo. Eventuali errori dell'applicazione causati da un inserimento errato dei dati da parte dell'utente saranno notificati a quest'ultimo mediante un'apposita "pagina d'errore". Nel caso della registrazione di un nuovo utente, se i campi del form sono compilati in maniera errata, verranno mostrati dei suggerimenti nella stessa pagina dove è presente il modulo.

9.5 Deployment

Il progetto contenente la parte grafica dell'applicazione è stato rilasciato in GlassFish, application server Java EE. Nulla vietava di utilizzare, come server, Apache Tomcat.

CAPITOLO 10: REMOTE METHOD INVOCATION - RMI

In questo capitolo sono illustrate le scelte effettuate per la realizzazione di una versione dell'applicazione basata su RMI.

10.1 Premessa

Per lo sviluppo dell'applicazione si ha la necessità di richiedere dei servizi presenti in oggetti che risiedono in processi diversi, eventualmente anche in un altro calcolatore. Per raggiungere tale scopo è stata usata la tecnologia Java RMI – Remote Method Invocation

10.2 JAVA RMI

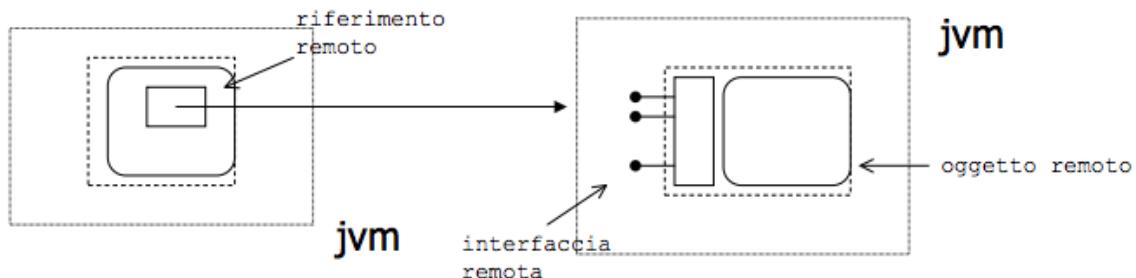
Java Remote Method Invocation (RMI) è l'implementazione Java del modello ad oggetti distribuiti che fornisce un'astrazione di programmazione distribuita orientata agli oggetti. In prima istanza, secondo questo modello, esiste una collezione di oggetti distribuiti che possono risiedere in processi diversi. Ogni oggetto può offrire servizi pubblici, mediante un'interfaccia, ed incapsulare uno stato privato. Gli oggetti cooperano mediante l'invocazione di metodi. In questo modello è possibile distinguere tra due tipi di oggetto: locali o remoti. Gli oggetti locali sono visibili localmente a un processo mentre quelli remoti possono essere distribuiti su più calcolatori o processi. Ciascun oggetto remoto implementa un'interfaccia remota che specifica quali metodi possono essere invocati remotamente.

In Java RMI la sintassi per le invocazioni remote è esattamente la stessa di quella delle invocazioni locali:

- Possono essere passati argomenti, calcolati nel contesto della macchina chiamante;
- Possono essere restituiti valori, calcolati nel contesto della macchina remota,
- Gli elementi chiave sono serializzazione e dynamic class loading.

Come già accennato, in Java RMI tre sono i concetti fondamentali:

- **Interfaccia Remota**: interfaccia java che specifica i metodi che un oggetto rende visibile ad altri oggetti come metodi remoti;
- **Oggetto Remoto**: oggetto java che implementa un'interfaccia remota;
- **Riferimento Remoto**: riferimento ad un oggetto remoto.



La semantica di un'invocazione remota non è uguale però a quella di un'invocazione locale. I tipi di dati primitivi (int, float, ...) sono sempre passati per valore (semantica Call By Value, CBV) ed il chiamato lavora su una copia del tipo. Gli oggetti remoti sono sempre passati per riferimento (Call By Reference, CBR). Il chiamato, in questo caso, riceve un riferimento all'oggetto.

La tabella seguente riassume tutti i casi che si possono verificare indicando per ogni tipo di dato il tipo di chiamata.

Invocazione	Tipi primitivi	Oggetti locali	Oggetti remoti
Locale	CBV	CBR	CBR
Remota	CBV	CBV	CBR

Un aspetto importante nelle applicazioni distribuite è la separazione fra interfaccia resa visibile all'esterno e l'implementazione dell'interfaccia stessa. L'interfaccia è definita mediante un linguaggio detto **Interface Definition Language**, IDL.

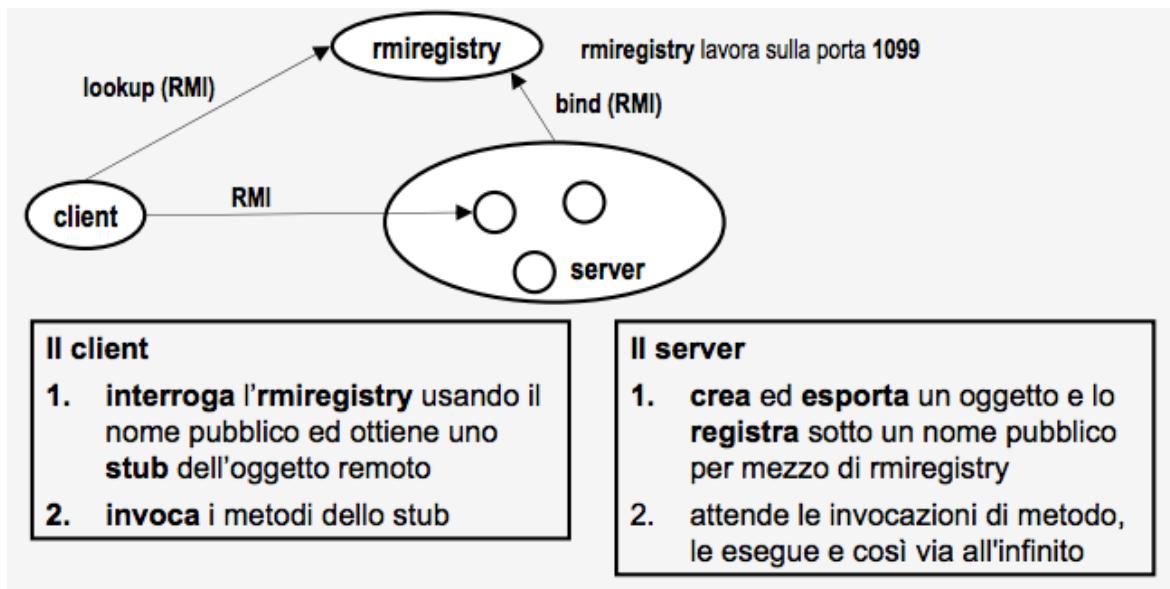
Il modo di operare tipico in Java RMI è il seguente:

- Si progetta l'interfaccia remota;
- Si implementa l'interfaccia remota in oggetti che assumeranno il ruolo di Serventi;
- Si implementa il Server;
- Si implementa il Client;
- Si avvia l'RMI compiler (rmic) per produrre lo Stub e lo Scheleton;
- Si avvia l'RMI Registry;
- Si registra il servizio nell'RMI Registry.

Il compito del Server è quello di istanziare l'oggetto servente, esportarlo come oggetto remoto registrandolo, attraverso un associazione con un nome simbolico, presso l'object registry (operazione **bind**).

L'RMI-Registry è un server RMI che gestisce le annotazioni nome simbolico – oggetto remoto.

I client utilizzeranno il nome simbolico per interrogare il registry ed ottenere un riferimento remoto per l'oggetto (operazione **lookup**).



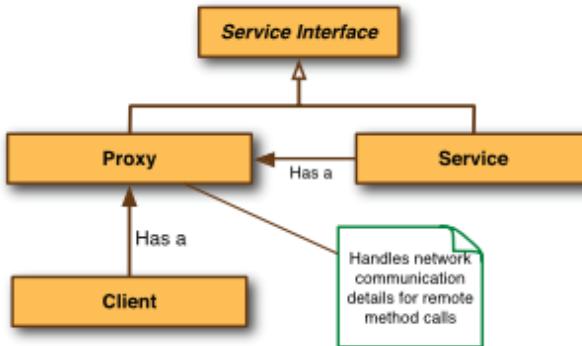
Il client quindi non colloquierà direttamente con l'oggetto remoto ma con un suo rappresentante chiamato **Stub**. Lato server esiste lo Skeleton che è il corrispondente dello Stub ed il cui compito è quello di accettare le comunicazioni dal client e tradurle in invocazioni.

10.3 RMI e Spring

Dopo aver effettuato una veloce panoramica su Java RMI è necessario analizzare come Spring renda molto più semplice usufruire di un servizio esposto mediante RMI. Nell'ambito del lavoro di tesi sono state sfruttate proprio le facilitazioni messe a disposizione da Spring per integrare, all'interno di un'applicazione web, dei servizi esposti mediante RMI.

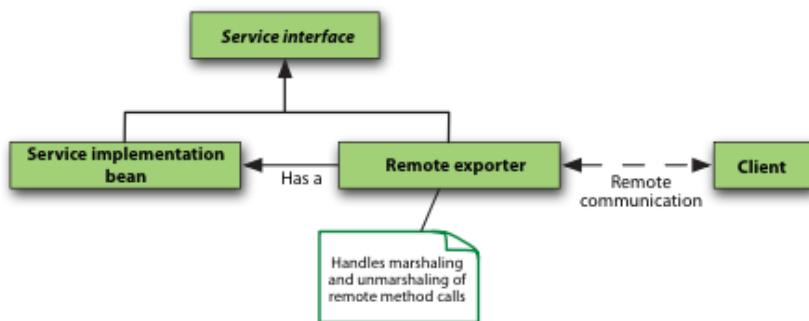
Sviluppare ed accedere a un servizio RMI, come è stato spiegato poc'anzi, comprende un insieme di attività sia di programmazione che manuali. Molte azioni da compiere sono ripetitive e possono rappresentare delle fonti d'errore. Spring semplifica il modello di RMI fornendo un proxy factory bean che consente di collegare i servizi esposti in RMI all'interno

dell'applicazione come se fossero Java Beans locali. La figura sottostante mostra come ciò sia possibile.

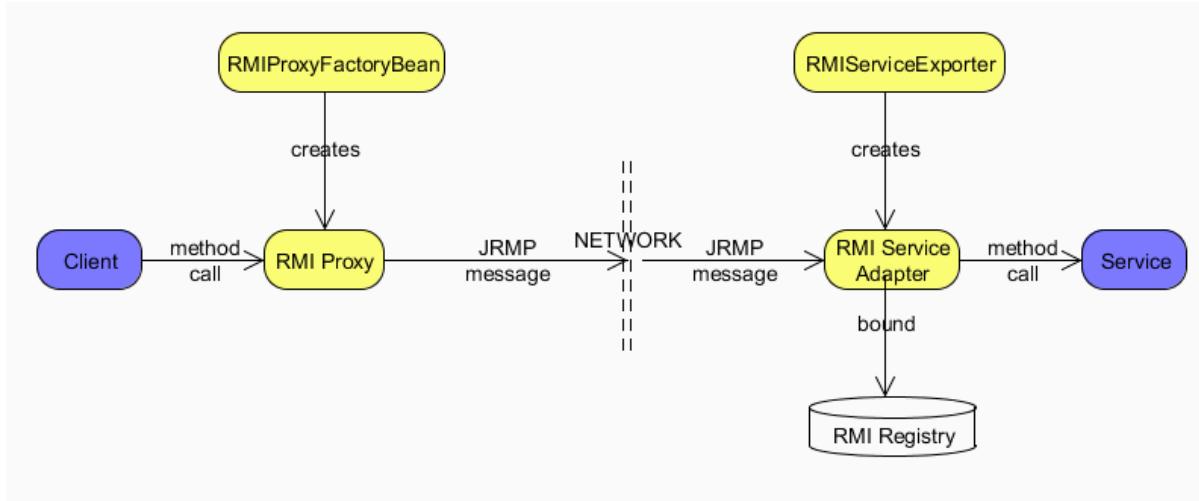


Dall'immagine si nota come il Client effettui delle chiamate al Proxy come se fosse il Proxy stesso a fornire le funzionalità del servizio. Il Proxy quindi comunica con il servizio remoto per conto del client gestendo, inoltre, tutti i dettagli della comunicazione remota.

La figura sottostante mostra come il Remote Exporter esponga metodi di servizi remoti.



Il seguente diagramma d'interazione illustra, in modo più dettagliato, il lavoro svolto da Spring Framework. I riquadri in viola devono essere implementati dall'utente mentre gli elementi in giallo sono gestiti direttamente da Spring.



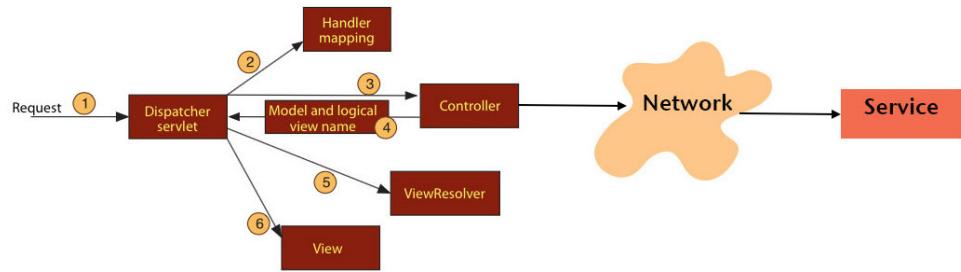
Dalla figura si evince che il Client, come detto, effettua la richiesta al Proxy creato dall'RMIProxyFactoryBean. Il Proxy converte la chiamata in un'invocazione remota che rispetta il protocollo JRMP. La chiamata viene intercettata dall'RMI Service Adapter, creato dall'RMIServiceExporter, che la inoltra finalmente al servizio.

In Spring, se si sta sviluppando codice che utilizza servizi remoti, l'implementazione o l'uso di tali servizi è puramente una questione di configurazione. In Spring non sarà pertanto necessario scrivere alcuna riga di codice per supportare la comunicazione remota.

10.4 Progetto Rmi

Per quanto riguarda il lavoro di tesi, è possibile identificare due macro-componenti: il client ed il server. Il ruolo del client è svolto dall'applicazione web. La richiesta inviata da un utente verrà gestita da

Spring MVC ed in particolare giungerà ad un controller opportuno. All'interno del controller ci sarà l'invocazione del servizio remoto.



Per quanto riguarda il server sono state create le seguenti interfacce remote:

- MacchinaService;
- PrenotazioneService;
- RuoloService;
- UtenteService.

Tali interfacce sono implementate da elementi, che ricoprono il ruolo di serventi, così chiamati:

- MacchinaServant;
- PrenotazioneServant;
- RuoloServant;
- UtenteServant.

Il frammento di codice seguente riporta l’interfaccia MacchinaService

```
package it.car2go.service;

import it.car2go.model.Macchina;

public interface MacchinaService {
    List<Macchina> getMacchine();
    List<Macchina> getMacchineNonPrenotate();
    Macchina getMacchina(int id);
    void updateMacchina(Macchina macchina);
    void addMacchina(Macchina macchina);
    List<Macchina> getMacchineSenzaBenzina();
    List<Macchina> getMacchineSporcheInterno();
    List<Macchina> getMacchineSporcheEsterno();
}
```

Tale interfaccia sarà implementata dall’elemento MacchinaService. Tale elemento, per completare gli scopi per i quali è stato creato, userà dei DAO per l’interazione con il database.

```
public class MacchinaServant implements MacchinaService {

    private MacchinaDAO macchinaDao;

    public MacchinaServant() {
    }

    public MacchinaDAO getMacchinaDao() {
        return macchinaDao;
    }

    public void setMacchinaDao(MacchinaDAO macchinaDao) {
        this.macchinaDao = macchinaDao;
    }

    public List<Macchina> getMacchine() {
        List<Macchina> lista = macchinaDao.getMacchine();
        return lista;
    }
}
```

Analizzata la conformazione del progetto è necessario capire, a questo punto, come avviene l’esposizione remota di tali servizi mediante Spring. Come visto, è compito dell’elemento Server istanziare l’oggetto servente, esportarlo come oggetto remoto per poi registrarla presso l’object registry. In Spring il Server, per raggiungere tali scopi, deve soltanto riedere il caricamento dell’ApplicationContext di Spring.

```

public class Server {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ApplicationContext context = new ClassPathXmlApplicationContext("spring-config-server.xml");
        System.out.println("Attendo richieste dai client ....");
    }
}

```

La definizione dell'ApplicationContext è contenuta all'interno del file spring-config-server.xml.

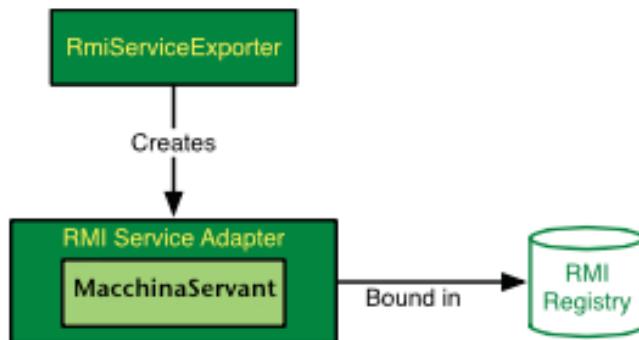
In tale file ci saranno specificate le regole per permettere l'iniezione delle dipende tra i serventi ed i DAO nonché le definizioni del servizio remoto.

```

<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <!-- does not necessarily have to be the same name as the bean to be exported -->
    <!-- serviceName represents RMI Service Name -->
    <property name="serviceName" value="macchinaService"/>
    <!-- service represents RMI Object(RMI Service Impl) -->
    <property name="service" ref="macchinaService"/>
    <!-- serviceInterface represents RMI Service Interface exposed -->
    <property name="serviceInterface" value="it.car2go.service.MacchinaService"/>
    <!-- defaults to 1099 -->
    <property name="registryPort" value="2099"/>
    <!-- <property name="registryHost" value="192.168.50.102"/> -->
</bean>

```

Le linee di codice appena mostrate sono quelle necessarie per esportare MacchinaService come servizio remoto.



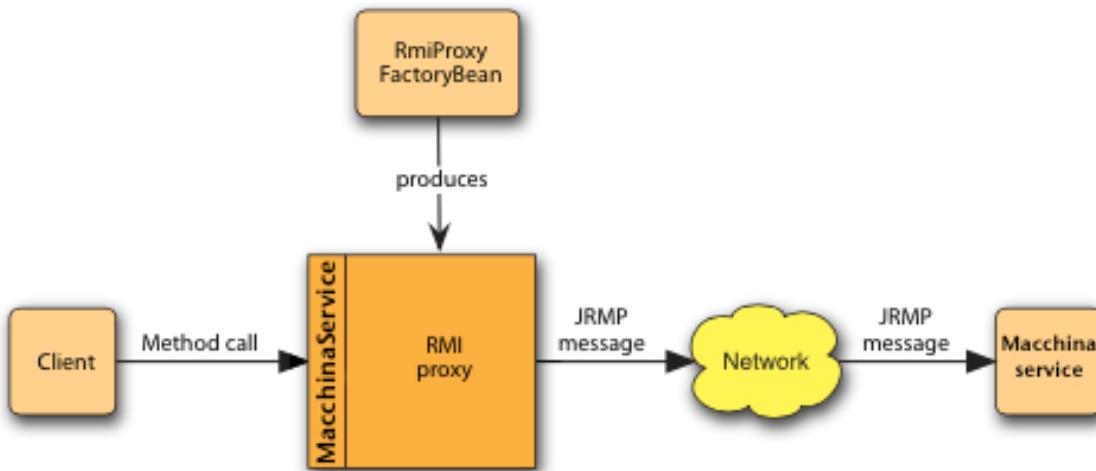
RmiServiceExporter (classe specificata nella prima riga di configurazione) esporta qualsiasi bean gestito da Spring come un servizio RMI. Come illustrato nella figura sopra, RmiServiceExporter per svolgere la sua funzione incapsula il bean in una classe che svolge il ruolo di adattatore. La classe adattatore viene quindi registrata nell'RMI Registry e sempre tale

classe rivolgerà le richieste che le arrivano al vero Servente (in questo caso MacchinaServant). Per quanto riguarda il registry RMI, la sua attivazione sarà svolta in automatico dal RmiServiceExporter.

Le righe successive della configurazione servono per indicare qual è la classe che implementa il Servente, qual è il nome del servizio remoto e la sua interfaccia. Può anche essere specificata una porta sulla quale esporre il servizio. Sarà compito di RmiServiceExporter effettuare il mapping tra la porta indicata nel file di configurazione e quella effettiva dell'RMI Registry. La realizzazione del **Client** è molto semplice.

```
<bean id="macchinaService" class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
    <property name="serviceUrl" value="rmi://localhost:2099/macchinaService"/>
    <property name="serviceInterface" value="it.car2go.service.MacchinaService"/>
</bean>
```

Le linee di codice di configurazioni appena mostrate sono sufficienti per effettuare il collegamento con un oggetto remoto.



RmiProxyFactoryBean è una factory il cui scopo è quello di creare un proxy ad un servizio RMI. Per la creazione di tale proxy è necessario specificare l'interfaccia e l'indirizzo del servizio. Il client può quindi inoltrare le richieste al proxy attraverso l'interfaccia del servizio senza sapere che in realtà il verso servente non è a lui locale.

CAPITOLO 11: MESSAGING

In questo capitolo sono illustrate le scelte intraprese per la realizzazione di una versione dell'applicazione che utilizza il paradigma di comunicazione asincrona basato su messaggi, code e topic.

11.1 Premessa

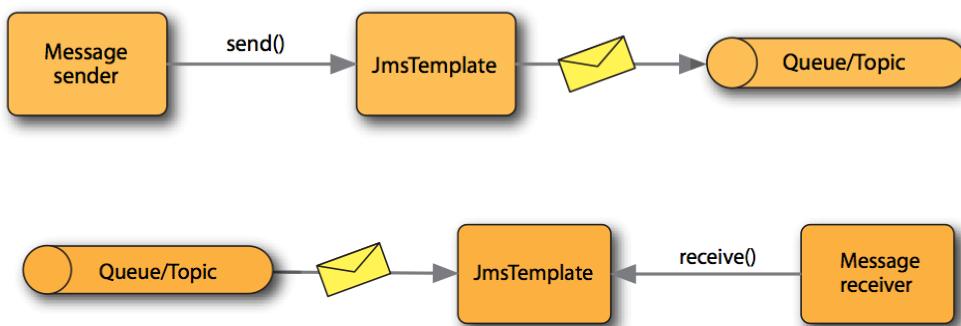
Molto spesso vi è l'esigenza di integrare componenti o servizi sviluppati indipendentemente. Tali componenti devono essere integrati per formare un sistema coerente ed interagire in modo affidabile mantenendo, però, basso l'accoppiamento. Per ottemperare a tali obiettivi, è stata prevista una versione del sistema in cui è presente lo scambio asincrono di messaggi tra componenti mediante un bus per messaggi.

11.2 Messaging

Il messaging è una tecnologia di comunicazione per sistemi distribuiti in cui i componenti o le applicazioni comunicano scambiandosi messaggi in una relazione peer-to-peer. Più nello specifico un componente nel ruolo di produttore può inviare un messaggio a un altro componente che riveste il ruolo di consumatore. Tali messaggi sono scambiati mediante un canale di comunicazione intermedio. Esistono due tipi di canali di comunicazione: le code ed i topic o argomenti. Le code sono un canale di comunicazione “a uno” intendendo che un messaggio inviato a tale destinazione intermedia sarà consumato da uno e un solo consumatore. Nei topic invece il messaggio è ricevuto da tutti i consumatori registrati presso la destinazione intermedia. Per tale motivo gli argomenti sono un canale di comunicazione “a molti”. Tra le caratteristiche più importanti del messaging e che la comunicazione, iniziata dal componente produttore del messaggio, è indiretta ed asincrona.

11.3 Spring JMS

Spring Framework rende più semplice lo sviluppo di applicazioni che necessitano di usare JMS. Per aiutare lo sviluppatore è stato creato un apposito modulo chiamato Spring JMS. Lo scopo di tale frammento dell'ecosistema Spring è quello di semplificare il lavoro del programmatore in fase di configurazione delle destinazioni intermedie ed evitare di riscrivere sempre le stesse porzioni di codice. Per ricevere o inviare messaggi è possibile utilizzare la classe JmsTemplate. Tale classe permette di tralasciare tutte le istruzioni che altrimenti sarebbero necessarie per effettuare la connessione a un provider JMS per pubblicare o ricevere messaggi.



Tutte le informazioni relative al provider che si intende usare per la gestione delle destinazioni intermedie, così come il loro nome, sono definite in un file di configurazione.

```
<!-- Destinazioni -->
<bean id="queue" class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="test.coda" />
</bean>
```

Il frammento appena inserito è necessario per definire una coda di nome “test.coda”.

11.4 Progetto Jms

Per mostrare l'uso di JMS è stato realizzato un semplice caso d'uso. Come spiegato nei capitoli precedenti, un utente che intende noleggiare una vettura, dopo averla prenotata e prima di mettersi alla guida, deve indicare, tra le altre cose, se la macchina presenta delle anomalie nella carrozzeria. Se l'utente segnala un qualche tipo di problema, l'amministratore potrà richiedere di riparare la macchina. Il caso d'uso realizzato va a completare ed ampliare tale funzionalità. Se l'utente segnala che lo stato esterno della macchina è danneggiato (selezionando la voce “non sufficiente” dal menù a tendina dalla pagina per effettuare la guida) la macchina invierà ad una coda il tipo di guasto che la affligge. Più in dettaglio è stato creato un elenco di guasti ed il componente che gestisce le macchine, automaticamente in maniera casuale, ne selezionerà uno e lo invierà in una coda. Il componente consumatore che è registrato presso la coda processerà tali messaggi salvandoli nel database. In questo modo quando l'amministratore vorrà far riparare le vetture, oltre all'indicazione delle auto che necessitano di essere riparate, avrà le informazioni sui malfunzionamenti che sono presenti in tale vettura. Per realizzare tale funzionalità sono quindi stati usati due tipi di componenti (oltre alla coda dove giungono i vari guasti). E' stato realizzato un semplice elemento sender, il cui scopo è quello di inviare i messaggi ad una specifica coda. I messaggi sono della forma “nomeGuasto,idMacchina”. Il ricevitore è un consumatore asincrono. Ogni qualvolta arriverà un messaggio alla coda, verrà richiamato dal container il metodo onMessage che ricaverà i dati necessari da rendere persistenti sul database. Per la gestione delle destinazioni intermedie è stato utilizzato Apache ActiveMQ che ben si integra con GlassFish e Spring.

CAPITOLO 12: ENTERPRISE BEAN

In questo capitolo sono illustrate le scelte effettuate per la realizzazione di una versione dell'applicazione basata su Enterprise Bean.

12.1 Premessa

Nell'attività di sviluppo di un'applicazione, oltre all'implementazione della logica di business, è importante prestare attenzione ad aspetti come la gestione delle transazioni, concorrenza, scalabilità, affidabilità, sicurezza, portabilità e riusabilità. Per potersi concentrare sullo sviluppo della logica di business senza tralasciare le qualità appena elencate, è possibile usare gli **Enterprise Bean**. Per tali motivi, nell'ambito del lavoro di tesi, è stata realizzata una versione dell'applicazione che fa uso degli Enterprise Bean.

12.2 Enterprise Bean

Gli Enterprise Bean sono componenti lato server, scritti in Java, che incapsulano la logica di business di una applicazione ed implementano lo standard Enterprise Java Bean (**EJB**). Per logica di business s'intende il codice realizzato per soddisfare gli scopi per i quali è stata creata l'applicazione. Gli EJB devono essere rilasciati all'interno di un EJB container che fornisce a tali componenti un certo numero di servizi come, per esempio, gestione della sicurezza, delle transazioni e supporto per i web-services.

Esistono diverse tipologie di Enterprise Bean:

- **Entity Bean:** il loro scopo è fornire le funzionalità di persistenza dei dati, inglobando oggetti lato server che memorizzano dati.

- **Session Bean:** encapsulano la logica applicativa e sono un'interfaccia tra i client e i servizi offerti dai componenti disponibili sul server.
- **Message Driven Bean:** permettono la ricezione e l'elaborazione di messaggi asincroni.

In particolare un Session Bean può essere di due tipi:

- **Stateful Session Bean**, in cui ciascuna istanza rappresenta lo stato di una sessione con un singolo client per tutta la durata della sessione.
- **Stateless Session Bean**, dove le istanze non mantengono informazioni sulle sessioni con i loro client.

12.3 Benefici nell'uso degli Enterprise Bean

Per vari motivi, gli Enterprise Bean semplificano lo sviluppo di grandi applicazioni distribuite. Il contenitore EJB, come già accennato, deve fornire dei servizi a livello di sistema per gli Enterprise Bean e lo sviluppatore si può quindi concentrare sulla risoluzione dei problemi di business.

In secondo luogo, poiché la logica di business è contenuta negli Enterprise Bean, gli sviluppatori dei client si possono focalizzare sugli aspetti di presentazione. Il client, libero dal dover implementare la logica di business oppure i metodi per accedere al database, sarà semplice e potrà andare in esecuzione anche su dispositivi di modesta potenza di elaborazione.

In terzo luogo, siccome gli enterprise bean sono componenti portabili, un'applicazione può essere assemblata a partire da bean esistenti. A patto di usare le API standard, queste applicazioni possono essere eseguite su qualsiasi server compatibile Java EE.

12.4 Quando usare gli Enterprise Beans

Si dovrebbe considerare l'utilizzo di Enterprise Bean se l'applicazione richiede uno dei seguenti requisiti.

- **L'applicazione deve essere scalabile.** Per soddisfare un numero crescente di utenti, è possibile distribuire i componenti dell'applicazione su più macchine. Non solo gli Enterprise Bean possono essere in esecuzione su macchine diverse, ma anche la loro posizione rimarrà trasparente ai clients.
- **Le operazioni devono garantire l'integrità dei dati.** Gli Enterprise Bean supportano le transazioni: meccanismi che gestiscono l'accesso simultaneo di oggetti condivisi.
- **L'applicazione può avere una varietà di clients.** Con solo poche righe di codice, client remoti possono facilmente individuare gli Enterprise Bean. Questi client possono essere semplici, numerosi e di varia natura.

12.5 Progetto EJB

Sono state create due versioni dell'applicazione basate su Enterprise Bean, entrambe rilasciate nell'application server JEE GlassFish.

Una prima versione è stata progettata creando:

- Un Session Bean di tipo stateful con nome Application EJB
- Un Session Bean di tipo stateless per ogni tipo di servizio
 1. MacchinaServiceEJB, implementa l'interfaccia MacchinaService
 2. UtenteServiceEJB, implementa l'interfaccia UtenteService
 3. RuoloServiceEJB, implementa l'interfaccia RuoloService
 4. PrenotazioneServiceEJB, implementa l'interfaccia PrenotazioneService
 5. GeoEJB, implementa l'interfaccia GeoService

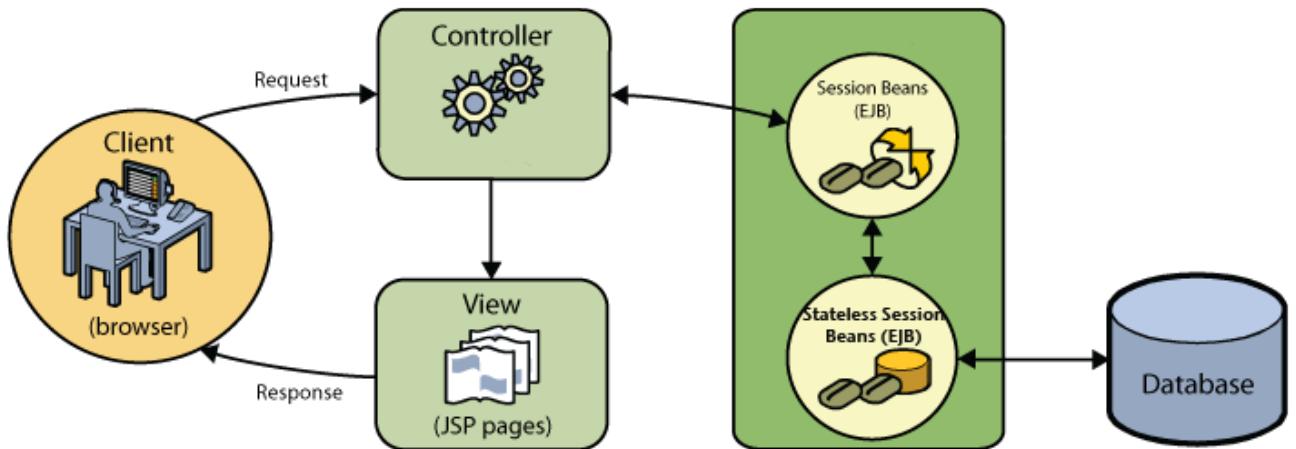
- Un Session Bean di tipo stateless per ogni tipo di dato per l'accesso al database
 1. MacchinaDAOEJB, implementa l'interfaccia MacchinaDAO
 2. UtenteDAOEJB, implementa l'interfaccia UtenteDAO
 3. RuoloDAOEJB, implementa l'interfaccia RuoloDAO
 4. PrenotazioneDAOEJB, implementa l'interfaccia PrenotazioneDAO

Una seconda versione è stata progettata usando direttamente:

- Un Session Bean di tipo stateless per ogni tipo di servizio
 1. MacchinaServiceEJB, implementa l'interfaccia MacchinaService
 2. UtenteServiceEJB, implementa l'interfaccia UtenteService
 3. RuoloServiceEJB, implementa l'interfaccia RuoloService
 4. PrenotazioneServiceEJB, implementa l'interfaccia
PrenotazioneService
 5. GeoEJB, implementa l'interfaccia GeoService
- Un Session Bean di tipo stateless per ogni tipo di dato per l'accesso al database
 1. MacchinaDAOEJB, implementa l'interfaccia MacchinaDAO
 2. UtenteDAOEJB, implementa l'interfaccia UtenteDAO
 3. RuoloDAOEJB, implementa l'interfaccia RuoloDAO
 4. PrenotazioneDAOEJB, implementa l'interfaccia PrenotazioneDAO

In ambedue le versioni dell'applicazione, ciascun Enterprise Bean implementa un'interfaccia remota (che costituisce l'interfaccia fornita) e può utilizzare, mediante l'iniezione delle dipendenze (grazie all'annotazione @EJB), altri Bean (ovvero l'interfaccia richiesta).

L'immagine seguente mostra come sono state strutturate, in linea generale, entrambe le versioni delle due applicazioni.



Dai Controller realizzati mediante l'approccio suggerito da Spring MVC, vengono richiamati gli EJB che contengono le funzionalità di business. Il compito di interfacciarsi con il database per recuperare i dati e gestire la persistenza è affidato ad Enterprise Bean di tipo Stateless.

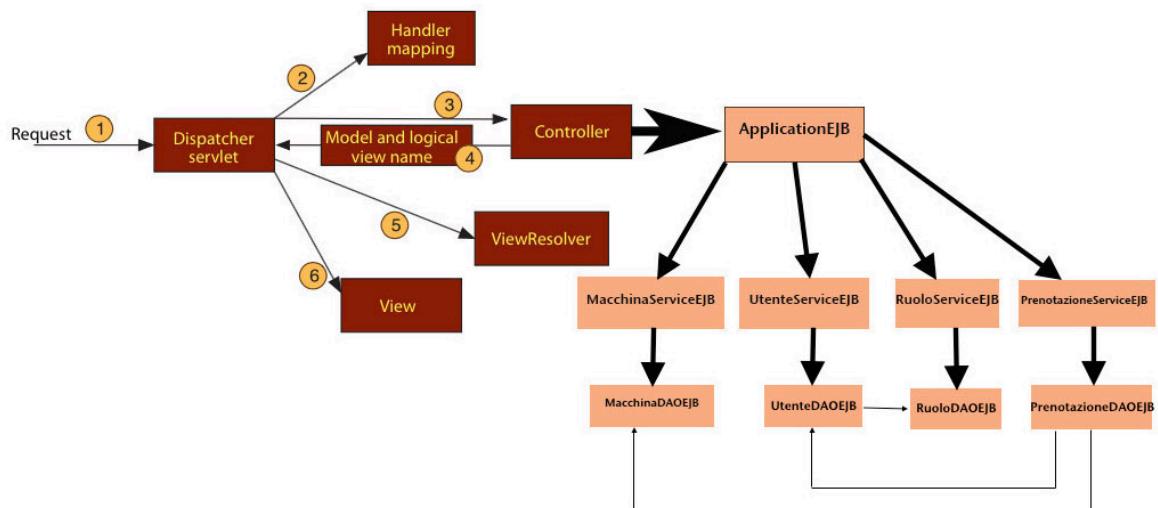
Data una visione d'insieme della struttura delle due applicazioni, è necessario analizzare nel dettaglio come sono state realizzate ambedue le versioni.

12.5.1 Prima versione (uso di Stateful Session Bean)

La prima versione dell'applicazione è caratterizzata dall'uso di tre strati di Enterprise Bean. Al primo livello c'è un Enterprise Bean di tipo

Stateful nominato ApplicationEJB. Questo bean ha il compito di ricevere le richieste provenienti dai Controller, gestire lo stato della sessione e utilizzare i bean di tipo Stateless (secondo livello) per poter completare le operazioni di sistema che gli sono state delegate dall'applicazione web. Infine al terzo livello troviamo gli EJB di tipo Stateless per gestire la persistenza dei dati.

La figura sottostante riassume le relazioni esistenti tra il Controller, l'ApplicationEJB e tutti gli altri Enterprise Bean di tipo Stateless.



L'ApplicationEJB implementa l'interfaccia remota ApplicationEJBRMote.

```

@Remote
public interface ApplicationEJBRemote {
    /*
     * Operazioni relative a Macchina
     */
    public Macchina getMacchina(int idMacchina);
    public List<Macchina> getMacchineLibere();
    public void creaMacchina(String tipo, int codice);
    public List<Macchina> getMacchineSenzaBenzina();
    public void effettuaRifornimento(int idMacchina);
    public void pulisciMacchina(int idMacchina);
    public void riparaMacchina(int idMacchina);
    public List<Macchina> elencoMacchineSporcheInterno();
    public List<Macchina> elencoMacchineDaRiparareEsterno();

    /*
     * Operazioni relative a Utente
     */
    public Utente getUtenteById(int idUtente);
    public boolean verificaPresenzaUtente(String username);
    public void salvaNuovoUtente(Utente utente);
    public Utente recuperaUtente(String username, String password);
    public int totaleNumPrenotazioni(int idUser);
    public int totaleTempo(int idUser);
    public int totaleDistanza(int idUser);
    public List<Utente> elencoUtentiSemplici(int idUtente);
    public void abilitaUtente(int idUtente);

    /*
     * Operazioni relative a Prenotazione
     */
    public int effettuaPrenotazione(int idUtente, int idMacchina);
    public void cancellaGuida(int idPrenotazione, int idMacchina);
    public void effettuaGuida(int idUtente, int idPrenotazione, int idMacchina, int interno, int esterno, String note);

    /*
     * Operazioni relative a Ruolo
     */
    public Ruolo recuperaRuolo(int idRuolo);
}

}

```

Per utilizzare i bean MacchinaServiceEJB, UtenteServiceEJB, RuoloServiceEJB e PrenotazioneServiceEJB viene sfruttata l'iniezione delle dipendenze.

```

/**
 * Session Bean implementation class ApplicationEJB
 */
@Stateful(mappedName = "ApplicationEJB")
@Remote
public class ApplicationEJB implements ApplicationEJBRemote {

    @EJB(lookup="MacchinaServiceEJB")
    private MacchinaService macchinaService;

    @EJB(lookup="UtenteServiceEJB")
    private UtenteService utenteService;

    @EJB(lookup="RuoloServiceEJB")
    private RuoloService ruoloService;

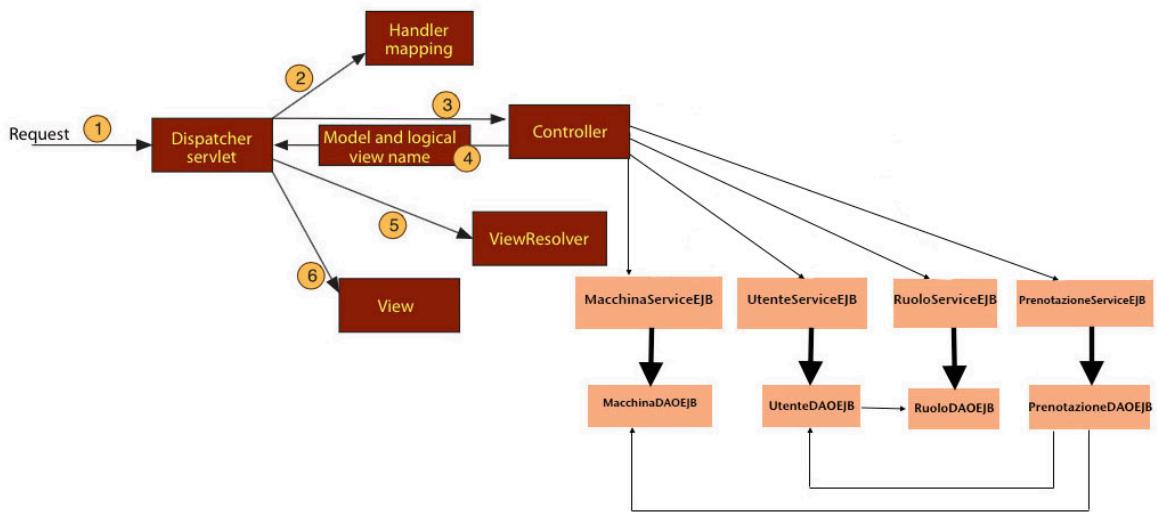
    @EJB(lookup="PrenotazioneServiceEJB")
    private PrenotazioneService prenotazioneService;

    @EJB(lookup="GeoEJB")
    private GeoService geoService;
}

```

12.5.2 Seconda versione (senza uso di Stateful Session Bean)

Nella seconda versione dell'applicazione non è stato fatto uso del Bean Stateful di nome ApplicationEJB. I Controller dell'applicazione web utilizzano direttamente i Bean Stateless che implementano la logica di business come MacchinaServiceEJB, UtenteServiceEJB, RuoloServiceEJB e PrenotazioneServiceEJB.



Sarà l'applicazione web a mantenere le informazioni di sessione di un utente.

12.6 Integrazione Spring MVC con EJB

Nei capitoli precedenti è stato ampiamente spiegato come una delle caratteristiche più importanti di Spring Framework sia l'integrabilità. Spring è facilmente integrabile nei diversi livelli di un'applicazione, dal frontend al backend, dimostrandosi utilissimo anche nello strato applicativo nell'implementazione della logica di business. L'intenzione di questo paragrafo è evidenziare come sia semplice integrare Spring all'interno di una applicazione distribuita che fa uso di EJB3 e come facilmente i servizi

da essi esposti possano essere iniettati nel contesto applicativo ed utilizzati all'interno di un Controller. Il primo passo da compiere è quello di settare all'interno del file di configurazione di Spring le linee di codice necessarie per recuperare i Session Bean. Per tale scopo è possibile usare il tag <jee:jndi-lookup> con l'attributo jndi-name. Tale tag indica di effettuare una ricerca JNDI andando a rintracciare un bean il cui nome è specificato nell'attributo jndi-name. JNDI è gestito dall'application server in cui sono rilasciati gli EJB. Nell'immagine seguente è mostrato il frammento del file di configurazione che permette di collegare gli EJB all'interno dell'applicazione web gestita da Spring.

```
<jee:jndi-lookup id="macchinaService" jndi-name="MacchinaServiceEJB">
    <jee:environment>
        org.omg.CORBA.ORBInitialHost = localhost
    </jee:environment>
</jee:jndi-lookup>

<jee:jndi-lookup id="utenteService" jndi-name="UtenteServiceEJB">
    <jee:environment>
        org.omg.CORBA.ORBInitialHost = localhost
    </jee:environment>
</jee:jndi-lookup>

<jee:jndi-lookup id="ruoloService" jndi-name="RuoloServiceEJB">
    <jee:environment>
        org.omg.CORBA.ORBInitialHost = localhost
    </jee:environment>
</jee:jndi-lookup>

<jee:jndi-lookup id="prenotazioneService" jndi-name="PrenotazioneServiceEJB">
    <jee:environment>
        org.omg.CORBA.ORBInitialHost = localhost
    </jee:environment>
</jee:jndi-lookup>
```

Senza l'utilizzo di Spring, all'interno di un client per ottenere il riferimento ad un EJB, è necessario scrivere delle righe di codice simili alle seguenti.

```

InitialContext ctx = null;
try{
    ctx = new InitialContext();
    MacchinaService macchinaService = (MacchinaService) ctx.lookup("MacchinaServiceEJB");
} catch(NamingException ne){
    //gestione NamingException
} finally{
    if(ctx != null){
        try{
            ctx.close();
        } catch (NamingException ne) {}
    }
}

```

Dall'analisi del listato precedente si nota come con Spring il codice per l'integrazione sia più semplice.

Per effettuare l'iniezione delle dipendenze di un bean all'interno del controller è sufficiente usare l'annotazione `@Autowired`.

```

@.Autowired
private UtenteService utenteService;
@Autowired
private RuoloService ruoloService;
@Autowired
private MacchinaService macchinaService;
@Autowired
private PrenotazioneService prenotazioneService;

```

CAPITOLO 13: WEB SERVICE

Nel seguente capitolo sono illustrate le scelte effettuate per la realizzazione di una versione dell'applicazione mediante Web Service di tipo Soap e Rest.

13.1 Premessa

Sempre più spesso un requisito importante nella realizzazione di un'applicazione è la possibilità di supportare l'interoperabilità tra componenti in esecuzione su piattaforme diverse sulla base di protocolli standard aperti ed universalmente accettati. Per rispondere a tale vincolo, nell'ambito del lavoro di tesi, è stata realizzata una versione dell'applicazione che faccia uso dei Web Service.

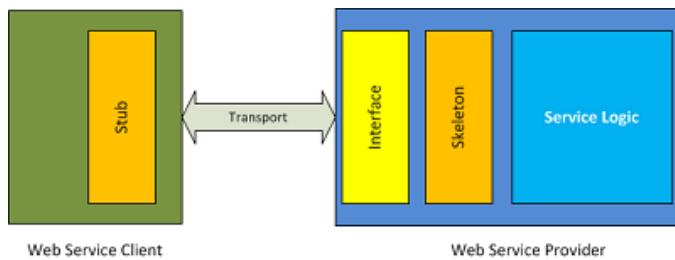
13.2 Web Service

Un Web Service, secondo la definizione data da Papazoglou, il cui scopo principale è quello di risolvere le problematiche legate all'interoperabilità tra processi di business spesso eseguiti da organizzazioni indipendenti con differenti piattaforme e linguaggi di programmazione, è:

- *un modulo o componente software, auto-contenuto e auto-descrittivo, accessibile mediante internet in modo indipendente dalla piattaforma*
- *ha lo scopo di svolgere un compito, risolvere un problema o condurre transazioni per conto di un utente o di un'applicazione – ovvero, di incapsulare un “servizio”.*

Nell’ambito del lavoro di tesi sono stati utilizzati sia i Web Service Soap sia Rest. In particolare sono state realizzate completamente tutte le funzionalità dell’applicazione mediante Web Service di tipo Soap. I Web Service di tipo Rest sono stati usati per recuperare da Google Maps informazioni geografiche e utili a contabilizzare il costo di un noleggio. Per mostrare che i Web Service sono indispensabili per far colloquiare client diversi è stato realizzato un caso d’uso in cui da un client Swing si richiedono informazioni sulle macchine presenti nel sistema.

13.3 Web Service Soap



Per realizzare una versione dell’applicazione basata su web service si è scelto un approccio di tipo “bottom-up” o “implementation first”. Nessun sviluppatore dovendo realizzare un Web Service, inizierebbe a codificare il WSDL del contratto, in quanto troverebbe più naturale la scrittura del codice di implementazione. Tutti gli ambienti di sviluppo, inoltre, dispongono di strumenti necessari alla pubblicazione di un Web Service a partire dalla logica applicativa, sia essa preesistente o di nuova realizzazione. Anche se, a rigore, un simile approccio va a contraddirsi il modello di progettazione orientato agli oggetti, secondo il quale la definizione delle interfacce è propedeutica all’implementazione della logica applicativa, questa tecnica permette di pubblicare agevolmente un Web Service. Il programmatore quindi non si deve confrontare con la complessa

realizzazione del WSDL con il quale gli ambienti di sviluppo sono in grado di derivare le classi di stub, endpoint e di trasformazione XML-Java. Per la realizzazione di tale Web Service è stata usata l'API JAX-WS, acronimo di Java API for XML Web Services, tecnologia dedicata esclusivamente allo sviluppo di Web Service in Java che comunicano utilizzando messaggi SOAP/XML. Usando l'API JAX-WS, la definizione del servizio è semplicemente una classe con alcune annotazioni. Tali annotazioni forniscono pertanto dei metadati sulla classe e sui suoi metodi per precisare il comportamento del web service. L'unica annotazione obbligatoria è `@WebService`, che indica che una classe implementa un web service.

Per la costruzione di tale Web Service è stata utilizzata una web application di nome car2go-soapws. Come endpoint sono stati esposti le classi RuoloServant, UtenteServant, MacchinaServant e PrenotazioneServant che forniscono dei servizi per ogni tipo di entità dell'applicazione. In tali classi sono state usate le annotazioni `@WebService` e `@WebMethod` per indicare l'esposizione di tali oggetti come servizi remoti. Tali classi, per risolvere i loro compiti di business, devono interagire con il database e pertanto necessitano di elementi Dao. Per effettuare in modo corretto l'iniezione delle dipendenze è stata creata una classe Factory, che implementa il pattern Singleton, con lo scopo di indicare alle classi esposte come endpoint i riferimenti ai Dao. La Factory ha anche il compito di iniettare ai vari Dao il riferimento a un oggetto di tipo Connection per l'interazione con il database.

Terminato di scrivere il codice lato server, per generare il WSDL è stato sufficiente utilizzare l'implementazione di JAX-WS offerta da Glassfish.

Lato client, per generare le risorse necessarie per interagire con il WebService, è stato utilizzato **wsimport**, un tool offerto da JEE. La sua

funzione è quella di creare tutte le risorse definite in un file wsdl passato come parametro.

La sintassi da utilizzare è la seguente:

```
wsimport [opzioni] <url file wsdl>
```

le principali opzioni da tener presente sono:

- **-d <directory>** rappresenta la cartella dove verranno salvati i file di output (.class) generati.
- **-help** mostra la lista completa delle opzioni utilizzabili e la loro descrizione.
- **-p <pkg>** indica il package delle classi prodotte.
- **-s <directory>** indica la cartella dove salvare I file sorgenti (.java) generate.
- **-target <version>** indica la versione dello standard da utilizzare: 2.0 per la specifica JAXWS 2.0.

13.4 Web Service Rest

I Web Service di tipo Representational State Transfer (detti anche REST o “lightweight”) sono un’alternativa ai WS SOAP. I Web Service Rest sostengono l’interoperabilità e sono più efficienti rispetto a quelli di tipo SOAP infatti si basano sul protocollo http e richiedono una infrastruttura più leggera. REST (Representational State Transfer) è uno stile architettonico per sistemi software distribuiti. Il termine è stato introdotto e definito nel 2000 da Roy Fielding, uno dei principali autori delle specifiche del protocollo http ed indica una serie di principi architettonici per la progettazione di Web Service. Il concetto alla base dello stile REST è quello di “risorsa”, univocamente identificabile mediante un URI (Uniform Resource Identifier). Una risorsa è una qualunque entità che possa essere indirizzabile tramite Web, cioè accessibile e trasferibile tra client e server.

Spesso rappresenta un oggetto appartenente al dominio del problema che si sta affrontando. Le risorse possono essere accedute, create, modificate oppure cancellate mediante messaggi autodescrittivi che invocano operazioni definite in corrispondenza con le operazioni tipiche di HTTP (GET, PUT, POST, DELETE). I Web Services di tipo REST prevedono la possibilità di scegliere il formato di interscambio più opportuno tra XML, HTML o JSON. Affinché un Web Service sia conforme alle specifiche REST deve avere alcune caratteristiche ben precise:

- architettura basata su client e server
- stateless. Ogni ciclo di request/response deve rappresentare un'interazione completa del client con il server. In questo modo non è necessario mantenere informazioni sulla sessione utente, minimizzando l'uso di memoria del server e la sua complessità.
- Uniformemente accessibile. Ogni risorsa deve avere un indirizzo univoco e ogni risorsa di ogni sistema presenta la stessa interfaccia, precisamente quella individuata dal protocollo http.

Per la realizzazione del Web Service Rest è stato utilizzato Jersey. Jersey è l'implementazione di riferimento della specifica JAX-RS (JSR 311) per la realizzazione di Web Service RESTful su piattaforma Java. Jersey permette di creare risorse semplicemente così come si sviluppano POJO (Plain Old Java Object) utilizzando delle specifiche annotazioni per i metodi e le classi. In altre parole il framework si occupa di gestire le richieste http e la negoziazione della rappresentazione e permette di concentrarsi sulla soluzione del problema. Per utilizzarlo è sufficiente che siano disponibili nel classpath le librerie jersey-core-1.18.jar, jersey-server-1.18.jar, jsr311-api.jar e asm.jar.

Il caso d'uso realizzato con i Web Service Rest ha riguardato l'esposizione della collezione di macchine presenti nel sistema. Come primo passo, per ogni risorsa è stata definita un URI di base, successivamente è stata creata

la classe esposta come endpoint che ha utilizzato annotazioni del tipo @Path (per specifica l'URI), @GET (per restituire un elemento o una collezione), @POST (per inserire un nuovo elemento nella collezione), @DELETE (per cancellare un elemento), @PUT per aggiornare una risorsa. Il seguente frammento di codice mostra la funzione per recuperare tutte le macchine non prenotate.

```
/*
 * Restituisce tutte le macchine non prenotate
 */
@Path("/allMacchineNonPrenotate")
@GET
@Produces(MediaType.APPLICATION_JSON)
public List<Macchina> getMacchineNonPrenotate() {
    List<Macchina> listaInput = macchinaDao.getMacchine();
    List<Macchina> listaOutput = new ArrayList<Macchina>();

    Iterator iterator = listaInput.iterator();

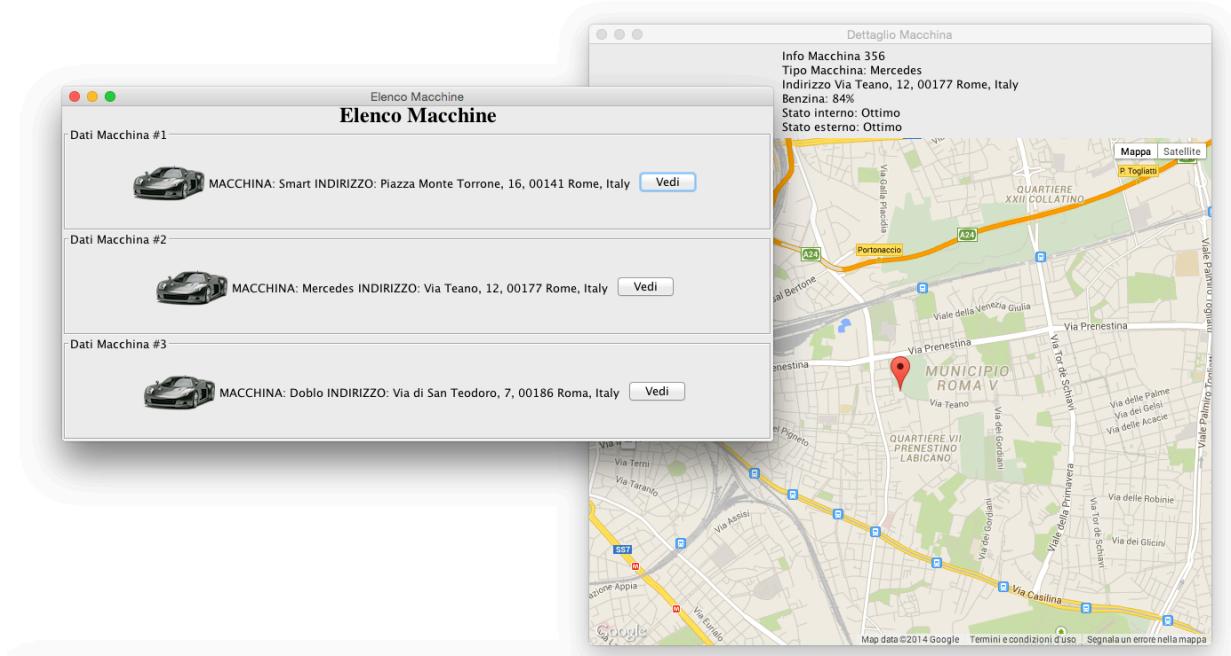
    while(iterator.hasNext()){
        Macchina m = (Macchina) iterator.next();
        if(m.getPrenotata() == 0){
            listaOutput.add(m);
        }
    }
    return listaOutput;
}
```

Come nel caso del WebService Soap, è stata creata una classe Factory, che implementa il pattern Singleton, con lo scopo di indicare alle classi esposte come endpoint i riferimenti ai Dao.

Per quanto concerne il client che ha sfruttato i servizi messi a disposizione da Google Maps, le risposte ottenute mediante richieste http sono state interpretate facendo uso della libreria JSON Simple.

```
public int getNumeroMacchine(){
    Client client = Client.create();
    WebResource webResource = client.resource("http://localhost:8080/car2go-restws/api/macchina/totaleMacchineNonPrenotate");
    ClientResponse resp = webResource.accept(MediaType.APPLICATION_JSON).get(ClientResponse.class);
    if (resp.getStatus() != 200) {
        throw new RuntimeException("Failed : HTTP error code : "
                + resp.getStatus());
    }
    String output = resp.getEntity(String.class);
    //metto nella variabile tot il totale delle macchine
    int tot = Integer.parseInt(output);
    return tot;
}
```

Il frammento di codice appena mostrato indica come recuperare il numero di auto non prenotate. Le immagini sottostanti mostrano il client Swing che usa il WebService Rest in azione.



Per integrare una mappa di Google all'interno di un frame Swing è stata utilizzata un'apposita libreria.

CAPITOLO 14: CONCLUSIONI E SVILUPPI FUTURI

L’obiettivo della tesi è stato quello di realizzare un’infrastruttura di supporto alla sperimentazione dei concetti affrontati in un corso di Architetture Software. Per raggiungere tale obiettivo sono stati identificati diversi strumenti e tecnologie per permettere ad uno studente di applicare in maniera pratica quanto appreso a livello teorico.

A tal proposito una prima fase del lavoro di tesi ha previsto un’attenta analisi di quelli che potevano essere gli strumenti e le tecnologie che ben si prestavano ad un uso congiunto con il corso di Architetture Software. Nella scelta di tali strumenti le linee seguite sono state le seguenti: facilità di utilizzo, bassa curva di apprendimento per non aggravare ulteriormente la quantità di studio richiesta ai discenti e la possibilità di fornire conoscenze spendibili nel mondo lavorativo. Diverse sono state le idee proposte come Vagrant, Spring, Maven e GitHub.

Con Vagrant è stato possibile effettuare, con estrema facilità, il provisioning di ambienti virtuali. L’uso di tale strumento è stato pensato anche con un’ottica rivolta verso il cloud in quanto Vagrant ben si integra con i servizi di Amazon.

Maven è uno strumento che permette la gestione efficace di grandi progetti in Java mentre Spring è un noto framework largamente utilizzato dalla comunità degli sviluppatori. Visto il suo largo utilizzo, ne è stata effettuata un’attenta analisi architetturale dove sono stati messi in evidenza i fattori che ne hanno determinato il successo: facilmente integrabile, testabile, configurabile e intuitivo.

GitHub è una piattaforma di social-coding che permette, in modo efficace, la collaborazione tra membri di un team di sviluppo.

Tali strumenti sono stati sperimentati direttamente mediante la realizzazione di diverse versioni di un'applicazione per la gestione di un servizio di car-sharing. Nella fase iniziale e durante tutto lo sviluppo delle varie versioni dell'applicazione è stato seguito il pattern architetturale “Ports and Adapters” che ha permesso la realizzazione di un sistema modulare in cui è possibile utilizzare servizi/funzionalità di natura diversa senza dover cambiare il resto del sistema. L'applicazione è stata quindi realizzata prima come un semplice sito web. Nelle fasi successive l'interfaccia web è rimasta inalterata mentre per la logica di business si è provveduto ad implementarla mediante RMI, EJB e Web Service. Un particolare caso d'uso è stato realizzato mediante JMS così come è stato implementato un client Swing che effettua delle richieste mediante un Web Service di tipo Rest. Per concludere gli aspetti implementativi, è stata realizzata una versione dell'applicazione in cui tutte le tecnologie sono state fuse insieme.

Nonostante il lavoro svolto per la realizzazione della tesi sembra piuttosto lineare, molti sono stati i problemi incontrati. La maggior parte di essi sono stati causati dalla difficoltà intrinseca di configurazione di ambienti virtuali e, in alcuni casi, nella mancanza di documentazione. Per risolvere tali ostacoli, si è fatto ricorso a materiale consultabile in rete. Emblematico il caso di configurazione di un ambiente virtuale con Glassfish in Vagrant dove è stato necessario ricorrere all'ausilio dell'ideatore di tale sistema per un bug trovato proprio nel corso del lavoro di tesi.

In conclusione è possibile affermare che l'utilizzo degli strumenti e delle tecnologie proposte ha effetti positivi. Con Vagrant infatti il docente può configurare alcune macchine virtuali da mettere a disposizione degli studenti in modo che questi possano lavorare tutti su un medesimo ambiente di produzione. In tale modo i discenti potranno confrontarsi solo

su aspetti relativi alla difficoltà del problema senza perdersi in dettagli relativi al provisioning dell’ambiente in cui è richiesto loro di lavorare. GitHub stimola e facilita la collaborazione tra studenti e, come ricorda Vygotskij, questo ha effetti benefici sulle capacità di apprendimento di un discente. Maven permette di gestire, mediante un approccio dichiarativo, tutte le librerie di un progetto facilitando uno sviluppatore nelle operazioni di configurazione ed è comunemente usato in ambito lavorativo. Spring, oltre che rappresentare una conoscenza fondamentale per uno sviluppatore Java, rappresenta un’alternativa alla piattaforma Java Enterprise Edition e può essere interessante da mostrare nel corso di Architetture Software.

Per tutti i motivi sopra descritti è possibile affermare che l’obiettivo della tesi è stato raggiunto perché realizzare un’applicazione funzionante stimola l’interesse di uno studente verso la materia. Il discente con l’applicazione pratica delle nozioni teoriche può apprendere meglio riuscendo anche ad acquisire la conoscenza di strumenti diffusi in ambito produttivo.

Un possibile sviluppo futuro per tale lavoro potrebbe essere l’utilizzo di **Vert.x**. Tale strumento è una piattaforma per costruire applicazioni scalabili e real-time sulla Java Virtual Machine. Attraverso un Event Bus distribuito che permette l’interazione asincrona di diversi componenti e un sistema modulare, è possibile realizzare applicazioni con diversi linguaggi come JavaScript, Java, Ruby, Groovy, Python e altri non supportati ufficialmente. Vert.x fornisce inoltre il supporto a tecnologie come WebSocket e SockJS per realizzare applicazioni real-time con notifiche push lato server. Ogni componente che viene rilasciato su Vert.x è un Verticle (vertice), ovvero una singola unità che permette di soddisfare una particolare richiesta e che può dialogare con altre componenti attraverso l’Event Bus. [VERTX]

Vert.x è una piattaforma davvero interessante, sia per il paradigma architettonico che per le feature esposte. Una delle sue maggiori potenzialità risiede nel fatto di poter realizzare componenti disaccoppiati, che comunicano attraverso un canale asincrono e soprattutto che possono essere implementati in differenti linguaggi. Se si ha la necessità di utilizzare Ruby per un componente, magari perché solo in quel linguaggio si dispone di una particolare libreria, la restante parte dell'applicazione potrebbe essere tranquillamente realizzata in un altro linguaggio senza doversi preoccupare di interazioni specifiche. Anche la possibilità di realizzare applicazioni con WebSocket o SockJS permette di affrontare una serie di scenari che risultano molto più complicati con framework di vecchia generazione. Infine è doveroso specificare che Vert.x è altamente scalabile e gestisce al suo interno rinomate piattaforme come Hazelcast per la distribuzione su cluster e Netty per la gestione del I/O.

BIBLIOGRAFIA

[GOO]

Titolo: Growing Object-Oriented Software, Guided by Test

Autori: Steve Freeman, Nat Pryce

Casa Editrice: Addison-Wesley

Edizione: 2010

[IDDD]

Titolo: Implementing Domain-Driven Design

Autore: Vaughn Vernon

Casa Editrice: Addison-Wesley

Edizione: 2013

[JSKM]

Titolo: Just Spring

Autore: Konda Madhusudhan

Casa Editrice: O'Reilly

Edizione: 2011

[SIAW]

Titolo: Spring in Action

Autore: Carig Walls

Casa Editrice: Manning

Edizione: 2011

[SIIA]

Titolo: Spring Integration in Action

Autori: Fisher, Partner, Bogoevici, Fuld

Casa Editrice: Manning

Edizione: 2012

[BHLM]

Titolo: Beginning Hibernate

Autori: Jeff Linwood, Dave Minter

Casa Editrice: Apress

Edizione: 2010

[PISP]

Titolo: Principi di Ingegneria del software

Autore: Roger S. Pressman

Casa Editrice: McGraw-Hill

Edizione: Quinta edizione – 2005

[SCMB]

Titolo: Software Configuration Management

Autore: W.A. Babich

Casa Editrice: Addison-Wesley

Edizione: 1986

Siti Web

[VOGL]

Sito web: <http://www.vogella.com>

Data ultima consultazione: Novembre 2014

[OJEE]

Sito web: <http://docs.oracle.com>

Data ultima consultazione: Dicembre 2014

[GITH]

Sito web: <http://www.github.com>

Data ultima consultazione: Dicembre 2014

[MOKB]

Sito web: <http://www2.mokabyte.it>

Data ultima consultazione: Dicembre 2014

[VERTX]

Sito web: <http://vertx.io>

Data ultima consultazione: Dicembre 2014

[SPRING]

Sito web: <http://www.spring.io>

Data ultima consultazione: Novembre 2014

[MKYNG]

Sito web: <http://www.mkyong.com>

Data ultima consultazione: Novembre 2014

[BRJB]

Sito web: <http://www.briansjavablog.blogspot.it>

Data ultima consultazione: Ottobre 2014

[MATV]

Sito web: <http://matteo.vaccari.name>

Data ultima consultazione: Ottobre 2014

Articoli Scientifici

Titolo: Hexagonal architecture

Autore: Alistar Cockburn

<http://alistair.cockburn.us/Hexagonal+architecture>

Guide Ufficiali

Apache Maven

Vagrant

Hibernate

Apache ActiveMQ

Java EE 6 Tutorial