Università degli Studi di Padova

Applicazioni web in Groovy e Java

Laurea Magistrale

Laureando: Davide Costa
Relatore: Prof. Giorgio Clemente

Dipartimento di Ingegneria dell'Informazione
Anno Accademico 2011-2012

Indice

Introduzione

Le nuove tecnologie portano sempre più all'uso dei browser web come strumento di lavoro per interagire con le applicazioni (magari non in modo esplicito come ad esempio in un Ipad). Oltre al trend natural dovuto principalmente allo sviluppo di internet, si può osservare una notevole spinta commerciale da parte di alcune compagnie interessate (giganti come Google o Apple), che hanno investito notevolmente sullo sviluppo dei servizi sul web. Inoltre il browser è divenuto lo strumento più usato a livello di popolazione mondiale e negli anni è notevolmente aumentata la facilità d'uso. I browser web sono diventati di supporto in moltissimi campi, e negli ultimi anni si ha avuto un'esplosione nello sviluppo di applicazioni web che hanno letteralmente invaso la rete internet.

In particolare una tendenza che è emersa di recente è quella di remotizzare, ovvero unificare l'accesso alle informazioni contenute in un server sul luogo di lavoro o altra locazione.

Da queste riflessioni nasce questo lavoro, in cui si intende esplorare ed approfondire lo sviluppo di applicazioni web con un linguaggio di programmazione dinamico, in opposizione ai linguaggi statici; si individuerà un problema e si realizzerà un prototipo di applicazione web per sperimentare alcuni aspetti delle tecnologie scelte, per verificare se con esse sia possibile risolvere il problema, quali operazioni siano necessarie e quali possibili. Nel lavoro verrà quindi inizialmente affrontato il problema di individuare le tecnologie sperimentali, andando a scegliere linguaggio di programmazione, IDE di sviluppo, framework ecc….

Tutti i linguaggi di programmazione che prevedono un funzionamento remoto presentano limitazioni nelle operazioni, dovute principalmente a motivi di sicurezza. Molte di queste limitazioni sembrano rilassate in Java, o meglio nei linguaggi che utilizzano la Java Virtual Machine, uno di questi (il migliore disponibile al momento) è Groovy.

Groovy è linguaggio dinamico, moderno, compatto ed espressivo; oltre ad essere Java-like, è perfettamente compatibile con Java a livello sorgente (le istruzioni Java sono anche istruzioni Groovy)ed a livello ldi librerie, consentendo così di utilizzare

l'enorme patrimonio del codice sviluppato in Java. Groovy verrà trattato più approfonditamente nel capitolo 1.

L'elaborato è suddiviso in parti principali:

a) introduzione al linguaggio Groovy e all'ambiente di sviluppo integrato in NetBeans
b) architettura del framework e scelta dello stesso fra Google App Engine, Play! e Griffon
c) sviluppo di un'applicazione di prova per verificare le principali ipotesi di progetto

I primi capitoli della tesi hanno quindi lo scopo di introdurre gli strumenti scelti per sviluppare un'applicazione completa. In particolare all'inizio verrà analizzato il linguaggio Groovy con le sue caratteristiche e i suoi pregi; e quindi presentato NetBeans, ovvero il principale IDE di sviluppo.

Nella seconda parte sono presentati alcuni framework che utilizzano Groovy come principale linguaggio di supporto o che sono stati sviluppati tramite Groovy stesso. Per ciascuno di essi sono sviluppati alcuni esempi applicativi e se ne valutano i pro e i contro.

Al termine della valutazione la scelta del framework è caduta su Griffon, che è apparso il più allineato agli obiettini di integrazione che ci si era proposti, obiettivi in gran parte dichiarati e condivisi dai progettisti del framework.

L'ultima parte è dedicata allo sviluppo di un prototipo di un'applicazione web di complessità non banale, che si presta ad essere ripartita in vari modi tra server e client in base alle esigenze di utilizzo.

L'applicazione si propone di archiviare, catalogare, ricercare, modificare e presentare in maniera coerente documenti multipli (costituiti cioè da più sezioni con contenuti di tipo diverso). In altre parole si vuole individuare una soluzione per realizzare applicazioni per la visione unificata via browser di oggetti, integrando e unificando gli strumenti tramite un linguaggio di scripting , indipendentemente dalla localizzazione delle informazioni. La soluzione proposta è stata realizzata in un prototipo funzionante.

Per sperimentare l'applicazione in ambito reale si supponga che gli oggetti da gestire siano i documenti didattici forniti a supporto di un corso universitario (ad esempio del corso di Sistemi Operativi, in cui sono forniti i temi di esame, integrati con un'analisi del problema ed una proposta di soluzione). Si può ipotizzare che ogni documento sia suddiviso in sezioni distinte:

1) identificazione, contenente autore,data, titolo (non modificabile)
2) tema d'esame proposto nel documento (non modificabile)
3) soluzione del problema, ovvero il codice per i temi di programmazione (modificabile dall'utente che usa l'applicazione)

4) commenti alla soluzione (modificabile)
5) keyword per la classificazione

Per ridurre la quantità dei file in gioco, l'occupazione di memoria secondaria, diminuire gli eventuali tempi di trasmissione e nascondere l'organizzazione interna di tali documenti si ipotizza di unire tutte queste sezioni in un unico archivio compresso nel formato zip.

Occorre presentare all'utente quindi un'interfaccia grafica che permetta di aprire questi documenti, visualizzarli ed eventualmente modificarli.

Ma non è finita qui, per organizzare i documenti (contenuti dell'archivio), individuarli nel caso l'utente non li conoscesse o non li ricordasse, è previsto l'uso di un database di supporto, contenente per ogni documento alcuni campi che rappresentano il contenuto dello stesso, si stabilisce che i campi del database siano il nome del documento, alcune parole chiave e l'argomento trattato nello stesso.

In definitiva l'applicazione web offrirà, tramite interfaccia grafica unificata, gli strumenti per ricercare un documento interrogando un database, scegliere tra i possibili risultati, visualizzare ed eventualmente modificare un documento strutturato realizzato in parti diverse e contenuto in un archivio compresso.

1 Groovy

Questo capitolo presenta le principali caratteristiche di Groovy e le sue differenze con Java; sono illustrati i concetti principali del linguaggio stesso ed evidenziati gli aspetti che rendono Groovy un linguaggio "superiore" a Java; si discute del perché si integra con Java in maniera cosi eccellente e viene descritto il meccanismo di scripting. È riportato totalmente il changelog per dell'ultima versione stabile (1.8) rilasciata per allertare il lettore che può possedere vecchie versioni sui cambiamenti recenti. etc

Si procede quindi con una spiegazione su come installare Groovy, nelle varie piattaforme (Windows, Linux, Mac) attraverso differenti pacchetti di installazione, pre-costruiti e di uso semplice, o da codice sorgente in modo più complicato ma maggiormente personalizzabile.

Viene introdotto l'ambiente di programmazione e la shell, con i comandi principali per interagire con Groovy e la groovyConsole che fornisce un'interfaccia grafica interattiva.

Il capitolo si conclude con alcuni semplici esempi che chiariscono i concetti espressi nell'introduzione e mostreranno di cosa è capace Groovy.

1.1 Introduzione a Groovy

Groovy è un linguaggio di programmazione orientato agli oggetti per la piattaforma Java (ovvero che gira sulla piattaforma Java ed è perfettamente integrato in essa), oltretutto ha una sintassi Java-like e consente anche lo scripting. Si tratta di un linguaggio dinamico con caratteristiche simili a quelle di Python, Ruby, Perl, e Smalltalk (ma che a scelta del programmatore può diventare per alcune cose statico come Java: ad esempio nella definizione delle variabili, si può scegliere se dichiarare il tipo della variabile o semplicemente dire che esiste ma non si sa o non si vuole comunicare al programma stesso il tipo della variabile).

Fig.1 Il logo di Groovy

Groovy usa una sintassi Java-like. E' compilato dinamicamente in bytecode per la Java Virtual Machine (JVM) e interagisce con codice e librerie Java.La maggior parte del codice Java è anche sintatticamente valido in Groovy.Anche se Java e Groovy sono simili, il codice in Groovy può essere più compatto, perché non richiede tutti gli elementi richiesti da Java. Questo rende possibile per i programmatori Java imparare gradualmente Groovy iniziando con familiarità dalla sintassi Java prima di acquisire nozioni più avanzate.

La prima versione del linguaggio, la 1.0, è stata rilasciata da James Strachan il 2 gennaio 2007. Da quel momento sono state rilasciate varie versioni, fino all'attuale 1.8, ultima versione stabile scaricabile dal sito del progetto. Groovy, naturalmente, può essere installato su qualsiasi sistema con supporto la Java Virtual Machine.

Le caratteristiche principali di Groovy sono le seguenti:

- consente la modifica a runtime di classi e metodi, ad esempio definendo un nuovo metodo di classe;
- le istruzioni sono separate dal carattere di nuova linea, non è necessario inserire il carattere ';' alla fine della linea;
- gestisce le liste e le espressioni regolari come costrutti built-in nel linguaggio;
- sintassi semplificata rispetto a Java;
- linguaggio completamente ad oggetti a tipizzazione statica e/o dinamica;
- rappresentazione e utilizzo di liste e dizionari tramite keyword nel linguaggio;
- possibilità di definire e usare *closure* ovvero pezzi di codice parametrizzabili che possono essere assegnati a variabili e passati come parametri a funzioni;
- utilizzo a livello di linguaggio di espressioni regolari per fare pattern matching/substitution su stringhe di testo;
- costrutto "switch" molto più potente che in Java;
- possibilità di generare intervalli di valori tramite sintassi del linguaggio;
- possibilità di fare l'overload dei principali operatori del linguaggio (ad esempio +, *, /, ecc.).

Funzionalità Groovy non disponibili in Java comprendono la tipizzazione statica e dinamica (con la parola chiave **def**), chiusure, overloading degli operatori, sintassi nativa per liste e array associativi (Maps), il supporto nativo per le espressioni regolari, iterazione polimorfico, le espressioni incorporate all'interno delle stringhe, ulteriori metodi di supporto, e l'operatore della sicurezza della navigazione "?». per controllare automaticamente i null (ad esempio, "variabile?. metodo ()", o "variabile?. campo") .La sintassi Groovy può essere resa più compatta di Java. A differenza di Java, un file sorgente Groovy può essere eseguito come uno (non compilato) script se contiene codice al di fuori di ogni definizione di classe, è una classe con un metodo main, o è un Runnable o GroovyTestCase. Ma, a differenza di alcuni linguaggi di script come Bourne Shell, uno script Groovy è completamente analizzato, compilato, e generato prima dell'esecuzione (simile a Perl e Ruby) (la versione compilata non viene salvato come un artefatto del processo).

Perché si è scelto Groovy rispetto ad altri linguaggi?

Groovy è un linguaggio di programmazione innovativo e in continua evoluzione e sviluppo (nel momento in cui viene scritta questa tesi è in via di sviluppo la versione 2.0), con continui miglioramenti radicali in ogni versione, inoltre è un linguaggio nuovo ma che è arrivato a una versione stabile ed è quindi difficile arrivare a trovare continui bug come negli anni precedenti.

Vi sono molti altri motivi per giustificare la scelta di Groovy su altri linguaggi di programmazione :

- imparare un linguaggio di programmazione dinamico ma simile a Java, in modo da non renderne troppo difficoltoso l'apprendimento; solitamente i linguaggi di programmazione dinamici sono difficoltosi da imparare ma il fatto che Groovy sia simile a Java lo rende di più semplice apprendimento
- Groovy si integra in Java, e ormai siamo in un mondo in cui esiste un programma Java per qualsiasi cosa; bene con Groovy è appunto possibile andare ad usare librerie Java, siano esse API oppure scritte da altri programmatori, che possono integrare un'applicazione senza dover riscrivere codice o dover modificarlo(il codice Java può coesistere dentro lo stesso codice Groovy)
- sintassi compatta: tipizzazione opzionale, chiusure, costruttori e altre chicche in Groovy fanno del codice ripetitivo Java una cosa del passato; si risparmia un sacco di *codice* e sforzo aumentando così la produttività dello sviluppatore rispetto allo stesso lavoro fatto in Java
- produttività: vi sono alcuni framework per Groovy molto utili che permettono di sviulppare facilmente anche applicazioni complesse, ed inoltre attorno a questi framwork sono sorti molti plugin utili
- Groovy usa la Java Virtual Machine ed è quindi altamente portabile
- ci sono altri linguaggi che funzionano sulla JVM ma si tratta solo di port di alri linguaggi stessi, quindi con le proprie API, le proprie politiche, ecc…ed è facile incorrere in problemi di compatibilità.

Scripting

Groovy può essere utilizzato come un linguaggio di script per la piattaforma Java. Java è un ottimo linguaggio per sviluppare applicazioni di qualsiasi tipo ma in certi contesti sarebbe preferibile affiancarlo con tecnologie di scripting. Cerchiamo di capire quali sono le motivazioni che ci portano a ritenere che Groovy sia la tecnologia di scripting migliore che attualmente possiamo affiancare a Java.

Il linguaggio Java è sicuramente uno dei linguaggi di programmazione più usati e apprezzati nell'ambito dello sviluppo software. Ormai sono diversi anni che è presente sul mercato e si può affermare con sicurezza che è un ambiente di programmazione molto maturo ed adatto ad essere utilizzato nei più svariati contesti applicativi.Non a caso il suo utilizzo è diffuso nei più diversi settori applicativi, sia legati al mondo enterprise sia a livello home e personale; è uno strumento potente e flessibile e, grazie anche al suo SDK nativo, permette di sviluppare applicazioni complesse in breve tempo.

Quello che quindi ci si può chiederechiederci a questo punto è se l'ambiente Java sia sempre lo strumento giusto da usare nella scrittura di software. La risposta corretta è "sfortunatamente non sempre". Certe tipologie di applicazioni che sono molto noiose da scrivere in un linguaggio di programmazione a basso livello (inteso nel senso più ampio del termine) quali ad esempio programmi di pattern matching/pattern substitution su documenti di testo, piccole utility di manutenzione del sistema, e così via. Questo ha determinato nel corso degli anni la diffusione di molti linguaggi di scripting (in particolare Perl, Python e Ruby,), molto più espressivi di Java, C, o C++ nella risoluzione di questi problemi, e soprattutto con una sintassi molto più leggera che permette di semplificare e velocizzare enormente la scrittura di tali programmi. Tanto per fare un esempio, il classico "Hello world!" nella maggior parte dei linguaggi di scripting può essere espresso tramite una sola riga di codice mentre in Java avrà una forma del genere:

```
public class HelloWorld {
     public static void main(String[] args) {
        System.out.println("Hello world!");
     }
}
```

Sono 5 righe solo di sintassi, un po' eccessivo per scrivere un programma banale come questo!

La brevità e compattezza del codice non è la sola caratteristica apprezzata nei linguaggi di scripting. Solitamente alcuni tipi di dato molto utilizzati dai programmatori quali liste e dizionari sono espressi all'interno del linguaggio stesso e non attraverso l'uso di librerie esterne. E questo vale talvolta anche per le espressioni regolari per matching di pattern su stringhe. Tutte queste caratteristiche fanno si che la scrittura di codice risulti molto semplificata, riducendo i possibili errori e rendendo la programmazione molto più agile e veloce.

Se si suppone però che una persona sviluppi i suoi progetti generalmente in Java i software da scrivere saranno di solito abbastanza complessi e richiederanno di essere modulari. Alcune parti del codice saranno critiche per il funzionamento dell'applicazione da un punto di vista delle prestazioni (quindi è molto importante che siano efficienti), altre potrebbero non avere questo requisito e potrebbero beneficiare enormemente dall'utilizzo di linguaggi di scripting. In quest'ultimo caso la scrittura dei moduli sarebbe più veloce e semplice, rendendo sicuramente più mantenibile il codice scritto.

In un esempio di questo tipo la difficoltà più grande è quella di rendere interoperabile il codice scritto con linguaggi diversi. Quando si parla di interoperabilità si intende nel senso ampio del termine. Potrebbe essere necessario sia utilizzare codice scritto in Java all'interno di un modulo script sia usare scripting all'interno di codice Java. E questo, per essere indolore al programmatore, dovrebbe poter essere fatto in modo semplice, senza dover escogitare trucchi o forzature nel codice. In quest'ultimo caso infatti avremmo come risultante una accresciuta complessità del software che porterà ad annullare i vantaggi ottenuti dall'uso di linguaggi di scripting.

Sfortunatamente, i linguaggi di scripting più famosi (Perl, Python e Ruby su tutti) non sono facilmente integrabili a runtime con Java (a meno che non si usi qualche implementazione particolare, ad esempio Jython e Jruby) e quindi questa nostra speranza di integrazione sarebbe vana. Dico sarebbe perchè è stato così fino a che non è nato Groovy.

Il linguaggio Groovy viene definito così sul sito ufficiale:

> *An agile dynamic language for the Java Platform with many features that are inspired by languages like Python, Ruby and Smalltalk, making them available to Java developers using a Java-like syntax.*
>
> *Developing web applications , writing shell scripts easily, writing concise, meaningful, test cases using Groovy's JUnit integration, or prototyping and producing real industrial strength applications have never been so concise and groovy.*

In pratica Groovy è un linguaggio molto simile a Java ma con una sintassi molto semplificata che permette di scrivere "codice utile" in modo molto più veloce. I programmi scritti in Groovy possono essere sia interpretati (quindi in questo caso si comporta come un comune linguaggio di scripting), sia compilati in bytecode. In quest'ultimo caso il codice generato è totalmente compatibile con le specifiche della JVM quindi è possibile usarlo in modo trasparente all'interno di classi Java come se fosse codice nativo.

L'altra cosa interessante è che è possibile fare anche l'inverso e usare direttamente classi scritte in Java all'interno di codice Groovy. In questo modo, si può ad esempio sfruttare tutto il JDK per scrivere le proprie applicazioni Groovy. Questo, in effetti, non è una cosa da poco perchè tutta l'esperienza di conoscenza del JDK (che come si sa è un insieme di librerie estremamente vasto) che si hanno nel corso degli anni non andrà persa e può essere riutizzata per programmare in Groovy. In effetti, a causa di

questa totale compatibilità con Java, Groovy non fornisce una suo SDK di base (così come gli altri linguaggi di scripting) ma estende alcune classi JDK con metodi che risultano estremamente utili durante la normale stesura di codice.

1.2 Groovy Ver 1.8 (stable)

Highlights

- the dynamic expressiveness of Groovy, specifically for defining DSLs
- runtime performance
- concurrent and parallel execution
- design by contract
- functional programming style
- first-class JSON support
- compile-time meta programming
- and more helpers and library additions

These features have undergone the Groovy developer process with formal descriptions, discussion, and voting (GEP - Groovy Enhancement Proposal) for core parts and less formal developer discussions and JIRA voting for additional parts.

Our goal has stayed the same, though: to give the Java developer a tool that makes him more productive, allows him to achieve his goals faster and with a smaller margin of error, and extend the scalability of the Java platform from full-blown enterprise projects to everyday "getting things done" tasks.

Command chains for nicer Domain-Specific Languages

Thanks to its flexible syntax and its compile-time and runtime metaprogramming capabilities, Groovy is well known for its Domain-Specific Language capabilities. However, we felt that we could improve upon the syntax further by removing additional punctuation symbols when users chain method calls. This allows DSL implementors to develop command descriptions that read almost like natural sentences.

Before Groovy 1.8, we could omit parentheses around the arguments of a method call for top-level statements. But we couldn't chain method calls. The new "command chain" feature allows us to chain such parentheses-free method calls, requiring neither parentheses around arguments, nor dots between the chained calls. The general idea is that a call like `a b c d` will actually be equivalent to `a(b).c(d)`. This also works with multiple arguments, closure arguments, and even named arguments. Furthermore, such command chains can also appear on the right-hand side of assignments. This new command chain approach opens up interesting possibilities in terms of the much wider range of DSLs which can now be written in Groovy. This new feature has been developed thanks to the Google Summer of Code program,

where our student, Lidia, helped us modify the Groovy Antlr grammar to extend top-level statements to accept that command chain syntax.

Performance improvements

Groovy's flexible metaprogramming model involves numerous decision points when making method calls or accessing properties to determine whether any metaprogamming hooks are being utilized. During complex expression calculations, such decision points involved identical checks being executed numerous times. Recent performance improvements allow some of these checks to be bypassed during an expression calculation once certain initial assumptions have been checked. Basically if certain preconditions hold, some streamlining can take place.

Groovy 1.8.0 contains two main streams of optimization work:

There are several optimizations for basic operations on integers like plus, minus, multiply, increment, decrement and comparisons. This version doesn't support the mixed usage of different types. If an expression contains different types, then it falls back to the classical way of performing the operation, i.e. no streamlining occurs.

There is also an optimization for direct method calls. Such a method call is done directly if it is done on "this" and if the argument types are a direct match with the parameter types of the method we may call. Since this is an operation that does not behave too well with a method call logic based on runtime types we select only methods where the primitive types match, the parameter types are final or for methods that take no arguments. Currently methods with a variable parameter list are not matched in general, unless a fitting array is used for the method call.

Those two areas of optimization are only the beginning of further similar improvements. Upcoming versions of the Groovy 1.8.x branch will see more optimizations coming. In particular, primitive types other than integers should be expected to be supported shortly.

GPars bundled within the Groovy distribution

The GPars project offers developers new intuitive and safe ways to handle Java or Groovy tasks concurrently, asynchronously, and distributed by utilizing the power of the Java platform and the flexibility of the Groovy language. Groovy 1.8 now bundles GPars 0.11 in the libraries of the Groovy installation, so that you can leverage all the features of the library for Fork/Join, Map/Filter/Reduce, DataFlow, Actors, Agents, and more with all the Groovy goodness.

To learn more about GPars, head over to the GPars website, read the detailed online user guide, or check out chapter 17 of Groovy in Action, 2nd Edition (MEAP).

Closure enhancements

Closures are a central and essential piece of the Groovy programming language and are used in various ways throughout the Groovy APIs. In Groovy 1.8, we introduce the ability to use closures as annotation parameters. Closures are also a key part of what gives Groovy its functional flavor.

Native JSON support

With the ubiquity of JSON as an interchange format for our applications, it is natural that Groovy added support for JSON, in a similar fashion as the support Groovy's always had with XML. So Groovy 1.8 introduces a JSON builder and parser.

New AST Transformations

The Groovy compiler reads the source code, builds an Abstract Syntax Tree (AST) from it, and then puts the AST into bytecode. With AST transformations, the programmer can hook into this process. A general description of this process, an exhaustive description of all available transformations, and a guide of how to write you own ones can be found for example in Groovy in Action, 2nd Edition (MEAP), chapter 9.

Alignments with JDK 7

Groovy 1.9 will be the version which will align as much as possible with the upcoming JDK 7, so beyond those aspects already covered in Groovy (like strings in switch and others), most of those "Project Coin" proposals will be in 1.9, except the "diamond operator" which was added in 1.8, as explained in the following paragraph.

New DGM methods

- count Closure variants
- countBy
- equals for Sets and Maps now do flexible numeric comparisons (on values for Maps)
- toSet for primitive arrays, Strings and Collections
- min / max methods for maps taking closures

Oftentimes, when using a map, for example for counting the frequency of words in a document, you need to check that a certain key exists, before doing something with the associating value (like incrementing it). Nothing really complex, but we could improve upon that with a new method, called `withDefault`.

Miscellaneous

- Slashy strings

- Slashy strings are now multi-line:

This is particularly useful for multi-line regexs when using the regex free-spacing comment style (though you would still need to escape slashes):

- Dollar slashy strings

A new string notation has been introduced: the "dollar slashy" string. This is a multi-line GString similar to the slashy string, but with slightly different escaping rules. You are no longer required to escape slash (with a preceding backslash) but you can use '$$' to escape a '$' or '$/' to escape a slash if needed.

- Compilation customizers

The compilation of Groovy code can be configured through the `CompilerConfiguration` class, for example for setting the encoding of your sources, the base script class, the recompilation parameters, etc). `CompilerConfiguration` now has a new option for setting *compilation customizers* (belonging to the `org.codehaus.groovy.control.customizers` package). Those customizers allow to customize the compilation process in three ways:

- adding default imports with the `ImportCustomizer`: so you don't have to always add the same imports all over again
- securing your scripts and classes with the `SecureASTCustomizer`: by allowing/disallowing certain classes, or special AST nodes (Abstract Syntax Tree), filtering imports, you can secure your scripts to avoid malicious code or code that would go beyond the limits of what the code should be allowed to do.
- applying AST transformations with the `ASTTransformationCustomizer`: lets you apply transformations to all the class nodes of your compilation unit.

- When you want to evaluate Math expressions, you don't need anymore to use the `import static java.lang.Math.*` star static import to import all the Math constants and static functions.
- (G)String to Enum coercion

Given a String or a GString, you can coerce it to Enum values bearing the same name.

- Grape/Grab Improvements

- Shorter notation for @GrabResolver

When you need to specify a special grab resolver, for when the artifacts you need are not stored in Maven central, you could use:

```
@GrabResolver(name = 'restlet.org', root = 'http://maven.restlet.org')
@Grab('org.restlet:org.restlet:2.0.6')
import org.restlet.Restlet
```

Groovy 1.8 adds a shorter syntax as well:

```
@GrabResolver('http://maven.restlet.org')
@Grab('org.restlet:org.restlet:2.0.6')
import org.restlet.Restlet
```

- Compact form for optional Grab attributes

The `@Grab` annotation has numerous options. For example, to download the Apache commons-io library (where you wanted to set the `transitive` and `force` attributes - not strictly needed for this example but see the Grab or Ivy documentation for details on what those attributes do) you could use a grab statement similar to below:

```
@Grab(group='commons-io', module='commons-io', version='2.0.1', transitive=false,
force=true)
```

The compact form for grab which allows the artifact information to be represented as a string now supports specifying additional attributes. As an example, the following script will download the commons-io jar and the corresponding javadoc jar before using one of the commons-io methods.

```
@Grab('commons-io:commons-io:2.0.1;transitive=false;force=true')
@Grab('commons-io:commons-io:2.0.1;classifier=javadoc')
import static org.apache.commons.io.FileSystemUtils.*
assert freeSpaceKb() > 0
```

- Sql improvements

The `eachRow` and `rows` methods in the `groovy.sql.Sql` class now support paging. Which will start at the second row and return a maximum of 2 rows.

- Storing AST node metadata

When developing AST transformations, and particularly when using a visitor to navigate the AST nodes, it is sometimes tricky to keep track of information as you visit the tree, or if a combination of transforms need to be sharing some context. The `ASTNode` base class features 4 methods to store node metadata:

```
public Object getNodeMetaData(Object key)
public void copyNodeMetaData(ASTNode other)
public void setNodeMetaData(Object key, Object value)
public void removeNodeMetaData(Object key)
```

- Ability to customize the GroovyDoc templates

GroovyDoc uses hard-coded templates to create the JavaDoc for your Groovy classes. Three templates are used: top-level templates, a package-level template, a class template. If you want to customize these templates, you can subclass the `Groovydoc` Ant task and override the `getDocTemplates()`, `getPackageTemplates()`, and `getClassTemplates()` methods pointing at your own templates. Then you can use your custom GroovyDoc Ant task in lieu of Groovy's original one.

1.3 Installare Groovy

Vi sono due modi per poter installare Groovy: usare un pacchetto di installazione pre-costruito rilasciato ufficialmente oppure compilare direttamente il codice sorgente.

Usando un pacchetto pre-costruito semplicemente si installa Groovy senza doversi preoccupare di niente, tranne alcuni dettagli spiegati successivamente. Vi sono differenti distribuzioni binarie a seconda del sistema operativo, utili se non si vuole perdere tempo a compilare i sorgenti. L'installazione da codice sorgente è leggermente più complessa e viene spiegata nel paragrafo seguente.

1.3.1 Installazione da codice sorgente

Requisiti:

- JDK 1.5+
- Apache Ant version **1.7.0 o superiore**
- Git client version **1.2 o superiore** (se non si dispone del codice sorgente)

Una volta estratti i codici sorgenti, basta eseguire il comando:

```
ant install
```

Una volta completato con successo si può provare la distribuzione scrivendo ad esempio in riga di comando:

```
./target/install/bin/groovysh
```

Installazione con pacchetto binario pre-costruito

- Linux

Ecco i passi da eseguire per installare Groovy:

- Scompattare il pacchetto scaricato in una directory a vostra scelta sul vostro hard disk. Una buona scelta per Unix-like è **/opt**. In questo caso assumeremo di scompattare il pacchetto nella directory **/opt/groovy**.
- A questo punto bisogna impostare la variabile d'ambiente GROOVY_HOME alla directory dove è stata scompattata la distribuzione Groovy. Assumendo di disporre di una shell Bash, si edita il file **~/.bash_profile** e si aggiunge la riga

  ```
  export GROOVY_HOME=/opt/groovy
  ```

  E' inoltre necessario aggiornare la variabile d'ambiente PATH per tenere traccia della directory bin di Groovy. Si aggiunge quindi anche la riga:

  ```
  export PATH=${PATH}:${GROOVY_HOME}/bin
  ```

- Per essere sicuri che tutto funzioni correttamente bisogna ricordarsi anche di definire la variabile d'ambiente JAVA_HOME che indica il path di installazione del JDK. Controllare quindi che l'output del comando **env** contenga tale variabile altrimenti aggiungerla al file **~/.bash_profile.**

A questo punto Groovy dovrebbe essere installato. Per vedere che tutto funzioni correttamente, aprire una shell e digitare al prompt il comando **groovy**. Dovrebbe venire fuori qualcosa del genere:

```
fagni:~ fagni$ groovy
error: neither -e or filename provided
usage: groovy
 -a,--autosplit <splitPattern>   automatically split current line
                (defaults to '\s'
 -c,--encoding <charset>         specify the encoding of the files
```

```
-d,--debug                    debug mode will print out full stack
                              traces
-e <script>                   specify a command line script
-h,--help                     usage information
-i <extension>                modify files in place, create backup if
                              extension is given (e.g. '.bak')
-l <port>                     listen on a port and process inbound
                              lines
-n                            process files line by line
-p                            process files line by line and print
                              result
-v,--version                  display the Groovy and JVM versions
```

- Mac OS X

L'installazione su Mac è eseguita in modo analogo a Unix ma può essere eseguita in modo alternativo dopo aver aperto il terminale con il comando:

```
sudo port install groovy
```

- Windows

Anche l'installazione su Windows avviene come quella su Unix solo che l'impostazione delle variabili avviene in maniera diversa: dalla finestra "Variabili d'ambiente" aggiungere alla variabile PATH il percorso dei file binari groovy scompattati; creare la variabile GROOVY_HOME con il percorso della cartella Groovy; creare la variabile JAVA_HOME se questa non è già presente con il percorso della cartella Java JDK.

1.4 L'interprete Groovy e groovyConsole

Il comando groovy lancia l'inteprete Groovy. Esso può eseguire sia un programma contenuto in un file (solitamente con estensione ".groovy") oppure un programma letto direttamente dallo standard input. Quest'ultima modalità è molto comoda per eseguire script al volo, magari da utilizzare in pipe con altri script o comandi del sistema operativo. Vediamo un piccolo esempio di uso di questa modalità. Aprire una shell e digitare al prompt:

```
ls | groovy -e "System.in.eachLine{println(it.toLowerCase())}"
```

Il comando "ls" da in output la lista dei file/directory contenuti nella directory corrente. Questa viene passata allo script Groovy che non far altro che prendere ciascun nome di file/directory, trasformare tutte le lettere in minuscolo e stampare la riga a video. Come si può capire l'esempio in se per se non è molto utile ma lascia intendere le potenzialità dell'approccio.

Nel caso il programma Groovy da eseguire sia contenuto in un file su disco basta lanciare il comando:

```
groovy nome_file
```

dove ovviamente "nome_file" è il file contenente lo script Groovy.

Il comando **groovyc** è l'equivalente Groovy del **javac** di Java, compila uno script groovy in... Java!

Il comando **groovyConsole** lancia una applicazione che permette di scrivere, editare ed eseguire script groovy in un ambiente grafico:

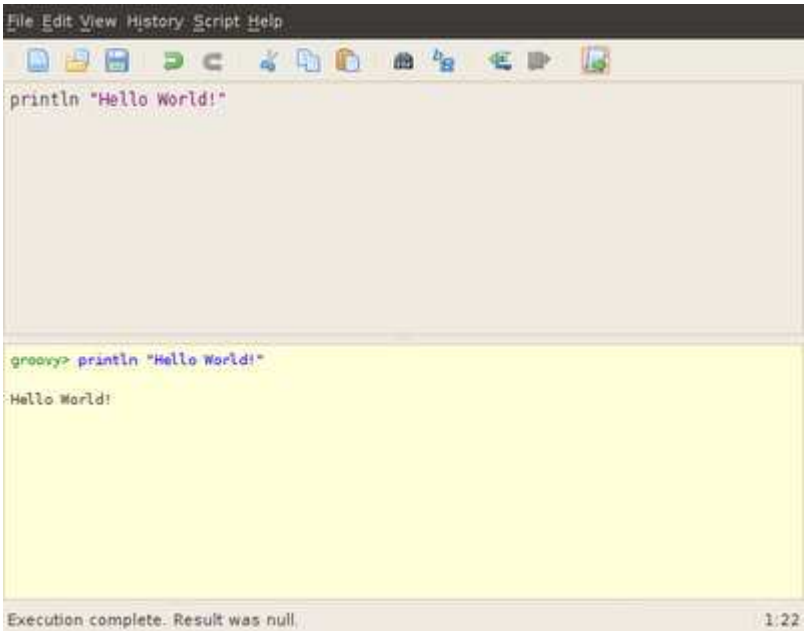


Fig.2 esempio di groovyConsole

1.5 Esempi

Ecco come era possibile in Java scrivere il famoso programma Hello World:

```
public class Main {
    public static void main(String[] arguments) {
        System.out.println("Hello World");
    }
}
```

Ora applicando un primo approccio Groovy questo codice diventa:

```
public class Main {
    public static void main(String[] arguments) {
        println "Hello World"
    }
}
```

Come si può notare invece di scrivere **System.out.println** in Groovy basta semplicemente scrivere **println** !

Ma non è tutto, il codice può essere ulteriormente semplificato e diventare semplicemente:

```
println "Hello World"
```

Non serve nemmeno dichiarare la classe in quanto Groovy è un linguaggio di scripting!

Esempio per vedere se è giorno o notte:

```
amPM = Calendar.getInstance().get(Calendar.AM_PM)
if (amPM == Calendar.AM) {
    println("Good morning")
} else {
    println("Good evening")
}
```

Ordinare una lista di colori in base alla lunghezza del nome:

```
colours = ['red','orange','yellow','green','blue','indigo','violet']

byLength = new Comparator() {
    int compare(a,b) { a.size() <=> b.size() }
}
println colours.sort(byLength)
```

Unire due array:

```
arr1 = ['red','green','blue']
arr2 = ['orange','yellow','indigo','violet']
colours = arr1 + arr2
println colours
```

Semplice esempio su come usare le chiusure:

```
class Equipment {
  def calculator

  Equipment(calc) { calculator = calc }

  def simulate()   {
    println "Running simulation"
    calculator() // You may send parameters as well
  }
}

eq1 = new Equipment() { println "Calculator 1" }

aCalculator = { println "Calculator 2" }

eq2 = new Equipment(aCalculator)
eq3 = new Equipment(aCalculator)

eq1.simulate()
eq2.simulate()
```

```
eq3.simulate()
```

6) Produrrà come output:

```
Running simulation
Calculator 1
Running simulation
Calculator 2
Running simulation
Calculator 2
```

2 Netbeans

Groovy é il linguaggio di programmazione che verrà usato nello sviluppo dell'applicazione presentata in questa tesi, ovviamente è possibile programmare anche con un semplice editor di testo notepad ma per una più semplice, efficiente e efficace programmazione è vivamente consigliato l'uso di un IDE di sviluppo; lo scopo di questo capitolo è quello di fornire lo strumento IDE per poter programmare in Groovy (e altri linguaggi di programmazione) e per poter eventualemente testare i programmi presentati.

Quindi questo breve capitolo, di scopo introduttivo, presenta rapidamente il concetto di IDE; concetto fondamentale per capire di cosa stiamo parlando.

Si inizia quindi con una presentazione dell'ambiente di sviluppo scelto nello sviluppo di questo progetto ovvero Netbeans, con una rapida panoramica sullo stesso e sulle semplici istruzioni per installarlo.

2.1 Introduzione a Netbeans

Un integrated development environment (IDE), in italiano ambiente di sviluppo integrato, (conosciuto anche come integrated design environment o integrated debugging environment, rispettivamente ambiente integrato di progettazione e ambiente integrato di debugging) è un software che aiuta i programmatori nello sviluppo del codice.

Normalmente consiste in un editor di codice sorgente, un compilatore e/o un interprete, un tool di building automatico, e (solitamente) un debugger.

A volte è integrato con un sistema di controllo di versione e con uno o più tool per semplificare la costruzione di una GUI.

Alcuni IDE, rivolti allo sviluppo di software orientato agli oggetti, comprendono anche un navigatore di classi, un analizzatore di oggetti e un diagramma della gerarchia delle classi.

Sebbene siano in uso alcuni IDE multi-linguaggio, come Eclipse, NetBeans e Visual Studio, generalmente gli IDE sono rivolti ad uno specifico linguaggio di programmazione, come Visual Basic o Delphi.

Ora che è stato chiarito il concetto di IDE si può passare a Netbeans:

NetBeans è un ambiente di sviluppo multi-linguaggio open-source mirato a fornire un solido prodotto per sviluppare software che guida le necessità di sviluppatori, utenti e commerciali che si affidano a NetBeans per i loro prodotti. Il progetto NetBeans è anche una fervida comunità dove chiunque da qualsiasi paese può pensare, porre domande, esprimere pareri e contribuire in tanti modi; è scritto interamente in Java ed è nato nel giugno 2000. È l'ambiente scelto dalla Oracle Corporation come IDE ufficiale, da contrapporre al più diffuso Eclipse.

Con oltre 18 milioni di download di NetBeans IDE sino ad oggi, e oltre 700,000 sviluppatori che partecipano, il progetto netbeans.org è fiorente. Ogni mese contribuiscono ed usano NetBeans visitatori di oltre 130 diversi paesi.

Possiede numerosi plug-in che lo rendono appetibile al pubblico, e richiede 512 Megabyte di Ram a causa dell'uso delle librerie grafiche standard di Java (Swing).

La storia di NetBeans incomincia nel 1997 come Xelfi, un progetto studentesco sotto la guida della Facoltà di Matematica e Fisica alla Charles University di Praga. Una società fu successivamente creata attorno al progetto, rilasciando una versione commerciale dell'IDE, finché non fu acquisita nel 1999 da Sun Microsystems. Sun decise di rilasciare i sorgenti di NetBeans nel mese di giugno dell'anno successivo.

La comunità di sviluppatori che gira attorno a NetBeans ha potuto così continuare lo sviluppo dell'IDE, grazie anche agli apporti avuti da compagnie e da sviluppatori esterni al progetto.

L'ultima versione di Netbeans, che è stata usata nello sviluppo di questo progetto è NetBeans 7.1.

Le principali novità introdotte dalla nuova versione dell'ambiente di sviluppo integrato Java-based rigurdano l'introduzione del supporto per JavaFX 2.0 come la compilazione, il debugging e il profiling di applicazioni, e la loro distribuzione per il desktop come applet o tramite JNLP.

Il supporto di NetBeans per JavaFX richiede, attualmente, l'utilizzo dell'SDK proprietaria ma open source JavaFX 2.0 SDK anche se è stato da poco integrato nella più recente release di aggiornamento di Java SDK 7.

I miglioramenti apportati allo Swing GUI Builder, strumenti di supporto per CSS3 e per il debugging sia di interfacce Swing e che JavaFX sono stati anche inseriti.

Il supporto a CSS3 è necessario per JavaFX in quanto è lo strumento necessario per la per personalizzare dei controlli dell'interfaccia utente, ma, naturalmente, ha anche la doppia valenza di spingere e migliorare il supporto per lo sviluppo di applicazioni web-based dell'IDE di casa Oracle.

La nuova versione, ancora, migliora e aggiunge definitivamente il support per respository Git gli sviluppatori Java EE troveranno maggiori opzioni per l'implementazione di Java EE con supporto per Glassfish, componenti JSF, e miglioramenti a Java Persistence.

I dettagli completi di questi e tutti gli altri miglioramenti che si possono trovare in NetBeans 7.1 sono disponibili sul wiki di NetBeans.

2.2 Installare Netbeans 7.1

Prerequisiti: avere Java JDK già installato sul computer nel quale si intende installare Netbeans, in caso contrario provvedere a installarlo scaricando l'ultima versione disponibile.

Ecco i passi necessari per installare Netbeans:

Microsoft Windows, Solaris OS, and Linux

- Dopo aver completato il download, eseguire il programma di installazione.

- Per Windows, il file eseguibile di installazione ha l'estensione exe. Fare doppio clic sul file di installazione per eseguirlo
- Per le piattaforme Solaris e Linux, il file di installazione ha l'estensione sh. Per queste piattaforme, è necessario rendere il file di installazione eseguibile utilizzando il seguente comando: `chmod + x <installer-file-name>`

- Se avete scaricato il pacchetto All bundle, è possibile personalizzare l'installazione. Effettuare le seguenti operazioni nella pagina iniziale della procedura guidata di installazione:
- - Fare clic su Personalizza.
  - Nella finestra di dialogo Installazione Personalizza, effettuare le selezioni.
  - Fare clic su OK.
- Nella pagina iniziale della procedura guidata di installazione, fare clic su Avanti.
- Nella pagina Contratto di licenza, leggere il contratto di licenza, fare click sulla casella di controllo per l'accettazione, e fare clic su Avanti.
- Nella pagina Contratto di Licenza JUnit, decidere se si desidera installare JUnit e selezionare l'opzione appropriata, fare clic su Avanti.
- Nella pagina di installazione di NetBeans IDE, procedere come segue:
  - Accettare la directory di installazione predefinita per l'IDE NetBeans o specificare un'altra directory.
  - Accettare l'installazione di default JDK da utilizzare con il NetBeans IDE o selezionare una differente installazione dal menu a discesa. Se la procedura guidata di installazione non ha trovato un'installazione compatibile JDK da utilizzare con il NetBeans IDE, il JDK non è installato nel percorso predefinito. In questo caso, specificare il percorso di un JDK installato e fare

clic su Avanti, o annullare l'installazione corrente. Dopo aver installato il JDK richiesto è possibile riavviare l'installazione.

- Se la pagina di installazione del server GlassFish Open Source Edition 3.1.2 si apre, accettare la directory di installazione predefinita o specificare un altro percorso di installazione.
- Se si sta installando Apache Tomcat, sulla sua pagina di installazione, accettare la directory di installazione predefinita o specificare un altro percorso di installazione. Fare clic su Avanti.
- Nella pagina di riepilogo, verificare che l'elenco dei componenti da installare sia corretto e che si disponga di uno spazio adeguato sul vostro sistema per l'installazione.
- Fare clic su Installa per avviare l'installazione.
- Nella pagina Installazione completata, fornire i dati (anonimi), se desiderato, quindi fare clic su Fine.

Mac OS X

- Dopo aver completato il download, eseguire il programma di installazione. Il file di installazione ha l'estensione dmg.
- Nel pannello che si apre fare clic sull'icona del pacchetto. Il pacchetto ha l'estensione Mpkg. La procedura guidata di installazione ha inizio.
- Nella pagina iniziale della procedura guidata di installazione, fare click su Continua.
- Rivedere il contratto di licenza e fare click su Continua. Fare clic su Accetta nella finestra pop-up per accettare la licenza.
- Al "Seleziona una pagina di destinazione", selezionare l'unità e fare click su Continua.
- Se avete scaricato il pacchetto All bundle, è possibile personalizzare l'installazione. Sull'ultimo pannello della proceduraguidata di installazione, premere il pulsante Personalizza nella parte inferiore sinistra del pannello. L'albero dei prodotti viene visualizzato.
- Selezionare i prodotti che si desidera installare.
- Immettere il nome dell'amministratore e la password per il sistema e fare click su OK per avviare l'installazione.

2.3 "Hello World!" in Netbeans

Si riporta una rapido tutorial di esempio per mostrare un primo approccio all'utilizzo di Netbeans. Nel progetto è stato seguito un porcedimento molto simile, con la differenza che è stato usato anche Groovy.

- Avviare NetBeans. Dal menu **File** cliccare su **Nuovo Progetto**.... Dalla finestra Nuovo Progetto **selezionare Java** dalla lista delle categorie e **Applicazione Java** dalla lista dei progetti e cliccare su **Avanti**;

- Al secondo passo dare un nome al progetto e selezionare, tramite il pulsante Visualizza... in corrispondenza dell'etichetta Project Location:, la rispettiva cartella di destinazione dove verranno salvati tutti i file ad esso associati. Come si può osservare dalla figura 3, se si lascia spuntata l'opzione Create Main Class il programma creerà automaticamente una classe con il nome indicato dal corrispettivo campo di testo e che conterrà il main dell'applicazione Java che andremo a scrivere. Lasciamo spuntato **Create Main Class** dando come nome alla classe HelloWorld e cliccare su **Termina**.

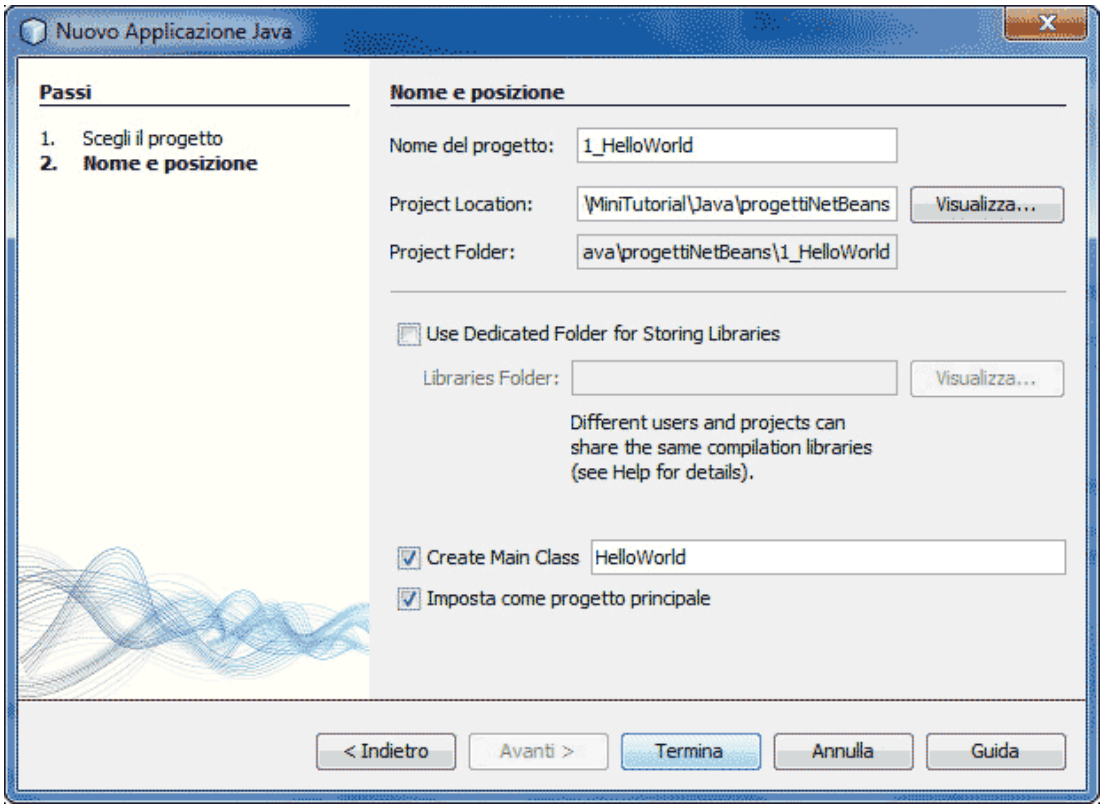


Fig.3 Creazione di un'applicazione Java in Netbeans

- A questo punto apparirà nell'editor di testo la classe HelloWorld con il rispettivo main. Inserire nella procedura main la seguente riga di codice la quale non fa altro che visualizzare a video la frase "Hello World!":

```
System.out.println("Hello World!");
```

- Adesso non resta che lanciare l'applicazione cliccando su **Run Main Project** dal menu Esegui, oppure cliccando sull'apposita icona come mostra la figura successiva e vedere l'output.

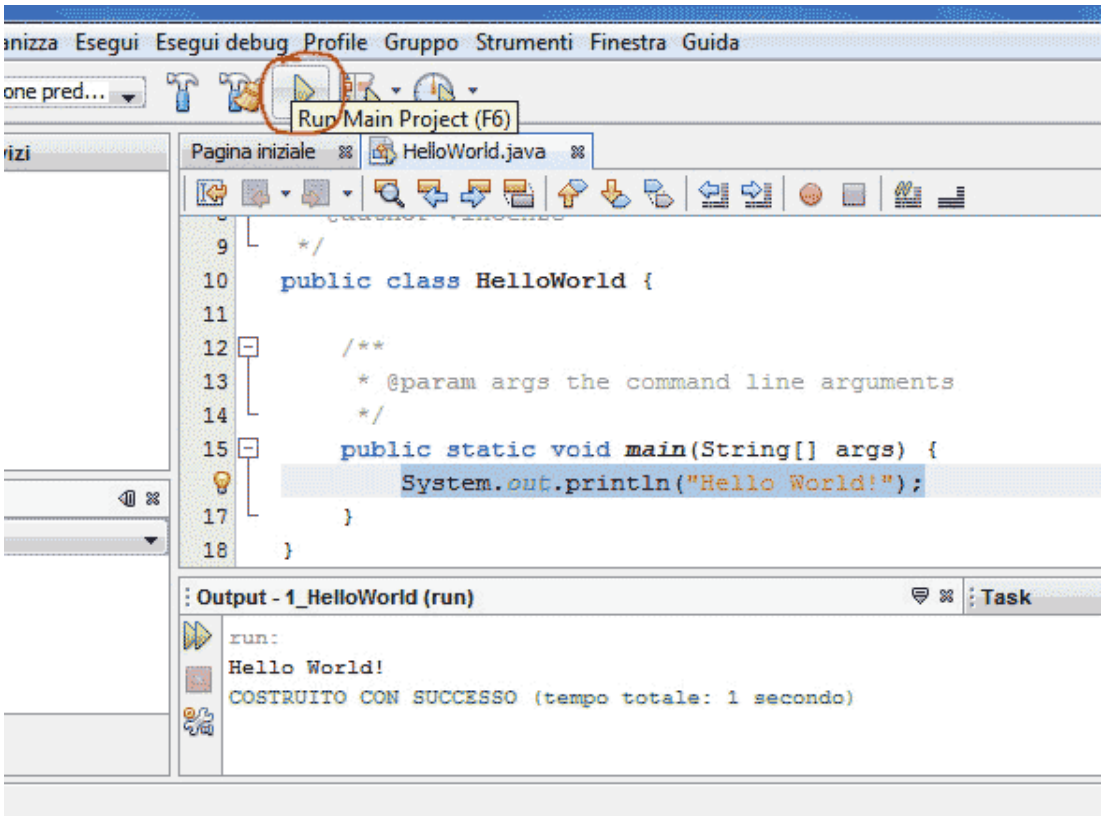


Fig.4 HelloWorld in esecuzione

3 Architettura MVC

In questo capitolo viene analizzato un altro elemento costitutivo del progetto: l'architettura MVC ovvero modello-vista-controllore. Questa architettura è quella più comunemente usata nei framework di sviluppo di applicazioni web e non, ed è senza dubbio un'architettura efficace ed efficiente nella progettazione delle stesse.

I framework utilizzati in questa tesi adottano tutti questo tipo di modello, che rende la programmazione stessa piuttosto piacevole e permette di eliminare alcune fastidiose perdite di tempo dovute ad aspetti di programmazione di basso livello, concettualmente banali ma fastidiosi in fase di realizzazione dell'applicazione.

Questo breve capitolo definisce quindi il concetto di architettura MVC, spiegandone i concetti principali e permettendo quindi al lettore di comprendere meglio la progettazione dell'applicazione finale e i framework presentati con relativi esempi realizzativi. Vengono presentate le caratteristiche principali e i vantaggi che si ottengono dall'utilizzo di questo modello con due esempi che verranno poi usati nel progetto.

3.1 Cos'è l'architettura MVC?

Con lo sviluppo delle interfacce utente grafiche (GUI) dei primi anni 80, i progettisti ed i programmatori si sono trovati a riaffrontare vecchi problemi ed a confrontarsi con nuove sfide. Da un lato, il problema della portabilità dell'interfaccia utente, risolta in ambiente a caratteri attraverso standard come l'ANSI o la creazioni di librerie come Curses, basate su un database di terminali (il famoso termcap di Unix) si presentava ora in una luce ed una complessità completamente nuove. Dall'altro, nuovi problemi come la stabilità e l'espandibilità dell'applicazione cominciavano ad emergere, in quanto i nuovi ambienti grafici consentivano di scegliere tra una più ampia gamma di modelli di interazione, e lasciavano intravedere possibilità sempre nuove.
Modificare il look di una applicazione, pur lasciandone inalterate le funzionalità,

diventava un problema sempre più sentito; non solo, anche la possibilità di rappresentare le stesse informazioni in modo diverso, sfruttando più a fondo le potenzialità della grafica, sembrava un problema difficile da affrontatare per chi proveniva dalla cultura dei terminali a carattere.
Il problema della portabilità è stato affrontato soprattutto in ambiente Unix, tradizionalmente attento allo sviluppo cross-platform: ciò ha portato alla creazione di piattaforme GUI come X Windows ed al concetto di toolkit, che in questo articolo verranno solo accennati.
Viceversa, i problemi di stabilità ed estendibilità hanno visto diversi tentativi di soluzione, di cui uno è stato particolarmente fortunato: il paradigma architetturale Model-View-Controller, introdotto da Trygve Reenskaug in Smalltalk nella prima metà degli anni 80, e poi trasformatosi nel paradigma Document/View che è ora alla base di diverse librerie commerciali come MFC ed OWL.

MVC viene definito un "pattern architetturale" ... mmm ... questo non ci aiuta molto, anzi. Cerchiamo di espandere questa definizione. Un "design pattern" (che è più o meno la traduzione di pattern architetturale) in ambito ingegneristico-informatico può essere definito come la soluzione generale ad un problema ricorrente. In pratica è un modello che può essere applicato a problemi che solitamente si presentano sempre uguali durante lo sviluppo di un software.

Ma torniamo alla nostra sigla iniziale: MVC. Model-View-Controller (talvolta tradotto in italiano Modello-Vista-Controllo) è un pattern architetturale molto diffuso nello sviluppo di interfacce grafiche di sistemi software object-oriented. Originariamente impiegato dal linguaggio Smalltalk, il pattern è stato esplicitamente o implicitamente sposato da numerose tecnologie moderne, come framework basati su PHP (Symfony, Zend Framework, CakePHP), su Ruby (Ruby on Rails), su Python (Django, TurboGears, Pylons, Web2py), su Java (Swing, JSF e Struts), su Objective C o su .NET.

A causa della crescente diffusione di tecnologie basate su MVC nel contesto di framework o piattaforma middleware per applicazioni Web, l'espressione framework MVC o sistema MVC sta entrando nell'uso anche per indicare specificamente questa categoria di sistemi (che comprende per esempio Ruby on Rails, Struts, Spring, Tapestry e Catalyst).

Il pattern è basato sulla separazione dei compiti fra i componenti software che interpretano tre ruoli principali: model, controller e view. Questo schema implica la tradizionale separazione fra la logica applicativa (in questo contesto spesso chiamata "logica di business"), a carico del controller e del model, e l'interfaccia utente a carico del view.

3.2 Le tre componenti dell’architettura

I tre componenti dell’architettura MVC sono:

MODEL

il core dell'applicazione viene implementato dal Model, che incapsulando lo stato dell'applicazione definisce i dati e le operazioni che possono essere eseguite su questi. Quindi definisce le regole di business per l'interazione con i dati, esponendo alla View ed al Controller rispettivamente le funzionalità per l'accesso e l'aggiornamento. Per lo sviluppo del Model quindi è vivamente consigliato utilizzare le tipiche tecniche di progettazione object oriented al fine di ottenere un componente software che astragga al meglio concetti importati dal mondo reale. Il Model può inoltre avere la responsabilità di notificare ai componenti della View eventuali aggiornamenti verificatisi in seguito a richieste del Controller, al fine di permettere alle View di presentare agli occhi degli utenti dati sempre aggiornati.

VIEW

La logica di presentazione dei dati viene gestita solo e solamente dalla View. Ciò implica che questa deve fondamentalmente gestire la costruzione dell' interfaccia grafica (GUI) che rappresenta il mezzo mediante il quale gli utenti interagiranno con il sistema. Ogni GUI può essere costituita da schermate diverse che presentano più modi di interagire con i dati dell'applicazione. Per far sì che i dati presentati siano sempre aggiornati è possibile adottare due strategie note come "push model" e "pull model". Il push model adotta il pattern Observer, registrando le View come osservatori del Model. Le View possono quindi richiedere gli aggiornamenti al Model in tempo reale grazie alla notifica di quest'ultimo. Benché questa rappresenti la strategia ideale, non è sempre applicabile. Per esempio nell'architettura J2EE se le View che vengono implementate con JSP, restituiscono GUI costituite solo da contenuti statici (HTML) e quindi non in grado di eseguire operazioni sul Model. In tali casi è possibile utilizzare il "pull Model" dove la View richiede gli aggiornamenti quando "lo ritiene opportuno". Inoltre la View delega al Controller l'esecuzione dei processi richiesti dall'utente dopo averne catturato gli input e la scelta delle eventuali schermate da presentare.

CONTROLLER

Questo componente ha la responsabilità di trasformare le interazioni dell'utente della View in azioni eseguite dal Model. Ma il Controller non rappresenta un semplice "ponte" tra View e Model. Realizzando la mappatura tra input dell'utente e processi eseguiti dal Model e selezionando la schermate della View richieste, il Controller implementa la logica di controllo dell'applicazione.

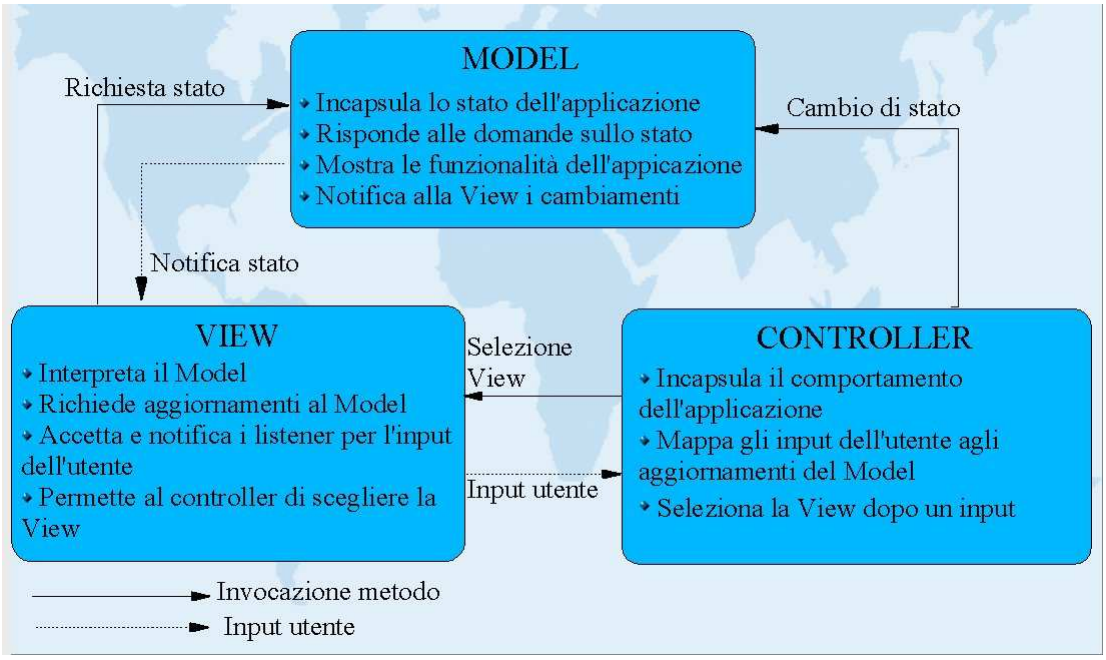


Fig.4 L'architettura MVC

Riassumendo quindi:

- Model sarà il componente a cui verrà demandato il compito dell'accesso ai dati utili all'applicazione;
- View sarà il componente che visualizzerà i dati nel model e che si farà carico dell'interazione con l'utente;
- Controller sarà il componente che riceve i comandi andando a modificare lo stato degli altri due componenti;

3.3 Esempi di applicazione

Ecco due esempi di scenari che potrebbero presentarsi utilizzando un applicazione MVC-based: la richiesta dell'utente di aggiornare dati e la richiesta dell'utente di selezionare una schermata, rispettivamente illustrate tramite diagrammi di sequenze nelle figure 5 e 6.

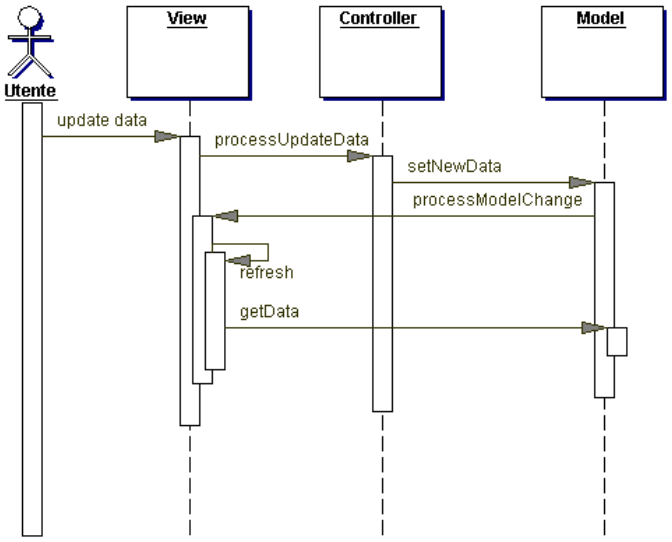


Fig. 5 Aggiornamento dei dati

Dalla figura 5 si evidenzia il complesso scambio di messaggi tra i tre partecipanti al pattern, che benché possa sembrare inutile a prima vista, garantisce pagine sempre aggiornate in tempo reale all'utente.

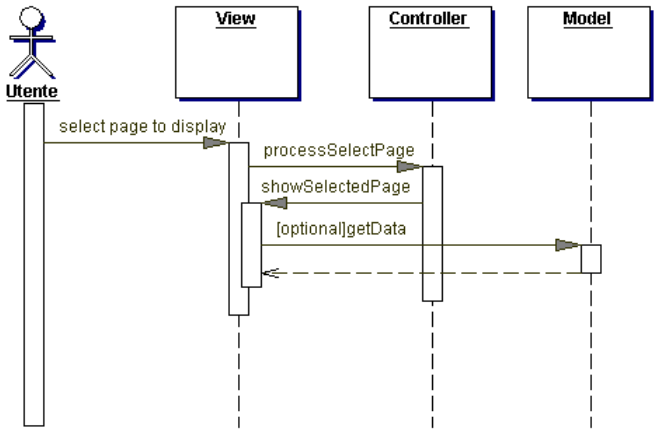


Fig. 6 Visualizzazione di una pagina scelta dall'utente

Nella figura 6 vengono enfatizzati i ruoli di View e Controller. La View infatti non decide quale sarà la schermata richiesta, delegando ciò alla decisione del Controller (che grazie al pattern Strategy può anche essere anche cambiato dinamicamente al runtime), piuttosto si occupa della costruzione e della presentazione all'utente della schermata stessa.

Uno scenario analogo a questo si presenterà nell'applicazione finale di questo progetto.

3.4 Riflessioni

Ecco alcune riflessioni sull'architettura MVC.

Le conseguenze principali di questo tipo di architettura sono:

- Riuso dei componenti del Model. La separazione tra Model e View permette a diverse GUI di utilizzare lo stesso Model. Conseguentemente, i componenti del Model sono più semplici da implementare testare e manutenere, giacché tutti gli accessi passano tramite questi componenti.
- Supporto più semplice per nuovi tipi di client, Infatti basterà creare View e Controller per interfacciarsi ad un Model già implementato.
- Avere il controllore separato dal resto dell'applicazione rende la sua progettazione piú semplice permettendo di concentrare gli sforzi sulla logica del funzionamento.
- Obbliga gli sviluppatori a rispettare uno standard nella stesura del progetto, che ne facilita poi la comprensione e le successive implementazioni, soprattutto utilizzabile in progetti di medie/grandi dimensioni.
- Software più flessibile, mantenibile ed aggiornabile nel tempo.
- Complessità notevole della progettazione. Questo pattern introduce molte classi extra, ed interazioni tutt'altro che scontate.
- Adatto soprattutto a progetti medio/grandi.
- Flessibilitá dipendente dal framework utilizzato.

In conclusione: il pattern MVC, introduce una notevole complessità all'applicazione, ed è sicuramente non di facile comprensione. Tuttavia il suo utilizzo si rende praticamente necessario in tantissime applicazioni moderne, dove le GUI sono insostituibili. Non conviene avventurarsi alla scoperta di nuove interazioni tra logica di business e di presentazione, se una soluzione certa esiste già.

4 I framework

Dopo aver scelto il linguaggio e l'IDE di sviulppo si passa ora alla scelta, molto importante, del framework con cui si vuole sviluppare l'applicazione finale.

In questo capitolo verranno presentati alcuni framework per lo sviluppo di applicazioni web in Groovy con supporto Java; questi framework vengono presentati, testati attraverso esempi di applicazioni e infine valutati per arrivare alla fine a scegliere il framework ritenuto migliore per lo sviluppo del progetto finale. Da questo capitolo inizia il cuore di questo lavoro di tesi in quanto si vanno a sviluppare applicazioni web in Groovy, non propriamente semplici.

Dopo una breve introduzione, nel capitolo viene prima descritto Google App Engine, il framework per applicazioni web offerto da Google, che supporta Groovy, viene presentato con un esempio realizzativo e valutato.

Viene poi descritto il framework Play!, che risulterà molto buono e semplice per lo sviluppo di applicazioni, ma presenterà un difetto: non accetterà Groovy come linguaggio, questo sembrerà strano in quanto Play! è scritto interamente in Groovy (e ci si aspetta il supporto Groovy nelle prossime versioni) .

Infine si passa al framework ben più conosciuto e famoso ovvero Griffon: questo framework è quello che verrà scelto per lo sviluppo del progetto finale, in quanto supporta Groovy pienamente, è piuttosto consolidato e affermato ed è open source, quindi modificabile a seconda delle eventuali esigenze .

Nel panorama dello sviluppo web vi è una vasta, vastissima scelta riguardo a possibili framework in grado di semplificare la vita di uno sviluppatore. Si inizia con la scelta del linguaggio di programmazione: PHP, python, ruby,Java, Groovy, ecc, ma ci si rende presto conto che, per ognuno di questi linguaggi, i framework sono virtualmente infiniti. Allora si passano al vaglio le caratteristiche di ognuno analizzando i pregi e i difetti e, in base alle proprie esigenze, alla propria esperienza e, perché no, alla simpatia rispetto ad un qualche particolare,si inizia ad eseguire qualche "prova". I tre framework che hanno attirato di più l'attenzione sono GAE,

Play! e Griffon. Si va quindi a vedere uno per uno questi framework ed infine dopo averne valutato ciascuno si sceglie quale si utilizzerà per l'applicazione finale.

4.1 Google App Engine framework

Google App Engine è la piattaforma di hosting e sviluppo di applicazioni di Google, è un insieme di tecnologie che permettono di costruire applicazioni web sulla stessa infrastruttura utilizzata da Google per le proprie applicazioni, nota per la proprià affidabilità. Applicazioni App Engine sono facili da costruire, mantenere nel tempo e soprattutto scalabili quando crescono le esigenze in termini di traffico e di storage dei dati.

Inoltre con App Engine non ci sono server da mantenere, nessuna macchina da gestire: basta fare l'upload della propria applicazione sull'infrastruttura messa in piedi da Google.App Engine di base è gratuito ma le risorse utilizzabili sono limitate: possono essere comunque acquistate a parte.

Ecco qui di seguito alcune caratteristiche utili:

- Facile da usare
- Scalabilità automatica
- Affidabilità, performance e sicurezza dell'infrastruttura di Google
- Periodo di prova gratuito

Facile da usare

App Engine offre un completo stack per lo sviluppo e l'hosting di applicazioni web usando tecnologie comuni. Con App Engine si può scrivere il codice della propria applicazione, testarlo sulla macchina locale e caricarlo su Google con un semplice click o comando. Una volta che l'applicazione è caricata su Google, viene ospitata dai server di Google. Non è più necessario preoccuparsi dell'amministrazione del sistema, introdurre nuove istanze dell'applicazione, condividere il database o acquistare macchinari. App Engine si prende cura di tutta la manutenzione in modo che il programmatore possa concentrarsi sulle funzionalità per gli utenti.

Scalabilità automatica

Per la prima volta le applicazioni possono usufruire delle stessa infrastruttura scalabile sulla quale sono costruite le applicazioni di Google, come ad esempio BigTable e GFS. La scalabilità automatica è parte integrante di App Engine, tutto quello che si deve fare è scrivere il codice dell'applicazione e App Engine penserà al resto. Non importa quanti utenti si hanno o quanti dati memorizza l'applicazione, la scalabilità di App Engine è fatta per soddisfare queste esigenze.

Affidabilità, performance e sicurezza dell'infrastruttura di Google

Google ha una reputazione di grande affidabilità e di un'infrastruttura dalle alte prestazioni. Con App Engine è possibile usufruire dei 10 anni di conoscenza che

Google ha nell'esecuzione di sistemi altamente scalabili e basati sulle prestazioni. La stessa politica di sicurezza, privacy e protezione dei dati che si ha per le applicazioni di Google viene attuata per tutte le applicazioni di App Engine. Un altro aspetto che viene tenuto in grande considerazione è la sicurezza, mettendo in atto delle misure per proteggere il codice e i dati delle applicazioni.

Efficienza dei costi di hosting

Per iniziare App Engine è gratuita, ed è possibile acquistare maggiori risorse, pagando solo quello che effettivamente si usa. Un listino dettagliato dei prezzi per l'utilizzo di tutto ciò che eccede la versione gratuita di 500 MB di memoria e di circa 5 milioni di pagine visualizzate al mese è disponibile sul sito di App Engine.

Periodo di prova gratuito senza rischi

Creare un applicazione con App Engine è facile. Si può creare un account e pubblicare un'applicazione che le persone possono utilizzare senza alcun costo e senza alcun obbligo. Un'applicazione efficace su un account gratuito può utilizzare fino a 500 MB di memoria e fino a 5 milioni di pagine visualizzate al mese. Quando si è pronti per più spazio, è possibile attivare la fatturazione, impostare un budget giornaliero massimo e destinare il budget per ogni risorsa in base alle proprie esigenze

Google App Engine rende facile costruire un'applicazione che viene eseguita in modo affidabile anche se è pesante o contiene una grande quantità di dati. App Engine include le seguenti caratteristiche:

- Web serving dinamico con pieno supporto per le tecnologie web più comuni
- Storage persistente che supporta query, ordinamento e transazioni
- Scalabilità automatica e load balancing
- API per l'autenticazione degli utenti e l'invio di email utilizzando gli account Google
- Un ambiente di sviluppo locale completo che simula Google App Engine sul proprio computer
- Code di elaborazione (task queues) dedicate ad effettuare delle operazioni al di fuori delle richieste web
- Operazioni pianificate per attivare eventi in momenti specifici e a intervalli regolari

Google App Engine supporta applicazioni scritte in due linguaggi di programmazione. Con l'ambiente Java messo a disposizione da App Engine è possibile costruire le applicazioni usando tecnologie Java standard, tra cui JVM, Java servlet, il linguaggio di programmazione Java o qualsiasi altro linguaggio basato su un interprete o compilatore JVM (quindi supporta Groovy). App Engine offre inoltre un ambiente Python dedicato, che include un veloce interprete Python e la libreria standard. Gli ambienti Java e Python sono costruiti per garantire che l'applicazione venga eseguita in modo rapido, sicuro e senza interferenze da parte di altre applicazioni sul sistema.

Le applicazioni vengono eseguite in ambiente sicuro che offre un accesso limitato al sistema operativo sottostante. Queste limitazioni permettono ad App Engine di distribuire le richieste web per l'applicazione su più server e di avviare e fermare i server per soddisfare le esigenze di traffico. La sandbox isola la propria applicazione nel suo ambiente sicuro e affidabile che è indipendente dall'hardware, dal sistema operativo e dalla posizione fisica del server web.

Ecco alcune limitazioni dell'ambiente sicuro della sandbox:

- L'applicazione può accedere ad altri computer su Internet attraverso appositi URL e servizi e-mail. Gli altri computer possono collegarsi all'applicazione solo facendo una richiesta HTTP (o HTTPS) sulle porte standard
- Un'applicazione non può scrivere un file di sistema. Un'applicazione può leggere file, ma solo i file caricati con il codice dell'applicazione. L'applicazione deve utilizzare lo storage dati, la cache o altri servizi App Engine per tutti i dati che persistono tra le richieste
- Il codice dell'applicazione viene eseguito solo in risposta ad una richiesta web, a un'operazione in una task queue o a un'operazione pianificata, e deve restituire una risposta entro 30 secondi in ogni caso. Il gestore della richiesta non può generare un sotto-processo o eseguire il codice dopo che la risposta è stata inviata.

App Engine offre un potente servizio di archivio dati caratterizzato da un motore di query e transazioni. Proprio come il server web cresce con il traffico, così l'archivio cresce con i dati.Il datastore di App Engine non è come un database relazionale tradizionale. Gli oggetti, o "entità", hanno una tipo ed un insieme di proprietà. Le query possono recuperare le entità filtrate e ordinate in base alle proprietà. I valori delle proprietà possono essere di qualsiasi tipo supportato.Le entità del datastore sono di tipo "non relazionale".

Per sviluppare applicazioni Groovy in Google App Engine può essere utile ricorrere all'utilizzo di Gaelyk , un ambiente di sviluppo creato proprio per facilitare questo compito.

Ecco alcune applicazioni web in GAE che rendono l'idea delle potenzialità di Groovy con questo framework. Per prima cosa bisogna modificare il file web.xml come segue, per dire all'applicazione di usare script Groovy:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5">
    <servlet>
        <servlet-name>GroovyServlet</servlet-name>
        <servlet-class>groovy.servlet.GroovyServlet</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>HelloServlet</servlet-name>
        <servlet-class>HelloServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>GroovyServlet</servlet-name>
        <url-pattern>*.groovy</url-pattern>
    </servlet-mapping>
```

```
    <servlet-mapping>
        <servlet-name>HelloServlet</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>
</web-app>
```

4.1.1 Applicazione 1: Manipolatore di immagini

Per prima cosa bisogna scrivere la view , ovvero la grafica che si vuole presentare all'utente. Si presenta un form con delle semplici opzioni di modifica di immagini:

```
html.html {
 head {
     title 'Sample to Resize Images on Google App Engine'
 }
 body {
     h1 'Sample to Resize Images on Google App Engine'
     form action: 'changeImage.groovy', method: 'post', enctype: 'multipart/form-data', {
         label 'Photo: ', {
            input type: 'file', name: 'photo'
        }
br()
     label 'Flip horizontally', {
         input type: 'checkbox', name:'flipHorizontal', checked:true
     }
br()
     label 'Flip vertically', {
         input type: 'checkbox', name:'flipVertical'
         }
     br()
     label 'Rotate degress clockwise: ', {
         select name: 'rotation', {
         option '0'
         option '90'
         option '180'
         option '270'
         }
     }
     br()
     label "I'm feeling lucky", {
         input type: 'checkbox', name:'feelLucky'
     }
     hr()
     label 'Size: ', {
     select name: 'imageSize', {
         option 'thumb'
         option 'medium'
         option 'large'
     }
     }
     br()
     label 'Custom Size - width', {
         input type:'text', name:'customWidth'
     }
     br()
     label 'Custom Size - height', {
```

```
            input type:'text', name:'customHeight'
        }
        br()
        input type: 'submit', name: 'submit', value: 'Resize'
        }
    }
}
```

Questa è la vista, il form che viene presentato all'utente; nel momento in cui l'utente preme il bottone submit viene chiamata la groovlet (ovvero lo script) contenuto nella classe changeImage.groovy, come dichiarato nelle prime righe del body. Il form presentato all'utente è il seguente:

Ora bisogna appunto scrivere il controller (la classe changeImage.groovy) in modo da eseguire le modifiche all'immagine desiderate e presentare una nuova view all'utente con il relativo output.

```
import org.apache.commons.io.IOUtils
import org.apache.commons.fileupload.util.Streams
import org.apache.commons.fileupload.servlet.ServletFileUpload
import com.google.appengine.api.images.Image;
import com.google.appengine.api.images.ImagesService;
import com.google.appengine.api.images.ImagesServiceFactory;
import com.google.appengine.api.images.Transform;
import com.google.appengine.api.images.CompositeTransform;

/* gets image upload.*/
uploads = [:]  // Store result from multipart content.
if (ServletFileUpload.isMultipartContent(request)) {
 def uploader = new ServletFileUpload()
 def items = uploader.getItemIterator(request)
 while (items.hasNext()) {
     def item = items.next()
     def stream = item.openStream()
     try {
         if (item.formField) {  // 'Normal' form field.
         params[item.fieldName] = Streams.asString(stream)
```

```
} else {
         uploads[item.fieldName] = [
         name: item.name,
         contentType: item.contentType,
         data: IOUtils.toByteArray(stream)]
         }
     } finally {
     IOUtils.closeQuietly stream
     }
 }
}
/* end upload processing */

if (params.submit) {

// Gaelyk provides a shortcut to the image service, but all the heavy lifting is done by the
ImagesServiceFactory.

 // get original image
 Image pic = ImagesServiceFactory.makeImage( uploads['photo'].data )

 // we can use a composite transform to store a series of transformations
 CompositeTransform cp = ImagesServiceFactory.makeCompositeTransform()

 // add a horizontal flip
 if( params.flipHorizontal == 'on'){
     cp.concatenate( ImagesServiceFactory.makeHorizontalFlip() )
 }

 // vertical flip
 if( params.flipVertical == 'on'){
     cp.concatenate( ImagesServiceFactory.makeVerticalFlip() )
 }

 // rotate image - note, this can only happen in increments of 90 degrees
 if( params.rotation ){
     cp.concatenate( ImagesServiceFactory.makeRotate( params.rotation as int ) )
 }

 // I'm feeling lucky - which provides image balancing
 if( params.feelLucky == 'on' ) {
     cp.concatenate( ImagesServiceFactory.makeImFeelingLucky() )
 }

 // resizing tranformations
 Transform resize
// first deal with cases where there is no custom width/ height specified
 if( !params.customWidth ){
     switch( params.imageSize ){
     case 'thumb':
     resize = ImagesServiceFactory.makeResize(50,50);
     break;
     case 'medium':
     resize = ImagesServiceFactory.makeResize(200,200);
     break;
     case 'large':
     resize = ImagesServiceFactory.makeResize(400,400);
     break;
     }
```

```
 } else {

 // handle case where there is a specified height / width

 resize = ImagesServiceFactory.makeResize( params.customWidth as int, params.customHeight as
int)
 }
 cp.concatenate( resize )

 // apply all the transformations to the original image
 // images is a Gaelyk shortcut for imageService. Personally, I think it should map to
imagesServiceFactory
 pic = images.applyTransform( cp, pic )

 // set response type
 response.setContentType( uploads['photo'].contentType )

 // render image out
 sout << pic.imageData
}
```

La prima parte dello script semplicemente prende l'immagine caricata e ne fornisce un riferimento; la seconda parte effettua le trasformazioni a seconda delle direttive dell'utente , attraverso API fornite e fornisce in risposa una pagina web con l'immagine modificata.

Ad esempio:

Immagine originale

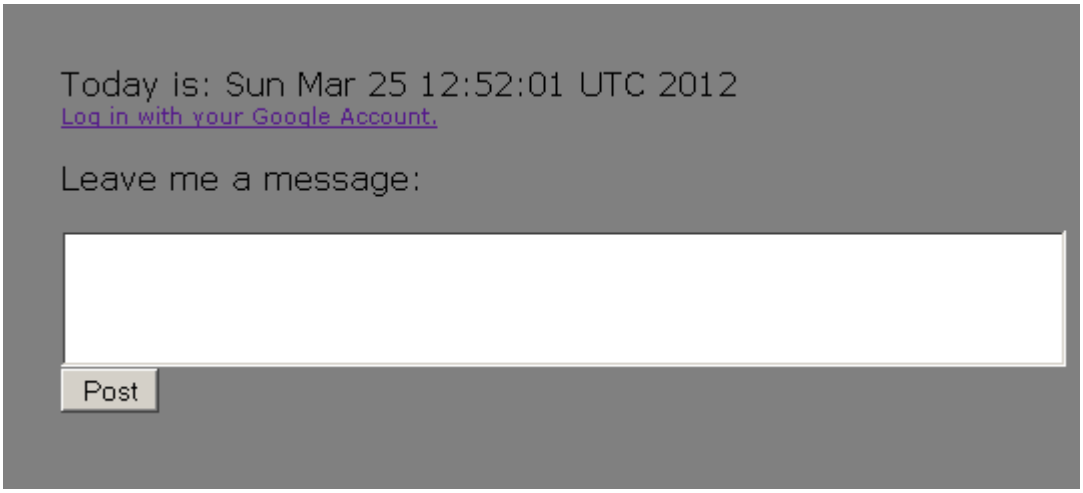
Immagine con alcune modifiche

4.1.2 Applicazione 2: Guestbook

Si definisce la view:

```
import com.google.appengine.api.users.User
import com.google.appengine.api.users.UserService
import com.google.appengine.api.users.UserServiceFactory
import javax.jdo.PersistenceManager
import guestbook.PMF
import guestbook.Greeting

UserService userService = UserServiceFactory.getUserService()
User u = userService.getCurrentUser()

PersistenceManager pm = PMF.get().getPersistenceManager()
String query = "select from " + Greeting.class.getName() + " order by date desc range 0,25"
List<Greeting> greetings = (List<Greeting>) pm.newQuery(query).execute()
html.html {
 head {
  title "Hello"
  link(type:"text/css", rel:"stylesheet", href:"/stylesheets/main.css")
 }
 body {
  div(class: "main"){
   div ("Today is: ${new Date()}")

   if (u == null) {
    div(class:"login"){
     a (href: userService.createLoginURL(request.getRequestURI()) , "Log in with your Google
Account.")
    }
   }
   else {
    div(class:"login"){
     span("Welcome ${u.nickname}. ")
     a (href: userService.createLogoutURL(request.getRequestURI()) , "Log out")
    }
   }

   p ("Leave me a message:")

   form(method: "POST", action: "/post.groovy"){
    div(){
     textarea(name: "content", rows: "3", cols: "60", "")
    }
    div(){
     input(type: "submit", value: "Post")
    }
   }
greetings.each{
    def user = "anonymous"
    if(it.author != null) user = it.author
    div(class:"entry-header", "On  ${it.date} ${user} wrote: ")
    div(class:"entry-body", "${it.content}")
   }
  }
 }
}
```

La prima parte semplicemente cerca i post precedenti, mostra la data corrente e offre la possibilità di login al proprio account Google se non si è già collegati e predispone l'area di testo col messaggio da lasciare. Se viene premuto il pulsante Post viene chiamato il Groovlet post.groovy. Ecco come appare la vista:



E ora il controller:

```
import com.google.appengine.api.users.User
import com.google.appengine.api.users.UserService
import com.google.appengine.api.users.UserServiceFactory
import javax.jdo.PersistenceManager
import guestbook.PMF
import guestbook.Greeting

UserService userService = UserServiceFactory.getUserService()
User user = userService.getCurrentUser()
String content = request.getParameter("content")
Date date = new Date()
Greeting greeting = new Greeting(user, content, date)

PersistenceManager pm = PMF.get().getPersistenceManager()
try {
    pm.makePersistent(greeting)
} finally {
    pm.close()
}

response.sendRedirect("/hello.groovy")
```

Che non fa altro che registrare nel database il post dell'utente e reindirizzarlo a hello.groovy ovvero la view vista nel paragrafo precedente.

4.1.3 Valutazioni

GAE effettivamente è un ottimo framework per sviluppare applicazioni web, tuttavia soffre di alcuni limiti che portano a scartarlo per lo sviluppo di questo progetto:

- è troppo legato e dipendente da Google: si vuole progettare un'applicazione indipendente e autonoma; se Google ha problemi o chiude il progetto l'applicazione viene persa con lui; si è costretti a usare servizi e risorse di Google.
- risorse limitate: man mano che si sviluppa l'applicazione, quando le dimensioni saranno troppo grandi sarà necessario acquistare comunque risorse.
- L'applicazione può accedere ad altri computer su Internet attraverso appositi URL e servizi e-mail. Gli altri computer possono collegarsi all'applicazione solo facendo una richiesta HTTP (o HTTPS) sulle porte standard
- Un'applicazione non può scrivere un file di sistema. Un'applicazione può leggere file, ma solo i file caricati con il codice dell'applicazione. L'applicazione deve utilizzare lo storage dati, la cache o altri servizi App Engine per tutti i dati che persistono tra le richieste
- Non possono essere avviati thread dall'applicazione
- Datastore non relazionale e non tradizionale: gli oggetti, o "entità", hanno una tipo ed un insieme di proprietà; c'è il limite di 1000 risultati per query.

4.2 Play! framework

Play è un veloce framework per applicazioni web in Java; è un progetto il cui codice è distribuito con licenza Apache 2.0 e la cui comunità è aperta e sempre in conttato tramite Twitter. Tra le feature di questo framework sono interessanti soprattutto hot reload, ovvero la possibilità di modificare a caldo il codice, e lo stateless model, che rende tale framework scalabile e ready per servizi RESTful. Play! framework si occupa della compilazione delle parti del codice sorgente nel momento in cui si rende necessario e le carica nella JVM, senza nemmeno richiedere un riavvio del server.

Ecco alcune caratteristiche di Play! framework:

- Permette uno sviluppo senza continue ricompilazioni, grazie ad un server locale in grado di prendersi carico dell'operazione. E' davvero incredibile come in questo modo diventa più piacevole la programmazione! Salva->F5 e via!;
- Supporta il testing, puro e buon testing, e lo rende una operazione quasi piacevole;
- Utilizza il pattern architetturale MVC (Model-View-Controller;
- Dispone di un sistema per il routing, utile per semplificare gli URL;
- Supporto di Scala;
- Play ha un sistema modulare che ne consente una facile estensione, oltre al fatto che sono già presenti utilissimi moduli, come ad esempio "CRUD" e "Secure";

- Possibilità di distribuzione attraverso diverse piattaforme, tra cui servlet JEE (Tomcat, JBoss, ecc), senza quindi utilizzare alcun contenitore esterno.

Si può dunque affermare che lo sviluppo web con Java si avvii a diventare davvero piacevole? Forse sarebbe esagerato dichiararlo con assoluta certezza, certo è che Play! framework in termini di semplicità, performance, robustezza e gradevolezza non sembra essere secondo a nessuno.

Installazione

Per installarlo basta scaricare l'archivio e decomprimerlo; il framework richiede una versione di Java 5 o superiore. All'interno si trovano i due script, per Linux e per Windows, che permettono di lavorare con Play. La creazione della prima web application è molto semplice, basta scrivere da riga di comando `play new prova` per creare tutta la struttura necessaria a una web application prova

Così facendo l'applicazione è già pronta per la programmazione e lanciando il comando `play run prova` si può vedere all'indirizzo `http://localhost:9000/` del PC cosa il framework ha creato. La struttura di questa nuova web application ha tre directory principali dove mettere il codice; sotto la cartella prova/app/ sono state create le cartelle controllers, models e views: banalmente esse si riferiscono alle corrispettive parti del paradigma MVC. Basta quindi andare ad aggiungere file, modificare quelli già esistenti e aggiungere codice per avere un'applicazione web perfettamente funzionante.

Ci si può domandare il perché si sia scelto Play! che apparentemente non supporta Groovy, ecco la risposta è che Play! è scritto e programmato interamente in Groovy, è anche questo uno dei motivi della sua facilità e semplicità d'uso. Quindi è vero che supporta solo Java per ora ma appunto essendo scritto in Groovy ci aspetta che nelle prossime versioni in uscita anche groovy venga supportato…insomma è il framework del futuro.

4.2.1 Applicazione Ipad Like

Ecco un esempio che mostra la potenza di questo framework: si vuole progettare un'applicazione web simile ad un Ipad, con alcune semplici funzionalità quali l'invio di email, la visualizzazione di video, la chat….

Le viste vanno tutte messe in app\views\Application , le immagini e i video possono essere messi in una cartella a piacimento; si parte con index.html in cui si presenta con delle icone intuitive tutte le applicazioni offerte:

```
#{extends 'main.html' /}
#{set title:'Applicazione play - Tesi magistrale' /}

<h2 >
        <a href="@{Application.Editor()}"><img src="@{'/public/images/Notepad
Icon.jpg'}"></a>
</h2>
<h2 >
```

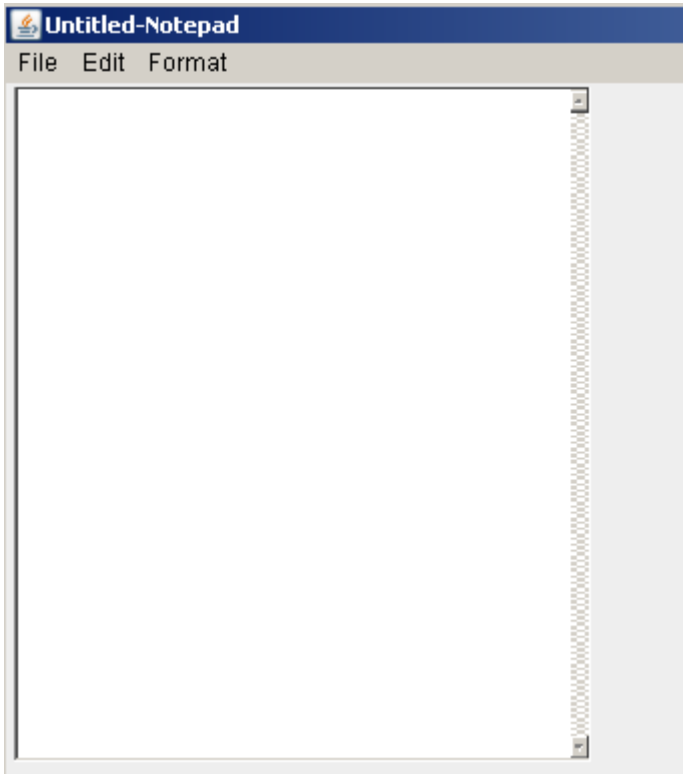
```
        <a href="@{Application.Videos()}"><img
src="@{'/public/images/windows_media_player.png'}" width='50' height='50'></a>
</h2>
<h2 >
        <a href="@{Application.Email()}"><img src="@{'/public/images/emailicon-1.jpg'}"
width='50' height='50'></a>
</h2>
<h2 >
        <a href="@{Application.Chat()}"><img src="@{'/public/images/Chat.png'}" width='50'
height='50'></a>
</h2>
```

Cliccando sull'icona si invoca il relativo metodo del controller. Nel controller vanno scritti tutti i metodi che verrano poi usati nelle viste; le viste possono solo invocare metodi del controllore.Ecco il controller principale:

```
package controllers;

import play.*;
import play.mvc.*;
import play.data.validation.*;

import java.util.*;
import java.awt.*;

import models.*;

public class Application extends Controller {

    public static void index() {
        render();
    }

    //invoca la chat a seconda del tipo di chat scelta
    public static void enterDemo(@Required String user, @Required String demo) {
        if(validation.hasErrors()) {
            flash.error("Please choose a nick name and the demonstration typeâ€¦");
            index();
        }

        // Dispatch to the demonstration
        if(demo.equals("refresh")) {
```

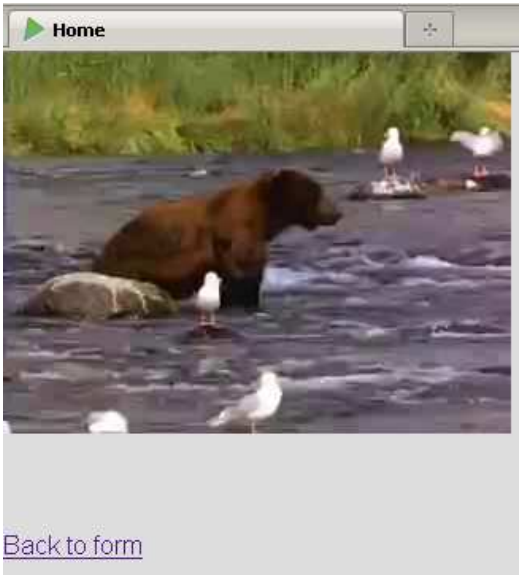
```
            Refresh.index(user);
        }
        if(demo.equals("longpolling")) {
            LongPolling.room(user);
        }
        if(demo.equals("websocket")) {
            WebSocket.room(user);
        }
    }

//invoca un'istanza dell'editor di testo
public static void Editor() {
    trail note = new trail("Untitled-Notepad");
     Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();
    note.setSize((int)dim.getWidth(),(int)dim.getHeight());
    note.setVisible(true);
    render();
}

public static void Videos() {
    render();
}

public static void Email() {
    render();
}

//metodo che invia un email
public static void sendEmail(String Mittente, String Destinatario, String Titolo,
      String     Messaggio) {
    SimpleEmail email = new SimpleEmail();
    email.setFrom(Mittente);
    email.addTo(Destinatario);
    email.setSubject(Titolo);
    email.setMsg(Messaggio);
    Mail.send(email);

    render();
}

public static void Chat(){
    render();
}
}
```

Il metodo render() semplicemente mostra all'utente la vista con il nome del metodo stesso in cui viene invocato. Il metodo sendEmail una volta ricevuti i parametri necessari invia un'email e mostra una vista e il metodo Editor crea un'istanza di un editor di testo,il cui codice viene risparmiato in quanto non significativo, viene comunque riportata qui di seguito l'immagine che mostra l'editor in esecuzione.



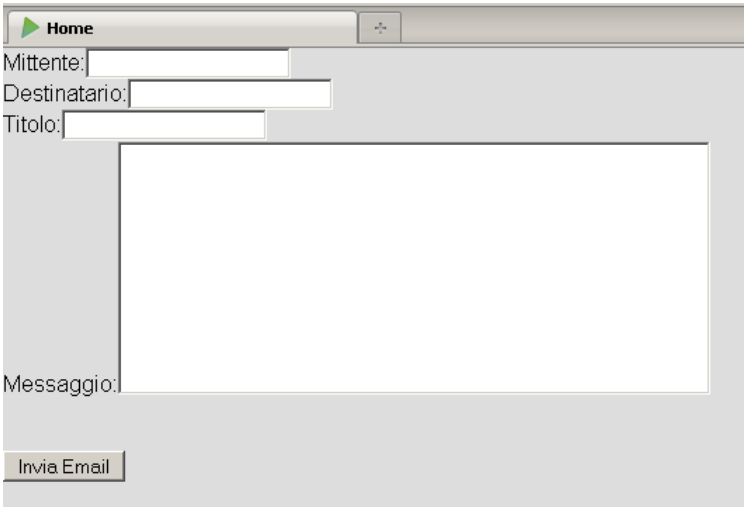
Questa view visualizza i video scelti dall'utente (in questo esempio solo uno):

```
#{extends 'main.html' /}
#{set title:'Home' /}

<video width="320" height="240" controls="controls">
  <source src="@{'/public/videos/movie.ogg'}" type="video/ogg" />
</video>
<br>
<br>
<br>
<br>
<a href="@{Application.index()}">Back to form</a>
```



Per quanto riguarda l'invio di email invece, la vista relativa con un form chiede all'utente i dati e poi invoca il metodo di invio di email nel controllore.

```
#{extends 'main.html' /}
#{set title:'Home' /}

<form action="@{Application.sendEmail()}" method="GET">
    Mittente:<input type="text" name="Mittente" /><br>
    Destinatario:<input type="text" name="Destinatario" /><br>
        Titolo:<input type="text" name="Titolo" /><br>
    Messaggio:<TEXTAREA ROWS=10 COLS=50 name="Messaggio" ></TEXTAREA>
<br><br><br>
        <input type="submit" value="Invia Email" />
</form>
<br>
<br>
<br>
<br>
<a href="@{Application.index()}">Back to form</a>
```

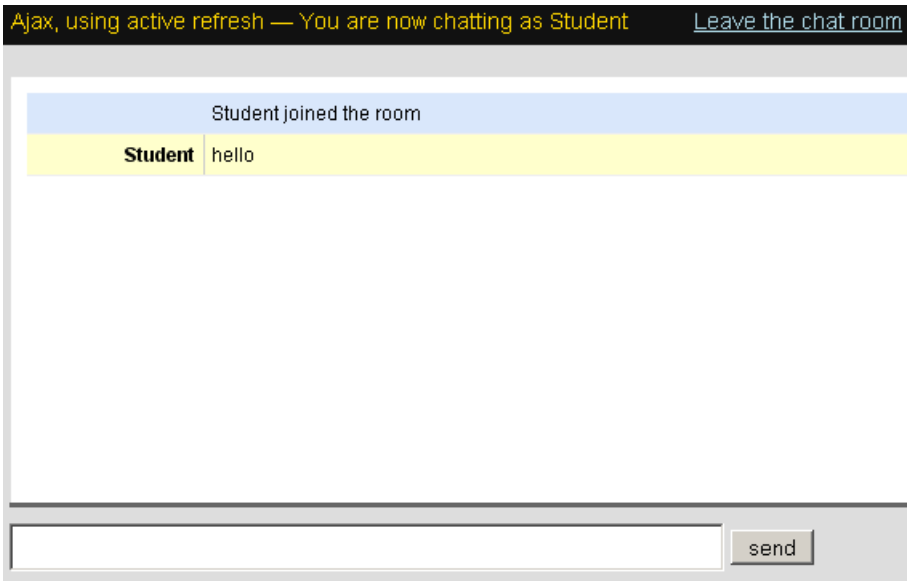


Cliccando sul pulsante Invia Email, viene invocato il metodo sendEmail del controllore che riceve come parametri i valori delle caselle di testo e invia l'email.

Infine la chat , un po' più complicata e lunga dal punto di vista della quantità di codice; si riportano quindi le parti più significative. La prima vista richiede alcuni parametri all'utente.

```
<div id="home">
    <div id="signin">
        #{form @enterDemo()}
            #{if flash.error}
                <p class="error">
                    ${flash.error}
                </p>
            #{/if}
            <p>
                <label for="nick">Nick name</label>
                <input type="text" name="user" id="user">
            </p>
            <p>
                <label for="nick">Tipo di chat</label>
                <select name="demo">
                    <option></option>
                    <option value="refresh">Ajax, active refresh</option>
                    <option value="longpolling">Ajax, long polling</option>
                    <option value="websocket">WebSocket</option>
                </select>
            </p>
            <p>
                <label></label>
                <input type="submit" id="enter" value="Entra nella chat!">
            </p>
        #{/form}
    </div>
</div>
<br><br>
<a href="@{Application.index()}">Back to form</a>
```

Una volta cliccato sul pulsante Entra nella Chat! , viene invocato il metodo del controllore enterDemo() che avvia la chat (attraverso funzionalità di polling e socket). Il risultato è il seguente:



4.2.2 Valutazioni

GAE effettivamente è un ottimo framework per sviluppare applicazioni web, tuttavia soffre di alcuni limiti che portano a scartarlo per lo sviluppo di questo progetto:

- è troppo legato e dipendente da Google: si vuole progettare un'applicazione indipendente e autonoma; se Google ha problemi o chiude il progetto l'applicazione viene persa con lui; si è costretti a usare servizi e risorse di Google.
- risorse limitate: man mano che si sviluppa l'applicazione, quando le dimensioni saranno troppo grandi sarà necessario acquistare comunque risorse.

- L'applicazione può accedere ad altri computer su Internet attraverso appositi URL e servizi e-mail. Gli altri computer possono collegarsi all'applicazione solo facendo una richiesta HTTP (o HTTPS) sulle porte standard
- Un'applicazione non può scrivere un file di sistema. Un'applicazione può leggere file, ma solo i file caricati con il codice dell'applicazione. L'applicazione deve utilizzare lo storage dati, la cache o altri servizi App Engine per tutti i dati che persistono tra le richieste
- Non possono essere avviati thread dall'applicazione
- Datastore non relazionale e non tradizionale: gli oggetti, o "entità", hanno una tipo ed un insieme di proprietà; c'è il limite di 1000 risultati per query.

4.3 Griffon framework

Griffon è un framework applicativo simile a Grails, per lo sviluppo di applicazioni in JVM, con Groovy come linguaggio principale di programmazione. Ispirato da Grails, Griffon segue il paradigma della Convention over Configuration, in coppia con un'architettura MVC intuitiva e un'interfaccia a riga di comando. Griffon segue anche lo spirito del'Application Framework Swing, definisce un semplice ma potente ciclo di vita dell'applicazione e meccanismo ad eventi. Un'altra caratteristica interessante deriva dal linguaggio Groovy stesso: il supporto automatico delle proprietà e il binding delle proprietà. Come se il binding delle proprietà non è sufficiente, lo SwingBuilder di Groovy semplifica anche la costruzione di applicazioni multi-threaded, dicendo addio al rettangolo brutto grigio (la rovina di applicazioni di Swing)!

Griffon sfrutta anni di esperienze e lezioni apprese dagli sviluppatori Grails, Groovy, Rails, Ruby, Java Desktop, e Java e le loro comunità. Griffon prende i loro stessi approcci: Model-View-Controller, convention-over-configuration, una lingua moderna e dinamica (Groovy), il domain specific languages (DSL) e il modello costruttore.

Gli sviluppatori di Grails dovrebbero sentirsi a proprio agio quando provano Griffon. Molte delle convenzioni e dei comandi Grails sono condivise con Griffon. Certo, Swing non è lo stesso del formato HTML / GSP, ma i Builders semplificano il compito di creare l'interfaccia utente.
Gli sviluppatori Java saranno anche in grado di accelerare il ritmo di apprendimento in fretta, in quanto il framework allevia l'onere di mantenere una struttura dell'applicazione, permettendo di concentrarsi su come ottenere il codice giusto.

Il framework Griffon è estensibile tramite plugin. Ce ne sono molti tra cui scegliere. Per esempio ci sono plugin per i componenti di Swing come Swingx, Jide e Macwidgets; plugin legati alla persistenza come DataSource, GSQL, Ebean e db4o tra gli altri, grafica 3D e il supporto di animazione tramite JOGL, LWJGL ed elaborazione. E molti altri!

4.3.1 Applicazione navigazione di documenti

Ecco un esempio di applicazione in Griffon: un navigatore della documentazione Java in Netbeans (può essere facilmente esteso a navigatore del file system); ovviamente Griffon seguendo il paradigma MVC richiede la scrittura di queste tre parti.

Si parte dalla view:

```
actions {
    action( id: 'collapseAllAction',
        name: "Collapse all",
        closure: controller.collapseAll,
        accelerator: shortcut('C'),
        mnemonic: 'C',
        shortDescription: "Collapse all categories",
        smallIcon: imageIcon("go-first.png")
    )

    action( id: 'expandAllAction',
        name: "Expand all",
        closure: controller.expandAll,
        accelerator: shortcut('E'),
        mnemonic: 'E',
        shortDescription: "Expand all categories",
        smallIcon: imageIcon("go-last.png")
    )
}

application( title: "Griffon Demo", size: [250,300], locationByPlatform: true ) {
    panel(  ) {
        borderLayout()
        scrollPane( constraints: CENTER ) {
            jxtree( id: "topics" )
        }
        toolBar( constraints: SOUTH ) {
            button( action: collapseAllAction )
            button( action: expandAllAction )
        }
    }
}
```

Questa è una semplice vista contenente un pannello (dove si potrà esplorare il file system) e due bottoni a cui sono associate due azioni, definite nella parte superiore, che si riferiscono a due metodi contenuti nel controllore. Di per sé questo è poco significativo , infatti ora si riporta il codice del controllore che è un po' il succo dell'applicazione:

```
import javax.swing.tree.*

class GriffonDemoController {

    def model
    def view

    def expandAll = { evt ->
```

```
        ViewUtils.expandTree( view.topics )
    }

    def collapseAll = { evt ->
        ViewUtils.collapseTree( view.topics )
    }


    def loadPages() {
        doOutside {
            def contents = new DefaultMutableTreeNode("NetBeans Javadoc")
            def leafNodes = new URL(model.menuUrl).text
            def lastCategory = null
            (leafNodes =~ /href="(([a-zA-Z-]+)\/(.+?)\.html)"/).each { match ->
                def category = new DefaultMutableTreeNode(match[2])
                def pageNode = new PageNode( title: match[3] )
                if( lastCategory?.toString() == category.toString() ){
                    lastCategory.add( new DefaultMutableTreeNode(pageNode) )
                }else{
                    lastCategory = category
                    category.add( new DefaultMutableTreeNode(pageNode) )
                    contents.add( category )
                }
            }
            doLater {
                view.topics.model = new DefaultTreeModel(contents)
            }
        }
    }
}
```

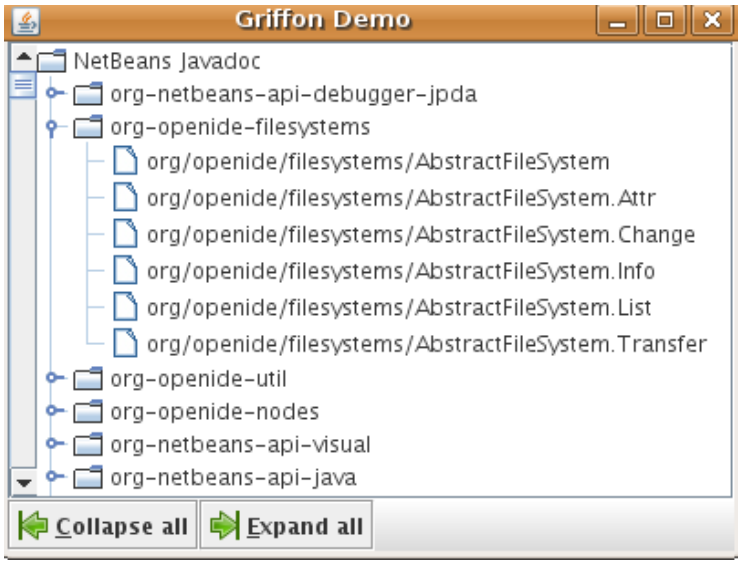
I primi due metodi semplicemente espandono o chiudono il nodo di un albero e utilizzano un'utiliti presente nelle API; l'ultimo metodo è quello che si occupa di inizializzare l'albero,costruendolo, e presentare all'utente una prima vista di default dell'albero; questo metodo non viene invocato esplicitamente dalla vista ma, modificando semplicemente un file di configurazione, viene eseguito all'avvio dell'applicazione.

E infine il modello molto semplice:

```
class GriffonDemoModel {
    String baseUrl = "http://bits.netbeans.org/dev/javadoc/"
    String menuUrl = baseUrl + "allclasses-frame.html"
}
```

Che non fa altro che definire il valore iniziale delle due variabili.

Il risultato finale è il seguente:



4.3.2 Valutazioni

Griffon è il framework che cercavamo per questo progetto! Groovy è il suo linguaggio principale, permette di sviluppare una buona grafica, implementa il modello MCV che rende più semplice la programmazione, è compatibile con Grails, è affermato da tempo e non presenta difetti degni di nota che costringano a scartarlo. L'applicaione finale verrà realizzata con griffon.

NOTA: tutti questi esempi in Groovy mostrano come con questo linguaggio in poche righe sia possibile scrivere applicazioni che in Java richiederebbero molte più righe di codice.

5 L'applicazione finale

In questo capitolo viene presentata l'applicazione finale, a completamento del percorso iniziato nel capitolo 1, ovvero dopo aver selezionato e illustrato tutti gli strumenti necessari a sviluppare un'applicazione web in Groovy.

Questo capitolo spiega in dettaglio le componenti sviluppate e riporta i risultati ottenuti; l'applicazione finale, perfettamente funzionante, consente la gestione di un archivio di documenti, (esercizi,ecc…) contenuti in un file compresso Zip.

In particolare si mostra come vengono implementate le operazione di visualizzazione e modifica di documenti attraverso un'interfaccia grafica; di accesso all'archivio compresso e modifica del contenuto dei file contenuti nello stesso.

Arrivati alla fine, sarà possibile vedere le potenzialità di Groovy (peraltro già intuibibili nei capitoli precedenti) che con poche righe di codice permette di sviluppare applicazioni importanti; a differenza di altri linguaggi che richiederebbero molto più codice; Groovy, per quanto innovativo nell'ambito delle applicazioni web, si rivelerà un'ottima scelta per chiunque sia intenzionato a sviluppare applicazioni e voglia un linguaggio altamente efficienteed espressivo.

Il capitolo si apre con una breve presentazionezione,degli aspetti principali da sviluppare, successivamente si si esaminano le operazioni preliminari necessarie (che possono essere eseguite manualmente oppure automatizzate tramite programmi); qundi si analizzano le componenti principali dell'applicazione, ovvero il database contenente informazioni per la ricerca dei documenti, poi le operazioni di elaborazione su archivio compresso (per cui è necessario il supporto di Java) e infine la grafica, ovvero l'interfaccia utente.

Il capitolo si conclude con alcune riflessioni finali sul risultato ottenuto ed alcune indicazioni su possibili miglioramenti del prototipo e sviluppo di nuove funzionalità.

5.1 Richiami

Come già detto,ora si vuole sviluppare un'applicazione per l'archiviazione, la catalogazione, la ricerca e la modifica di documenti, dove per documento si intende

un'entità complessa, costituita di più parti tra loro correlate e strutturate secondo un modello stabilito.

L'insieme dei documenti è contenuto in un archivio compresso nel formato zip e su di essi si possono eseguire operazioni di lettura e modifica, mentre per la ricerca del documento viene usato un database che per ognuno contiene il genere dell'argomentro trattato nello stesso e alcune parole chiave che lo caratterizzano.

All'utente bisogna presentare un form di ricerca, con possibilità di aprire un documento, dato il nome dello stesso, ed eventualmente modificarne le parti per cui è permesso dalle direttive del prototipo.

Il primo passo per sviluppare l'applicazione è quindi quello di implementare il database e permettere a Groovy di eseguire query su di esso, passare poi all'elaborazione di archivi compressi, per visualizzazione e modifica dei file, e infine sviluppare la GUI per le interazioni con l'utente e la visualizzazione.

5.2 Il database

Il modello informativo implica lo sviluppo di un database composto da più campi, alcuni mono valore, altri multi valore. Per ogni campo occorre un Dizionario (insieme de valori possibili), le ricerche possono essere incomplete e non ritornare alcun risultato ed i risultati pesati in base alla pertinenza del documento con le parole chiave cercate. Inoltre le ricerche dovrebbero essere definite in modo unificato (interfaccia) e realizzate in modo via via più complesso e raffinato (ad esempio ricercha approssimata del titolo).

Trattandosi di un primo prototipo che ha scopo illustrativo si sceglie di utilizzare un database formato da un'unica tabella con alcuni campi.

Il database deve contenere per ogni documento ovviamente il titolo ufficiale (nome), alcune parole chiave (tra cui il genere dell'argomento trattato nel documento, la complessità, la tecnica, etc); come prima implementazione si decide di quantificare il numero di parole chiave nel numero di due. Successivamente verrà proprosto un miglioramento.

Il database si presenterà quindi come una tabella con i seguenti campi:

- filename : ovvero il nome del documento
- genere : ovvero il genere dell'argomento trattato nel documento
- keyword1: ovvero la prima parola chiave per ricercare il documento nel database
- keyword2: ovvero la seconda parola chiave per ricercare il documento nel database

Su questo database verranno effettuate le interrogazioni per ritrovare i nomi dei documenti che corrispondono alle caratteristiche cercate (i parametri di ricerca verranno ricevuti dai form dell'interfaccia grafica), utilizzando delle query con le clausole SELECT-FROM-WHERE (query più complesse non sono necessarie e verrano proposte nel miglioramento del database).

Ora, vi sono più modi per permettere a Griffon di interagire con un database e connettersi ad esso per poi effetturare le interrogazioni in Groovy, fra cui installare MySQL, impostare un piccolo server ad hoc e infine connettersi ad esso dall'applicazione. Fortunatamente Griffon giunge in aiuto con uno dei suoi innumerevoli plugin: GSQL.

Il plugin GSQL consente l'accesso alle funzionalità di database in modo semplice, creando un'istanza di database senza aver bisogno di un programma esterno … è tutto integrato in Griffon e nell'applicazione! Questo plugin non fornisce classi di dominio e cercatori dinamici come fa GORM.

GSQL fornisce quindi supporto, in modo facile, nell'accesso al database se si utilizza Groovy. Per installare GSQL basta eseguire il comando da console:

```
griffon install-plugin gsql
```

Una volta installato il plugin genererà i seguenti file in `$appdir/griffon-app/conf` :

- `DataSource.groovy` - contiene il datasource e le definizioni
- `BootstrapGsql.groovy` – definisce informazioni di inizializzazione / distruzione per i dati che devono essere manipolati durante l'avvio o lo spegnimento dell'applicazione.

Un nuovo metodo dinamico di nome `withSql` verrà iniettato in tutti i controller, il quale darà accesso a un oggetto groovy.sql.Sql, con il quale si sarà in grado di effettuare chiamate al database.

Una volta installato GSQL, bastano pochi passi per avere il database perfettamente funzionante. Per prima cosa, una volta stabilito che database si vuole, bisogna creare lo schema del database; per fare questo bisogna creare un file in `griffon-app/resources/` chiamato `schema.ddl` con le direttive dello schema. Per l'applicazione che si vuole progettare lo schema risulta essere il seguente:

```
DROP TABLE IF EXISTS docs;
CREATE TABLE docs(
   filename VARCHAR(30) NOT NULL PRIMARY KEY,
   genre VARCHAR(30) NOT NULL,
   keyword1 VARCHAR(30),
   keyword2 VARCHAR(30)
);
```

Ovviamente con il nome del file come chiave primaria.

Una volta completato questo passo, all'avvio dell'applicazione il database verrà creato, anche se vuoto. Per inizializzarlo bisogna modificare il file BootstrapGsql.groovy inserendo i dati del database; in questa applicazione si suppone di inserire alcuni dati riguardanti il corso di Sistemi Operativi, giusto alcune entry per provare l'applicazione, il file risultante conterrà il seguente codice (ed esempi):

```
import groovy.sql.Sql

class BootstrapGsql {
    def init = { String dataSourceName = 'default', Sql sql ->
        def docs = sql.dataSet('docs')
```

```
        docs.add(filename: 'semafori', genre: 'concorrenza', keyword1: 'concorrenza',
keyword2: 'semafori')
        docs.add(filename: 'ada', genre: 'programmazione', keyword1: 'ada', keyword2:
'programmazione')
        docs.add(filename: 'petri', genre: 'reti di petri', keyword1: 'petri',
keyword2:'reti')
        docs.add(filename: 'messaggi', genre: 'concorrenza', keyword1: 'programmazione',
keyword2: 'messaggi')
    }

    def destroy = { String dataSourceName = 'default', Sql sql ->
    }
}
```

L'ordine delle parole chiave è indifferente ai fini del funzionamento dell'applicazione.

Dopo questo passo, all'avvio dell'applicazione si avrà un database funzionante e popolato con i dati inseriti su cui sarà possibile eseguire query per ottenere i risultati da fornire all'utente. Il tipo di query sarà appunto una ricerca sulle parole chiave in base al genere (al tema) trattato nel documento-esercizio.

Groovy permette di interrogare i database in modo semplice ed efficace con poche righe di codice; il fatto di usare GSQL semplifica di molto il codice oltretutto.

Quindi ora, dopo aver completato il setup del database, si passa al codice per l'interrogazione del database, che è il seguente:

```
withSql { dsName, sql ->
          def tmpList = []
          sql.eachRow("SELECT * FROM docs WHERE (keyword1 = '${model.keyword1}' OR keyword2
= '${model.keyword1}' OR keyword1 = '${model.keyword2}' OR keyword2 = '${model.keyword2}')
AND genre = 'reti di petri' " ) {
                tmpList << [filename: it.filename,
                            genre: it.genre]
            }
            execSync { model.docsList.addAll(tmpList) }
       }
```

Questa interrogazione viene eseguita su ogni riga del database col comando `eachRow` e ritorna la riga se rispetta i parametri della query, la notazione `'${model.variabile}'` indica il valore di una variabile (campo di testo, checkbox, ecc…) nella vista (form) presentata all'utente e che verrà discussa e vista successivamente.

`tmpList` è una lista temporanea dove vengono messi nome del file e genere del documento della riga in cui si ha la corrispondenza dei parametri, mentre `docsList` è la lista che poi viene presentata all'utente nella vista e che viene poi ordinata alfabeticamente. Questo codice verrà poi inserito nel controllore , mentre le variabili con riferimento del `$` verranno dichiarate nel modello.

Vi sono due modi per popolare il database: può essere popolato manualmente, come mostrato precedentemente, oppure può esser fatto eseguire come operazione preliminare un semplice programma di ricerca di parole chiave all'interno di un documento, che si occuperà di trovare le stesse e inserirle nel database (ad esempio riscrivendo il file di configurazione). Per far ciò si può usare JFlex, per maggiori informazioni visitare il sito ufficiale http://jflex.de/ .

Un modello più raffinato di database, per aggiungere un numero arbitrario di parole chiave può essere implementato con due tabelle: la prima con il nome del file, come chiave primaria, e il genere e la seconda con il nome del file , come chiave esterna, e la parola chiave associata al file ,in questo caso la coppia è la chiave primaria. Risulta un database del tipo:

```
DROP TABLE IF EXISTS docs;
CREATE TABLE docs(
   filename VARCHAR(30) NOT NULL PRIMARY KEY,
   genre VARCHAR(30) NOT NULL,
);
DROP TABLE IF EXISTS keywords;
CREATE TABLE keywords(
   filename VARCHAR(30) NOT NULL,
   keyword VARCHAR(30) NOT NULL,
   PRIMARY KEY(filename,keyword),
   FOREIGN KEY(filename) REFERENCES docs(filename)
);
```

Mentre la query per ottenere i file che rispondono a determinate parole chiave è:

```
SELECT * FROM docs d,keywords k WHERE k.keyword = '${model.keyword}' AND genre = '//genere
scelto nel form' AND d.filename=k.filename"
```

Cosi facendo si può avere un numero arbitrario di parole chiave per ogni file.

Questo completa la parte sul database, che è il primo passo nello sviluppo dell'applicazione.

5.3 L'archivio compresso

L'archivio compresso consiste in un file nel formato zip (ma si può facilmente modificare l'applicazione in modo da lavorare con i file jar) , anche se non necessariamente di questa estensione, che contiene i documenti che si intende visionare ed eventualmente modificare.

Nelle direttive di progetto del prototipo di applicazione ogni documento è stato considerato come diviso in quattro parti; si stabilisce un nome per il documento (ad esempio "java") e si è scelto (in fase di progettazione) di strutturare la stringa rappresentante il nome del file contenente ogni parte che compone il documento nel modo seguente: con il nome stabilito per il documento una prima parte della stringa, l'identificativo della parte appeso subito dopo la prima parte della stringa in modo da permettere di riconoscere di che parte si tratta; si sceglie come identificativo da appendere la stringa "_partx" dove x è il numero della parte (ad esempio "_part1" o "_part2"); il formato del file è indifferente purchè sia di testo, per semplicità in quest'applicazione si useranno file con estensione txt.

Il modello per rappresentare un documento è il seguente:

- parte 1: autore,data,ecc… (non modificabile)
- parte 2: testo dell'esercizio (non modificabile)
- parte 3: codice della soluzione (modificabile)
- parte 4: commenti aggiuntivi (modificabile)

A questo archivio l'applicazione andrà ad accedere una volta che viene dato un nome di file da aprire ,se il file non esiste ovviamente non si aprirà nulla, andando a:

1) prendere i quattro file di cui è composto il documento;
2) leggendone il contenuto;
3) mostrandolo in una vista appropriata;
4) consentire eventualmente di modificare le due parti riguardante codice e commento;
5) aggiornare i file dell'archivio andando a scrivere le modifiche apportate dall'utente.

Questo capitolo si occupa dei punti 1,2 e 5 ; gli altri punti verranno discussi successivamente.

Groovy non offre supporto per la manipolazione di file zip e quindi si è costretti a ricorrere all'aiuto di Java; la bontà di Groovy in questo caso si manifesta nel fatto che si può scrivere la classe che eseguirà le operazioni sul file zip in Java, per poi inserire e utilizzare questo codice nel programma Groovy senza alcun problema, come se esso stesso fosse scritto in Groovy; è perfettamente compatibile.

Per prima cosa bisogna creare l'archivio zip; per prova si inseriscono quattro file di testo riguardanti ada ed un altro casuale, per verificare che il programma operi solo su i file scelti dall'utente e lasci invariati gli altri; una volta predisposti i file di testo, bisogna creare l'archivio zip e andare a posizionarlo in `$app/staging` altrimenti non verrà trovato dall'applicazione; nell'esempio il nome associato all'archivio è "archivio.zip" ma va bene qualsiasi altro nome, anche con estensione diversa purchè compresso.

Quest'archivio va creato come operazione preliminare: l'utente deve occuparsi di scegliere i documenti che vuole vengano gestiti con quest'applicazione e creare l'archivio; in modo analogo all'inserimento delle parole chiave nel database, la creazione dell'archivio è un'operazione preliminare.

Una volta creato l'archivio si può iniziare a manipolarlo: in Java per manipolare i file zip servono le classi ZipFile, ZipEntry e ZipOutputStream, si riporta di seguito la documentazione ufficiale Javadoc con i metodi usati nell'applicazione.

ZipFile, per accesso all'archivio:

This class is used to read entries from a zip file.

- public ZipFile(String name) throws IOException

  Opens a zip file for reading.

  First, if there is a security manager, its checkRead method is called with the name argument as its argument to ensure the read is allowed.

  The UTF-8 charset is used to decode the entry names and comments.

  Parameters:

  - name - the name of the zip file

  Throws:

  - ZipException - if a ZIP format error has occurred

  - IOException - if an I/O error has occurred
  - SecurityException - if a security manager exists and its checkRead method doesn't allow read access to the file.

- public ZipFile(File file) throws ZipException, IOException

  Opens a ZIP file for reading given the specified File object.

  The UTF-8 charset is used to decode the entry names and comments.

  Parameters:

  - file - the ZIP file to be opened for reading

  Throws:

  - ZipException - if a ZIP format error has occurred
  - IOException - if an I/O error has occurred

- public void close() throws IOException

  Closes the ZIP file.

  Closing this ZIP file will close all of the input streams previously returned by invocations of the getInputStream method.

- public Enumeration<ZipEntry> entries()

  Returns an enumeration of the ZIP file entries.

- protected void finalize() throws IOException

  Ensures that the system resources held by this ZipFile object are released when there are no more references to it.

  Since the time when GC would invoke this method is undetermined, it is strongly recommended that applications invoke the close method as soon they have finished accessing this ZipFile. This will prevent holding up system resources for an undetermined length of time.

- public String getComment()

  Returns the zip file comment, or null if none.

- public ZipEntry getEntry(String name)

  Returns the zip file entry for the specified name, or null if not found.

- public InputStream getInputStream(ZipEntry entry) throws IOException

  Returns an input stream for reading the contents of the specified zip file entry.

  Closing this ZIP file will, in turn, close all input streams that have been returned by invocations of this method.

- public String getName()

  Returns the path name of the ZIP file.

- public int size()

  Returns the number of entries in the ZIP file.

ZipEntry,per la manipolazione del singolo elemento dell'archivio:

This class is used to represent a ZIP file entry.

- public ZipEntry(String name)

  Creates a new zip entry with the specified name.

  Parameters:

  - name - the entry name

  Throws:

  - NullPointerException - if the entry name is null
  - IllegalArgumentException - if the entry name is longer than 0xFFFF bytes

- public Object clone()

  Returns a copy of this entry.

- public String getComment()

  Returns the comment string for the entry, or null if none.

- public long getCompressedSize()

  Returns the size of the compressed entry data, or -1 if not known. In the case of a stored entry, the compressed size will be the same as the uncompressed size of the entry.

- public long getCrc()

  Returns the CRC-32 of the uncompressed entry data, or -1 if not known.

- public byte[] getExtra()

  Returns the extra field data for the entry, or null if none.

- public int getMethod()

  Returns the compression method of the entry, or -1 if not specified.

- public String getName()

  Returns the name of the entry.

- public long getSize()

  Returns the uncompressed size of the entry data, or -1 if not known.

- public long getTime()

  Returns the modification time of the entry, or -1 if not specified.

- public int hashCode()

  Returns the hash code value for this entry.

- public boolean isDirectory()

  Returns true if this is a directory entry. A directory entry is defined to be one whose name ends with a '/'.

- public void setComment(String comment)

  Sets the optional comment string for the entry.

  ZIP entry comments have maximum length of 0xffff. If the length of the specified comment string is greater than 0xFFFF bytes after encoding, only the first 0xFFFF bytes are output to the ZIP file entry.

- public void setCompressedSize(long csize)

  Sets the size of the compressed entry data.

- public void setCrc(long crc)

  Sets the CRC-32 checksum of the uncompressed entry data.

- public void setExtra(byte[] extra)

  Sets the optional extra field data for the entry.

- public void setMethod(int method)

  Sets the compression method for the entry.

- public void setSize(long size)

  Sets the uncompressed size of the entry data.

- public void setTime(long time)

  Sets the modification time of the entry.

- public String toString()

  Returns a string representation of the ZIP entry.

ZipOutputStream, per la scrittura dell'archivio dopo la modifica dei file, i metodi:

This class implements an output stream filter for writing files in the ZIP file format. Includes support for both compressed and uncompressed entries.

- public void close() throws IOException

  Closes the ZIP output stream as well as the stream being filtered.

- public void closeEntry() throws IOException

  Closes the current ZIP entry and positions the stream for writing the next entry.

- public void finish() throws IOException

  Finishes writing the contents of the ZIP output stream without closing the underlying stream. Use this method when applying multiple filters in succession to the same output stream.

- public void putNextEntry(ZipEntry e) throws IOException

  Begins writing a new ZIP file entry and positions the stream to the start of the entry data. Closes the current entry if still active. The default compression method will be used if no compression method was specified for the entry, and the current time will be used if the entry has no set modification time.

- public void setComment(String comment)

  Sets the ZIP file comment.

- public void setLevel(int level)

  Sets the compression level for subsequent entries which are DEFLATED. The default setting is DEFAULT_COMPRESSION.

- public void setMethod(int method)

  Sets the default compression method for subsequent entries. This default will be used whenever the compression method is not specified for an individual ZIP file entry, and is initially set to DEFLATED.

- public synchronized void write(byte[] b,

  Writes an array of bytes to the current ZIP entry data. This method will block until all the bytes are written.

- public ZipOutputStream(OutputStream out)

  Creates a new ZIP output stream.

  The UTF-8 charset is used to encode the entry names and comments.

  Parameters:

  - out - the actual output stream

Alcuni metodi di File, per la manipolazione di file.

Ora, trovati i metodi che servono per lavorare sull'archivio si può scrivere il codice.

Operazione di lettura

La lettura di file contenuti in un archivio zip non è difficile, basta semplicemente aprire l'archivio, ottenere un puntatore al file compresso che si vuole leggere e poi utilizzare lo stream input per leggere il file. Questo può essere fatto con il seguente codice:

```
public String  read(String filename) {

ZipFile zi = new ZipFile("archivio.zip");
ZipEntry entry = zi.getEntry(filename);
int len=(int)entry.getSize();

InputStream fobj = zi.getInputStream(filename);

for(int j=0;j<len;j++)
   {
    char c=(char)fobj.read();
    //eventuale operazione sul carattere letto
    }
}
```

Le prime due righe semplicemente permettono di ottenere i riferimenti necessari, la terza la lunghezza del file compresso (per sapere quanti byte leggere prima di raggiungere la fine del file) e infine le ultime righe sono le classiche operazioni di

lettura da file; ZipFile permette di usare uno stream input per una entry esattamente come se fosse un normalissimo file.

Operazione di scrittura

La scrittura di file dentro un archivio zip è più complicata e ha un difetto intrinseco a Java: non è possibile modificare o anche solo aggiungere un file ad un archivio già esistente, almeno fino alla versione JDK 6.

Andare a scrivere in un archivio zip non comporta l'aggiunta del file che si desidera inserire ma bensì la sovrascrittura dell'archivio; in pratica ogni volta che si scrive sul file zip si cancella il contenuto dello stesso lasciando solo il nuovo file scritto.

Questo difetto può essere risolto con uno stratagemma che permette di proseguire nello sviluppo del progetto: basta creare una copia dell'archivio "on the fly" andando a copiare i file che non devono essere modificati, cosi come sono, in un nuovo archivio temporaneo ma prima di chiudere lo stesso (chiudendolo si tornerebbe al problema appena citato) andare ad inserire i file modificati; infine basterà semplicemente cancellare il vecchio archivio e rinominare quello nuovo appena creato con i file modificati.

Questo stratagemma permette di risolvere il problema che ha Java nella scrittura dei file zip. Il codice per eseguire l'operazione di scrittura è il seguente:

```
public void write (String filename,String modifica){
ZipFile zipSrc = new ZipFile("archivio.zip");
ZipOutputStream zos = new ZipOutputStream(new FileOutputStream("temp.zip"));

Enumeration srcEntries = zipSrc.entries();
while (srcEntries.hasMoreElements()) {
    ZipEntry entry = (ZipEntry) srcEntries.nextElement();
    if(//criterio di selezione dei file da copiare e che non vanno modificati){
            ZipEntry newEntry = new ZipEntry(entry.getName());
            zos.putNextEntry(newEntry);

            BufferedInputStream bis = new BufferedInputStream(zipSrc
                            .getInputStream(entry));

            while (bis.available() > 0) {
                    zos.write(bis.read());
            }

            zos.closeEntry();

            bis.close();
            }
    }

   //a questo punto i file non modificati sono stati copiati

    zos.putNextEntry(new ZipEntry(filename));
    byte[] blab;
    blab = modifica.getBytes();
    zos.write(blab, 0, blab.length);

    //finalizza il file modificato
    zos.closeEntry();
    zos.close();
    zipSrc.close();
```

```
    //cancellazione vecchio archivio e cambiamento nome di quello temporaneo
    File f = new File("archivio.zip");
    f.delete();
    File toBeRenamed = new File("temp.zip");
    File newFile = new File("archivio.zip");
    toBeRenamed.renameTo(newFile);
    }
```

Il primo ciclo copia i file dell'archivio che non devono essere modificati, successivamente si aggiunge al nuovo archivio il file modificato andando a scrivere byte per byte la stringa che rappresenta il contenuto modificato del file stesso, infine semplicemente si esegue la sostituzione del vecchio archivio con quello nuovo.

La nuova versione di Java appena rilasciata (Java 7) introduce una nuova classe per trattare i file zip esattamente come un normale file system, ovvero per eseguire semplicemente operazioni di copia,cancellazione,rinominare, ecc…; la classe in questione è Zip File System Provider. Nel momento in cui questa nuova versione di Java è stata rilasciata, il progetto era già stato completato e quindi non è stato possibile usarla ma si riporta comunque l'informazione per completezza, e di seguito un piccolo esempio su come risulta più semplice operare ora con i file zip.

```
import java.util.*;
import java.net.URI;
import java.nio.file.Path;
import java.nio.file.*;

public class ZipFSPUser {
    public static void main(String [] args) throws Throwable {
        Map<String, String> env = new HashMap<>();
        env.put("create", "true");
        // locate file system by using the syntax
        // defined in java.net.JarURLConnection
        URI uri = URI.create("jar:file:/codeSamples/zipfs/zipfstest.zip");

        try (FileSystem zipfs = FileSystems.newFileSystem(uri, env)) {
            Path externalTxtFile = Paths.get("/codeSamples/zipfs/SomeTextFile.txt");
            Path pathInZipfile = zipfs.getPath("/SomeTextFile.txt");
            // copy a file into the zip file
            externalTxtFile.copyTo(pathInZipfile);
        }
    }
}
```

Che semplicemente crea un file system zip e ci copia un file dentro.

Questo completa la parte sull'elaborazione dell'archivio contenente i documenti, che è il secondo passo nello sviluppo dell'applicazione; ancora una volta, questi metodi vengono inseriti e usati nel controllore.

5.4 La vista

Quest'ultima parte riguarda l'interfaccia grafica da presentare all'utente e costituisce l'ultimo componente dell'applicazione.

In Groovy si può creare un'interfaccia grafica attraverso la classe SwingBuilder. SwingBuilder consente di creare vera e propria GUI swing in modo dichiarativo e conciso. Lo fa utilizzando un idioma comune in Groovy,i costruttori. I costruttori gestiscono il compito pesante di creare oggetti complessi per voi, come ad esempio gestire gli elementi figli, chiamare i metodi di Swing, e collegando gli elementi figli ai loro genitori. Di conseguenza, il codice è molto più leggibile e gestibile, pur consentendo di accedere a una gamma completa di componenti Swing.

Questa gerarchia di componenti normalmente vienecreato attraverso una serie di istruzioni ripetitive, impostazioni, e infine aggancio del figlio al suo rispettivo genitore. Utilizzare SwingBuilder, tuttavia, consente di definire questa gerarchia nella sua forma nativa, il che rende il design dell'interfaccia grafica comprensibile semplicemente leggendo il codice.

La flessibilità qui è resa possibile sfruttando Groovy stesso, attraverso le sue caratteristiche, quali chiusure, chiamate implicita al costruttore, l'aliasing importato, e interpolazione di stringhe. Naturalmente, questi non devono essere pienamente compreso, al fine di utilizzare SwingBuilder.

SwingBuilder con il suo stile dichiarativo permette di ridurre di molto il codice da scrivere!

Per questa applicazione si usa una vista contenente dei campi di testo e alcuni checkbox per offrire delle opzioni di ricerca all'utente; una tabella per mostare i risultati della ricerca e infine un piccolo pannello per inserire il nome del documento da aprire. Questo si può realizzare con i seguenti costrutti (nota: mig layout serve per un migliore aspetto grafico) :

- per prendere dei valori dai campi di testo basta usare l'oggetto textField con il bind di una variabile del modello con esso per permettere al controllore di utilizzarne il valore.

```
textField(text: bind(target: model, targetProperty: 'property'),columns:20)
```

- Ora ogni metodo del controllore può accedere al valore di questo campo di testo usando la notazione `'${model.property}'`
- per i checkbox in maniera analoga si usa il bind, ma stavolta verso una variabile boolean, in modo da sapere se è stato selezionato il relativo checkbox associato oppure no

```
checkBox(text: 'test', constraints:'wrap',selected:bind(target:    model,
  targetProperty:'property') )
```

- per permettere all'utente di interagire con l'applicazione vanno aggiunti dei bottoni e va associato ad un bottone un'azione da eseguire quando esso viene premuto, questo si può fare nel modo seguente:

```
button('Search!', actionPerformed: controller.function)
```

- Dove function è la funzione che verrà eseguita una volta premuto il bottone.
- per utilizzare un valore del modello per abilitare/disabilitare opzioni, o visualizzare valori bisogna eseguire un bind diverso, come il seguente:

```
textArea(text: “hello” ,editable:bind(source: model,  sourceProperty:'property'))
```

oppure

```
textArea(text: bind (source: model, sourceProperty:'property'))
```

Quindi una prima versione di vista da presentare all'utente per questa applicazione può essere la seguente:

```
application(title: 'Documents Manager',
  size: [640, 480],
  pack: true,
  locationByPlatform: true,
  iconImage: imageIcon('/griffon-icon-48x48.png').image,
  iconImages: [imageIcon('/griffon-icon-48x48.png').image,
               imageIcon('/griffon-icon-32x32.png').image,
               imageIcon('/griffon-icon-16x16.png').image]) {
  borderLayout()
  panel(border: emptyBorder(20),layout: new MigLayout('fill')) {
  panel(layout: new MigLayout('fill'), border: titledBorder('Search document'), constraints:
'grow 50'){
    label 'Enter a keyword'
    textField(text: bind(target: model, targetProperty: 'keyword'),columns:20)
    checkBox(text: 'concorrenza', constraints:'wrap',selected:bind(target: model,
targetProperty:'concorrenza') )
    checkBox(text: 'reti di petri', constraints:'wrap',selected:bind(target: model,
targetProperty:'petri'))
    button('Search!', actionPerformed: controller.search, constraints: 'span 5, bottom,
right')
    }
}
```

Che offre un campo di testo per inserire una parola chiave , due checkbox per scegliere il genere del documento e il bottone per eseguire il metodo che avvia la ricerca nel database. Di seguito si propone il codice completo dell'applicazione, dopo aver visto nei precedenti paragrafi le bozze (o per meglio un primitivo sviluppo) della stessa.

5.5 Il codice completo

L'applicazione è stata sviluppata secondo il paradigma MVC in Griffon. Partiamo quindi riportando il codice del modello:

```
package test

import groovy.beans.Bindable

import ca.odell.glazedlists.EventList
import ca.odell.glazedlists.BasicEventList
import ca.odell.glazedlists.SortedList

class TestModel {

    //lista ordinata dei risultati della query eseguita dal controllore
   EventList docsList = new SortedList(new BasicEventList(),
     {a, b -> a.filename <=> b.filename} as Comparator)
```

```
    /*parametri del sistema, i valori di questi vengono presi dalla vista e contengono
parole chiave,genere,nome del documento da aprire e scelta di aprire in scrittura o lettura
il documento */
    @Bindable String keyword1
    @Bindable String keyword2
    @Bindable String filename
    @Bindable boolean write
    @Bindable boolean petri
    @Bindable boolean schedulazione
    @Bindable boolean concorrenza
    @Bindable boolean programmazione
}
```

Ora la vista:

```
package test
import net.miginfocom.swing.MigLayout
import groovy.swing.SwingBuilder

application(title: 'Documents Manager',
  size: [640, 480],
  pack: true,
  locationByPlatform: true,
  iconImage: imageIcon('/griffon-icon-48x48.png').image,
  iconImages: [imageIcon('/griffon-icon-48x48.png').image,
               imageIcon('/griffon-icon-32x32.png').image,
               imageIcon('/griffon-icon-16x16.png').image]) {
  borderLayout()
/*pannello che gestisce la ricerca dei documenti: composto da campi di testo per inserire le
parole chiave e checkbox per selezionare il genere del documento cercato */
  panel(border: emptyBorder(20),layout: new MigLayout('fill')) {
  panel(layout: new MigLayout('fill'), border: titledBorder('Search document'), constraints:
'grow 50'){
    label 'Enter a keyword'
    textField(text: bind(target: model, targetProperty: 'keyword1'),columns:20)
    label 'Enter an other keyword'
    textField(text: bind(target: model, targetProperty: 'keyword2'),columns:20)
    separator(constraints:"cell 0 1")
    label ('Choose genres:')
    separator(constraints:"cell 0 2")
    checkBox(text: 'concorrenza', constraints:'wrap',selected:bind(target: model,
targetProperty:'concorrenza') )
    separator(constraints:"cell 0 3")
    checkBox(text: 'reti di petri', constraints:'wrap',selected:bind(target: model,
targetProperty:'petri'))
    separator(constraints:"cell 0 2")
    checkBox(text: 'programmazione', constraints:'wrap',selected:bind(target: model,
targetProperty:'programmazione'))
    separator(constraints:"cell 0 3")
    checkBox(text: 'schedulazione', constraints:'wrap',selected:bind(target: model,
targetProperty:'schedulazione'))
//bottone che avvia la ricerca
    button('Search!', actionPerformed: controller.search, constraints: 'span 5, bottom,
right')
    }

/*ora il pannello che contiene la tabella con i risultati della ricerca, mostra il nome del
documento e il genere */
    scrollPane(constraints: "span, grow") {
        table(id: 'docsTable') {
            tableFormat = defaultTableFormat(columnNames: ['filename', 'genre'])
```

```
            eventTableModel(source: model.docsList, format: tableFormat)
            installTableComparatorChooser(source: model.docsList)
        }
    }
/*pannello che consente di richiedere l'apertura di un documento, contiene un campo di testo
per il nome del documento e un checkbox per scegliere se parire il documento in sola lettura
o anche in scrittura */
    panel(layout: new MigLayout('fill'), border: titledBorder('Ricerca documento'),
constraints: 'grow 50'){
    label ('Enter filename to open:')
    textField(text: bind(target: model, targetProperty: 'filename'),columns:20)
    checkBox(text: 'write mode', constraints:'wrap',selected:bind(target: model,
targetProperty:'write'))
//bottone per avviare l'apertura del documento
    button('Open!', actionPerformed: controller.open, constraints: 'span 5, bottom, right')
    }
  }
}
```

La vista che contiene il documento che si è scelto di aprire qui non è presente in quanto bisogna inserirla nel controllore di cui si riporta ora il codice:

```
package test
import javax.*
import java.io.*;
import java.util.zip.*;
import groovy.swing.*;
import net.miginfocom.swing.MigLayout;


class TestController {

    def model
    def t1
    def t2

    /*funzione per l'esecuzione della query di ricerca, i risultati vanno in una lista
ordinata,le if servono per scegliere su che genere effettuare la ricerca */
    def search = { evt ->
        model.docsList.clear()
        if(model.petri){
        withSql { dsName, sql ->
            def tmpList = []
            sql.eachRow("SELECT * FROM docs WHERE (keyword1 = '${model.keyword1}' OR
keyword2 = '${model.keyword1}' OR keyword1 = '${model.keyword2}' OR keyword2 =
'${model.keyword2}') AND genre = 'reti di petri' " ) {
                tmpList << [filename: it.filename,
                            genre: it.genre]
            }
            execSync { model.docsList.addAll(tmpList) }
        }
        }
        if(model.schedulazione){
        withSql { dsName, sql ->
            def tmpList = []
            sql.eachRow("SELECT * FROM docs WHERE (keyword1 = '${model.keyword1}' OR
keyword2 = '${model.keyword1}' OR keyword1 = '${model.keyword2}' OR keyword2 =
'${model.keyword2}') AND genre = 'schedulazione' " ) {
                tmpList << [filename: it.filename,
                            genre: it.genre]
            }
            execSync { model.docsList.addAll(tmpList) }
```

```
        }
        }

        if(model.concorrenza){
        withSql { dsName, sql ->
            def tmpList = []
            sql.eachRow("SELECT * FROM docs WHERE (keyword1 = '${model.keyword1}' OR
keyword2 = '${model.keyword1}' OR keyword1 = '${model.keyword2}' OR keyword2 =
'${model.keyword2}') AND genre = 'concorrenza' " ) {
                tmpList << [filename: it.filename,
                            genre: it.genre]
            }
            execSync { model.docsList.addAll(tmpList) }
        }
        }

    if(model.programmazione){
        withSql { dsName, sql ->
            def tmpList = []
            sql.eachRow("SELECT * FROM docs WHERE (keyword1 = '${model.keyword1}' OR
keyword2 = '${model.keyword1}' OR keyword1 = '${model.keyword2}' OR keyword2 =
'${model.keyword2}') AND genre = 'programmazione' " ) {
                tmpList << [filename: it.filename,
                            genre: it.genre]
            }
            execSync { model.docsList.addAll(tmpList) }
        }
        }
    }



/*funzione per aprire il documento dato il nome, come stabilito da progetto una volta
ricevuto il nome del file appende gli identificatori della parte del documento da aprire e
poi richiama la classe Java per la gestione dei file zip,i risultati vengono messi in una
finestra con i due campi modificabili a seconda della scelta dell'utente */
def open = {
     String s = "${model.filename}"
     def part1 = new GestoreArchivio().read(s+"_part1.txt")
     def part2 = new GestoreArchivio().read(s+"_part2.txt")
     def part3 = new GestoreArchivio().read(s+"_part3.txt")
     def part4 = new GestoreArchivio().read(s+"_part4.txt")
            new groovy.swing.SwingBuilder().frame(title:'File opened',size: [800,600]
,show:true){
                panel(layout: new MigLayout('fill'), border: titledBorder('Content')){
                    separator(constraints:"cell 0 0")
                    textArea(text: part1 , editable: false)
                    separator(constraints:"cell 0 1")
                    textArea(text: part2 , editable: false)
                    separator(constraints:"cell 0 1")
                    t1 = textArea(text: part3 , editable: bind (source: model,
sourceProperty:'write'),columns:50)
                    separator(constraints:"cell 1 2")
                    t2 = textArea(text: part4 , editable: bind (source: model,
sourceProperty:'write'))
                    separator(constraints:"cell 0 3")
                    button('Save!', actionPerformed: save , enabled: bind (source: model,
sourceProperty:'write'), constraints: 'span 5, bottom, right')
                }
            }
    }
```

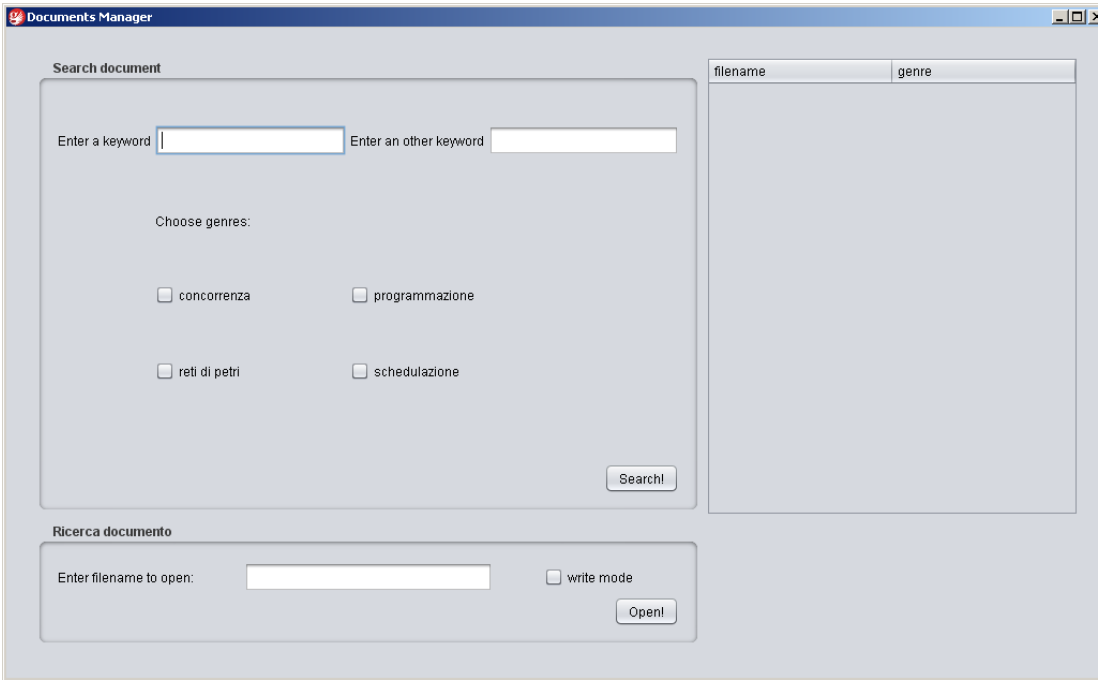
```
/*funzione per salvare i campi di testo modificati dall'utente,prende il valore del campo di
testo dai riferimenti alle textArea istanziate nella vista */
def save = { evt = null ->
    String s = "${model.filename}"
    new GestoreArchivio().write(s,t1.getText(),t2.getText())
    }
}


/* classe di supporto per la gestione dei file zip
public class GestoreArchivio {


/* funzione per leggere il contenuto di un file contenuto nell'archivio zip, il parametro è
il nome del file da aprire, ritorna il contenuto del file di testo */
    public String read(String filename) {
            String content="";
            try
            {
            ZipFile zi = new ZipFile("archivio.zip");
            ZipEntry entry = zi.getEntry(filename);
            int len=(int)entry.getSize();
            InputStream fobj = zi.getInputStream(filename);
                for(int j=0;j<len;j++)
                {
                char c=(char)fobj.read();
                content=content + c;
                }
            }
            catch (Exception e)
            {
                return "file not found!";
            }
            return content;
    }


/* funzione per modificare le due parti del documento (ovvero i due file contenenti queste
parti) , riceve come parametro il nome del file e i nuovi contenuti delle parti del
documento modificate */
public void write (String filename,String code,String comment){
    ZipFile zipSrc = new ZipFile("archivio.zip");
    ZipOutputStream zos = new ZipOutputStream(new FileOutputStream("temp.zip"));

        Enumeration srcEntries = zipSrc.entries();
        // ciclo di copia degli elementi da non modificare
        while (srcEntries.hasMoreElements()) {
            ZipEntry entry = (ZipEntry) srcEntries.nextElement();

       //selezione elementi da copiare,scartando quelli da modificare
if(!(entry.getName().compareTo(filename+"_part3.txt")==0)&&!(entry.getName().compareTo(filen
ame+"_part4.txt")==0)){

                ZipEntry newEntry = new ZipEntry(entry.getName());
                zos.putNextEntry(newEntry);

                BufferedInputStream bis = new BufferedInputStream(zipSrc
                            .getInputStream(entry));

                //copia del file
                while (bis.available() > 0) {
                    zos.write(bis.read());
                }
```

```
            /*System.gc() è richiesto a causa di un bug che non chiude bene lo stream e non
finalizza il file     */
            zos.closeEntry();
            bis.close();
            bis=null;
            System.gc();
            }
    }

    //inserimento primo contenuto modificato nel novo archivio
    zos.putNextEntry(new ZipEntry(filename+"_part3.txt"));
    byte[] blab;
    blab = code.getBytes();
    zos.write(blab, 0, blab.length);
    zos.closeEntry();

    //inserimento secondo contenuto modificato nel novo archivio
    zos.putNextEntry(new ZipEntry(filename+"_part4.txt"));
    blab = comment.getBytes();
    zos.write(blab, 0, blab.length);
    zos.closeEntry();

    //chiusura dell'archivio
    zos.flush();
    zos.close();
    zipSrc.close();
    zos = null;
    System.gc();

    //sostituzione archivio vecchio con quello nuovo
    File f = new File("archivio.zip");
    f.delete();
    File toBeRenamed = new File("temp.zip");
    File newFile = new File("archivio.zip");
    toBeRenamed.renameTo(newFile);

    }
}
```

5.6 Risultati e test dell'applicazione

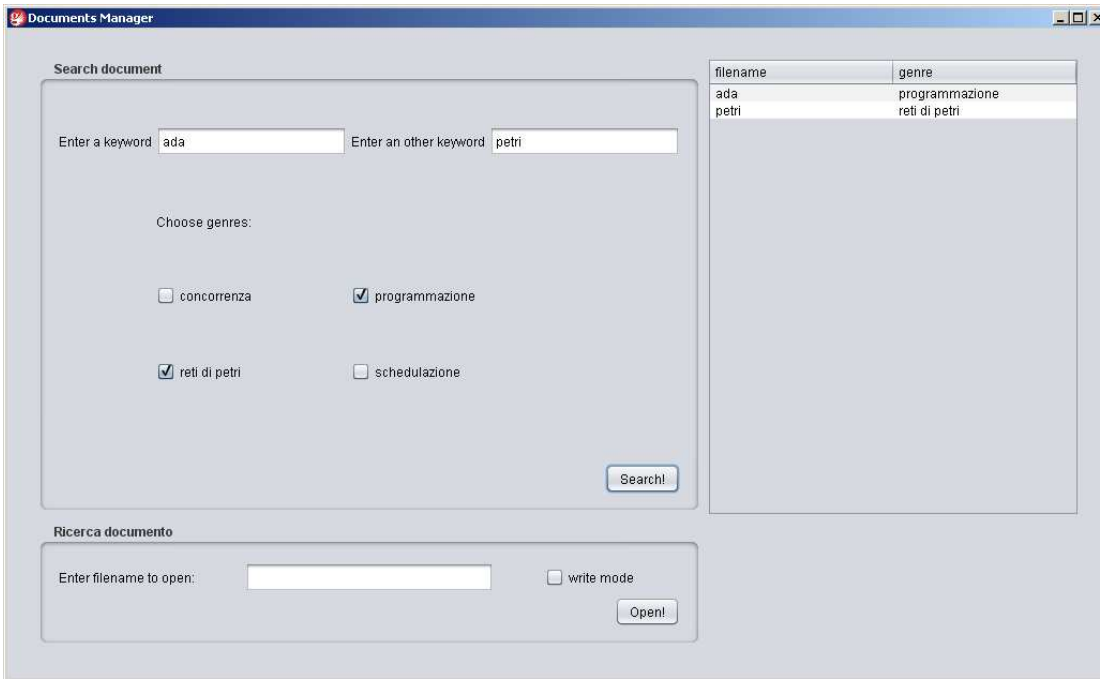
Ecco riportato il funzionamento dell'applicazione con i test sulle sue funzionalità:

- per prima cosa, come si presenta l'applicazione all'avvio, prima di qualunque iterazione con l'utente



Come si può vedere sono presenti i tre pannelli, uno per richiedere i parametri di ricerca con i campi di testo per le parole chiave e le checkbox per selezionare il genere trattato nel documento; uno sulla destra per visualizzare i risultati; e uno in basso per inserire il nome del documento da aprire.

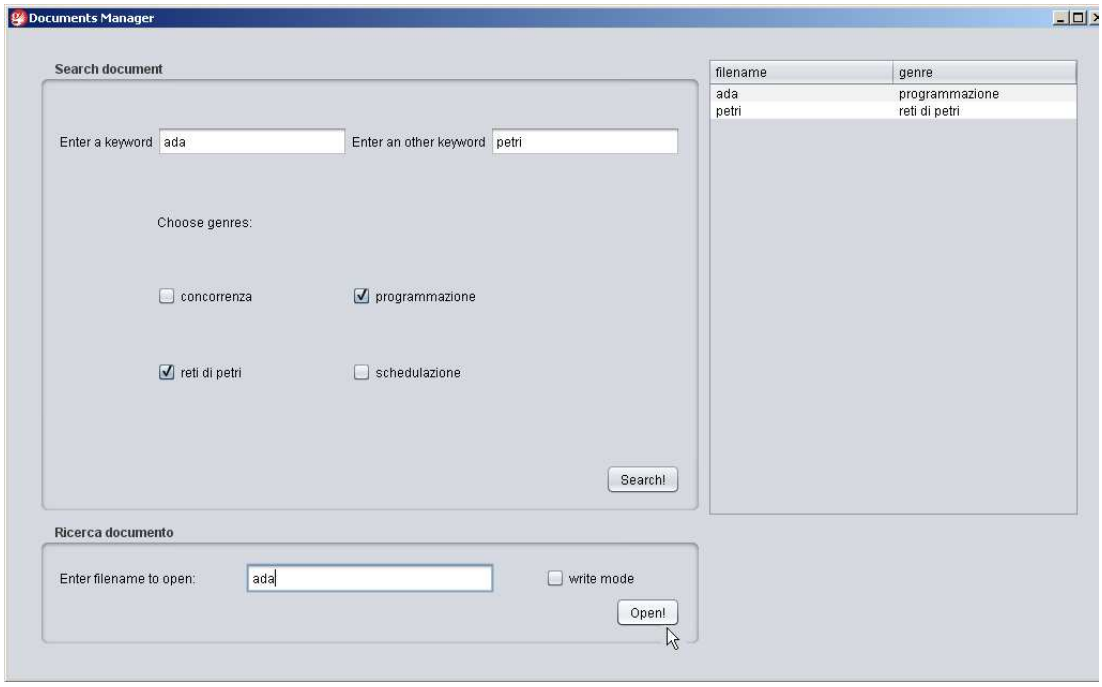
- ora si prova ad effettuare una ricerca, inserendo alcune parole chiave e scegliendo alcuni generi:

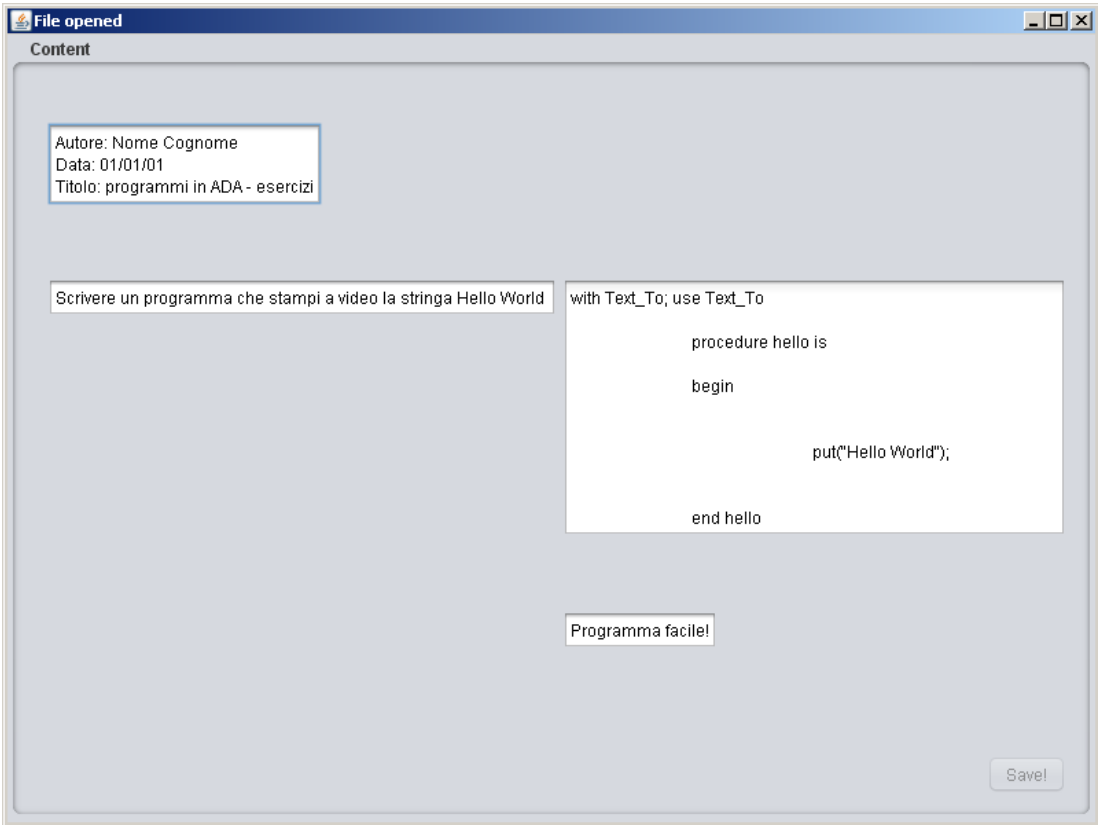


Come si può vedere, cercando documenti riguardanti la programmazione e le reti di petri, con le parole chiave "ada" e "petri" si ottengono alcuni match che vengono mostrati nella tabella di destra, ove viene mostrato nome del documento e genere dell'argomento trattato.

Nota: l'ordine delle parole chiave è indifferente, se fossero state scambiate si sarebbe ottenuto lo stesso risultato; da notare inoltre l'ordine alfabetico dei risultati; infine eseguire un'altra ricerca avrebbe l'effetto di svuotare la tabella cancellando i risultati della ricerca precedente per lasciare solo quelli della nuova ricerca effettuata.

- ora supponiamo che l'utente voglia aprire il documento riguardante ada, per vedere l'esercizio proposto, ma senza aver intenzione di modificarlo:



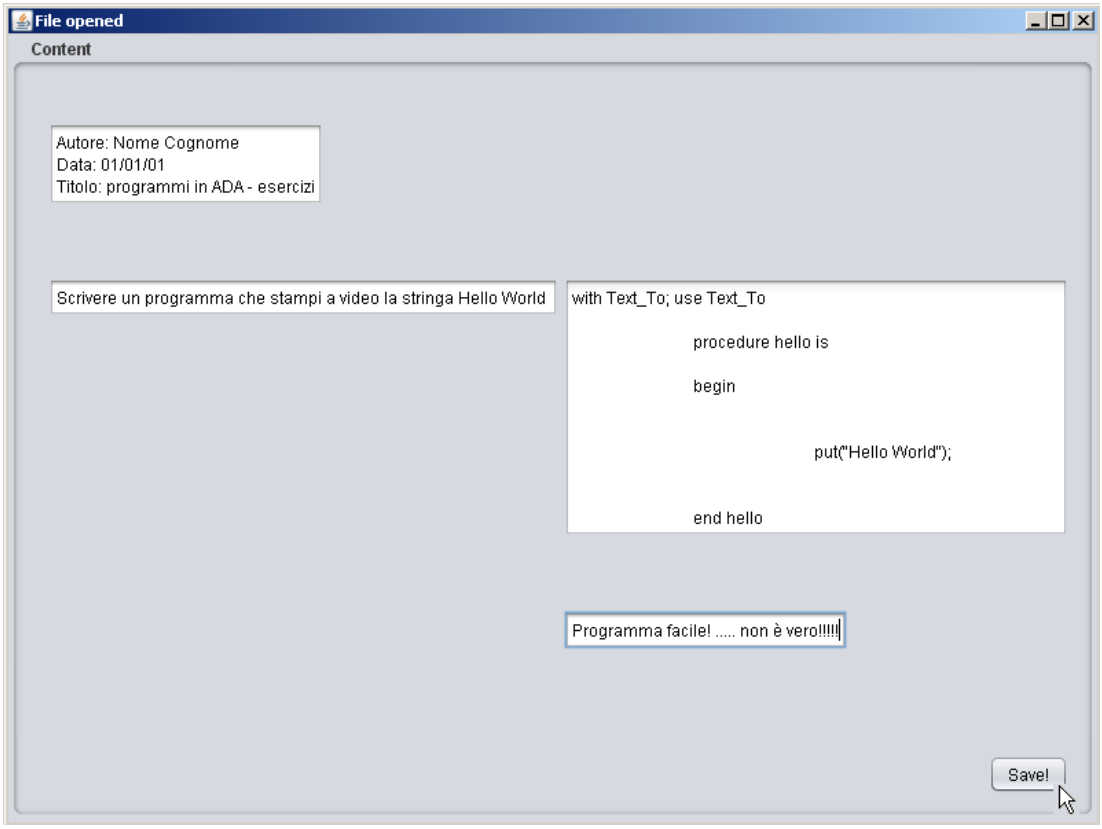


Cliccando sul pulsante “Open!” si apre il documento chiamato “ada” , il cui contenuto viene mostrato in un’altra finestra.

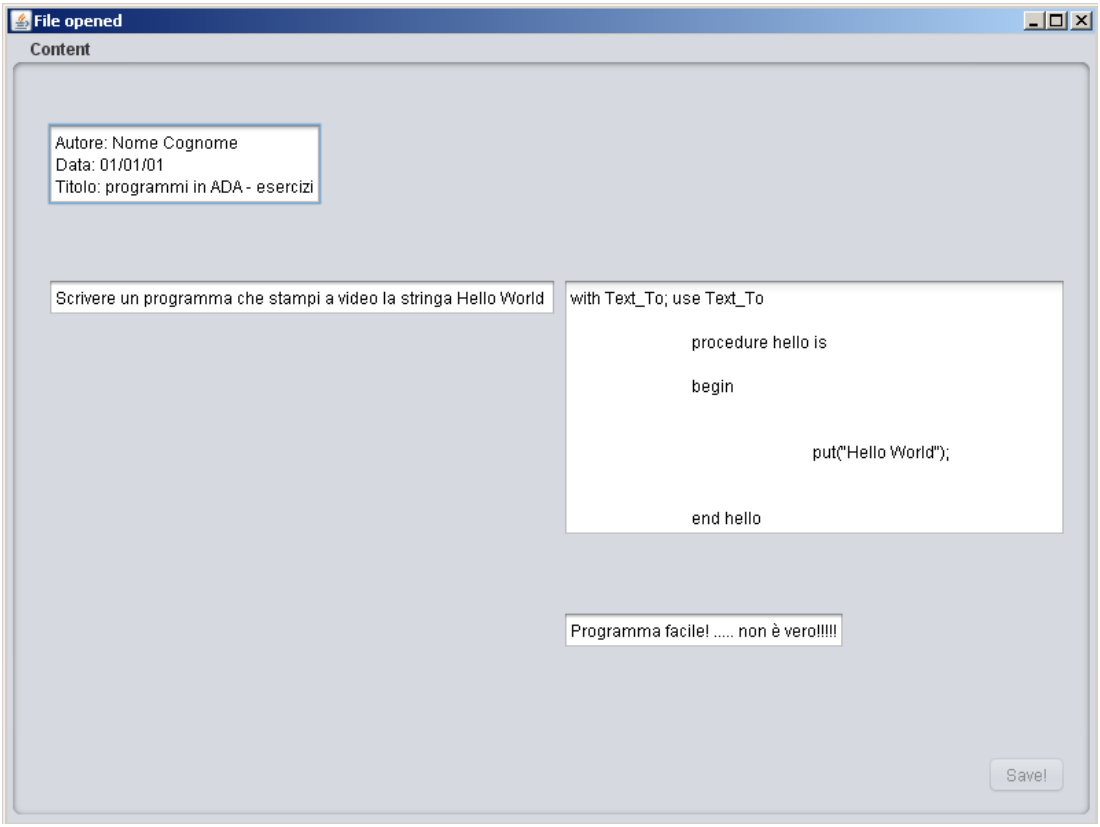
Come si può vedere, si possono distinguere le quattro parti di cui è composto: in alto la prima parte con autore e titolo, sotto la senconda parte con il testo l’esercizio e al suo fianco la terza con il codice della soluzione, infine in basso la quarta parte con i commenti all’esercizio.

Da notare come non avendo selezionato l’opzione di scrittura, non sia possibile editare il documento e il bottone di salvataggio delle modifiche sia disabilitato.

Ora lo si apre in scrittura e si apporta qualche modifica, salvandola:



Riaprendo il file si può vedere come le modifiche sono state salvate:



L'applicazione quindi funziona perfettamente secondo le consegne di progetto, offrendo tutte le funzionalità richieste.

5.7 Conclusioni

L'applicazione sviluppata, pur molto spartana nella sua realizzazione, rispetta pienamente le specifiche di progetto e svolge correttamente tutti i compiti assegnati.

Essa essendo un'applicazione web andrà distribuita su server e client, si possono avere due approcci per la distribuzione delle componenti dell'applicazione: se si tratta di un archivio condiviso da più persone e localizzato in remoto converrà mantere il database e l'archivio nel lato server, mentre controllore e vista nel lato client; se invece si tratta di archivio in locale, converrà manterere database e archivio nel lato client mentre controllore e resto dell'applicazione sul server. Oltretutto l'applicazione, essendo stata sviluppata in Griffon, può essere portata in Grails senza problemi.

Questo esempio mostra come Groovy sia il futuro per quanto riguarda la programmazione e lo sviluppo di applicazioni: il linguaggio risulta di facile apprendimento essendo simile a Java, ma è molto più efficiente ed efficace. Una cosa che è apparsa evidente nello sviluppo di quest'applicazione è che Groovy permette di scrivere pochissimo codice, circa un terzo di quello richiesto da Java. Il fatto che Groovy possa usare API e classi scritte in Java si rivela un altro elemento molto utile; infine sviluppare GUI in Groovy è decisamente più piacevole ed.

Come detto questo è un prototipo e quindi può essere ovviamente migliorato: ad esempio con un database diverso da uno relazionale o con attributi multivalore oppure passare a metodi di ricerca più raffinati come ricerca approssimata del titolo o ricerca pesata su keyword mostrando solo risultati significativi o ordinati per peso (ad esempio valutando la frequenza nel documento delle occorrenze della parola chiave cercata) ecc…; si può passare ad un modello di archivio più complesso, con documenti annidati in cartelle; possono essere progettati documenti più strutturati, con tag per le direttive di visualizzazione.

Concludendo questa tesi ha mostrato come la combinazione di Groovy e Griffon sia risultata eccellente per lo sviluppo di un'applicazione prototipale per la visione unificata di oggetti via browser.

Bibliografia

[1] Andres Almiray, Danno Ferrin, James Shingler, Griffon in Action, Manning Publications.

[2] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, Jon Skeet, Groovy in Action , Manning Publications.

[3] Venkat Subramaniam, Programming Groovy: Dynamic Productivity for the Java Developer, O'Reilly Vlg. Gmbh & Co..

[4] Kenneth Barclay, John Savage, Groovy Programming: An Introduction for Java Developers, Morgan Kaufmann.

[5] Alexander Reelsen, Play Framework Cookbook, Packt Publishing.

[6] Eugene Ciurana, Developing with Google App Engine, Apress.

[7] Dan Sanderson, Programming Google App Engine, O'Reilly Vlg. Gmbh & Co..

[8] Oracle, Java API - http://docs.oracle.com .

[9] Heiko Böck, The Definitive Guide to the NetBeans Platform 7, Apress.

[10] Codehaus.org , Griffon docs – http://griffon.codehaus.org/Documentation.

[11] Andres Almiray, Personal blog – www.jroller.com/aalmiray.