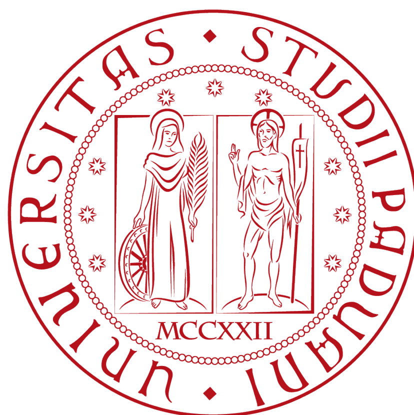


UNIVERSITA' DEGLI STUDI DI PADOVA

FACOLTA' DI INGEGNERIA

CORSO DI LAUREA TRIENNALE IN INGEGNERIA INFORMATICA

IL FRAMEWORK WEB RUBY ON RAILS E BREVE INTRODUZIONE ALLA PIATTAFORMA DI E-LEARNING CANVAS

RELATORE: CH.MO PROF. MICHELE MORO

LAUREANDO: DIEGO CARRARO

ANNO ACCADEMICO: 2012-2013

Indice

1 Introduzione **5**

2 Panoramica sul linguaggio Ruby **7**

2.1 Nomi 7
2.2 Metodi 8
2.3 Tipi di dati 8
2.3.1 Numeri 8
2.3.2 Stringhe 9
2.3.3 Array 9
2.3.4 Hash 9
2.3.5 Espressioni regolari 10
2.4 Eccezioni 11
2.5 Strutture Organizzative 11
2.5.1 Classi 11
2.5.2 Moduli 13
2.6 Oggetti Marshaling 13
2.7 Idiomi 14

3 L'architettura di un'applicazione Rails **15**

3.1 Model-View-Controller 15
3.2 Il Modello 16
3.2.1 Object-Relational Mapping (ORM) 16
3.2.2 Active Record 17
3.3 La Vista 17
3.4 Il Controller 17

4 Struttura di un'applicazione Rails **19**

4.1 Le directory dell'applicazione 19
4.2 Convenzioni sui nomi 21

5 Creare una semplice applicazione **24**

5.1 Creare un progetto 24
5.2 Configurare il database 24
5.3 Hello Rails! 24
5.4 Flusso di un'applicazione 26

5.5 Lo scaffolding 27

6 Modelli 28

6.1 Definire le classi del modello 28
6.2 Specificare relazioni fra modelli 29
6.2.1 Relazione uno-a-molti 29
6.2.2 Relazione uno-a-uno 30
6.2.3 Relazione molti-a-molti 31
6.3 Operazioni con i modelli (CRUD) 31
6.3.1 Inserire nuove righe: i metodi new e create 32
6.3.2 Interrogare il database 33
6.3.3 Modificare righe del database 34
6.3.4 Cancellare righe del database 34
6.3.5 Ciclo di vita dei modelli 35
6.4 Transazioni 36
6.5 Le migrazioni 37
6.5.1 Creare una migrazione 37
6.5.2 Eseguire una migrazione 38
6.5.3 Anatomia di una migrazione 39
6.5.4 Gestire le tabelle 40

7 Action Pack 42

7.1 Action Dispatch 42
7.1.1 Routing basato su REST 42
7.2 Action Controller 48
7.2.1 Metodi e Azioni 48
7.2.2 Visualizzare un template 48
7.2.3 Mandare file e altri dati 50
7.3 Action View 51
7.3.1 Utilizzare template 51
7.3.2 Utilizzare form 52
7.3.3 Utilizzare Helper 55
7.3.4 Layout e Partial 56

8 Il caching 59

8.1 Caching delle pagine 59

8.2 Caching delle azioni 61
8.3 Caching dei frammenti 62

9 Altro da sapere su Rails **63**

10 LMS Canvas **65**
10.1 Keywords del progetto 65

Sommario

L'obiettivo primario di questa tesi è esplorare il Framework web Ruby On Rails, senza la pretesa di scrivere una guida completa sull'argomento. Si tratta sostanzialmente di un percorso alla scoperta delle caratteristiche principali di RoR, dalle linee guida da seguire nella programmazione coadiuvate dalla solida struttura sulla quale si basano le applicazioni che vengono sviluppate, ai meccanismi tra le varie parti di codice, passando per l'analisi dei moduli che rendono Rails così interessante. Alcuni aspetti verranno corredati anche da piccoli esempi di codice, mentre altri verrano esplicati solo dal punto di vista concettuale. Inoltre, nella parte conclusiva, verrà presentata una breve overview della piattaforma Canvas, un esempio concreto di applicazione realizzata con il framework.

Per la realizzazione dell'elaborato si è utilizzato principalmente il testo " Agile Web Development with Rails" che è considerato la bibbia di Rails, in quanto uno degli autori è proprio il creatore del framework. Sono poi stati consultati vari siti più o meno ufficiali con ulteriori spiegazioni, esempi di codice, documentazioni e commenti. Il tutto senza però aver mai toccato con mano l'implementazione di una vera applicazione, ma solo di semplici esempi.

1 Introduzione

Ruby on Rails, o più semplicemente Rails, è un ambiente completo per lo sviluppo web ideato da David Heinemeier Hansson. Estratto dal codice di Basecamp, un'applicazione Web per il project management di 37signals e rilasciato come progetto open source nel 2004, Rails contiene al suo interno tutti gli elementi necessari alla realizzazione di siti complessi permettendo di gestire facilmente la creazione di pagine (X)HTML, di accedere semplicemente a database, di integrare le funzionalità che caratterizzano le applicazioni web moderne, come le funzionalità AJAX ed i Web service e molto altro ancora.

Viene principalmente distribuito attraverso RubyGems, che è il formato dei pacchetti e il canale di distribuzione ufficiale per librerie ed applicazioni Ruby. Negli ultimi anni ha creato un vero e proprio terremoto all'interno della comunità degli sviluppatori, diventando spesso motivo di dibattito e ispirando la nascita di progetti analoghi realizzati con tecnologie differenti, come Cake per PHP, Trails per Java, Turbogears e Subway per Python e molti altri. Rails, in altre parole, ha introdotto un fattore di novità rilevante nell'ambito della programmazione Web.

Riassumendo le sue caratteristiche in due righe, si può dire che usa tecniche di programmazione già sperimentate e non rivoluzionarie. La potenza di Rails è il racchiudere questi meccanismi all'interno di un modello nuovo promettendo di ridurre drasticamente i tempi di sviluppo, abolendo i file di configurazione, automatizzando tutto ciò che è possibile, usando una struttura portante solida e coerente per la generazione del codice.

L'autore tiene a ribadire che Rails non è stato sviluppato da subito come una piattaforma indipendente, ma che è il risultato dell'estrazione di funzionalità già provate in un'applicazione funzionante, e che ogni feature è mirata alla soluzione di problemi reali e non è frutto di ragionamenti astratti. L'opinione condivisa è che sia proprio questo a renderlo così efficace.

Indubbiamente parte del successo di Rails è dovuto al linguaggio con cui è scritto, ovvero Ruby, un linguaggio completamente ad oggetti di estrema espressività e potenza, che riesce a fondere in una sintassi semplice e chiara funzionalità ereditate da Perl, Python, Lisp e Smalltalk. Per questo molti dei progetti mirati a riscrivere Rails in un altro linguaggio hanno poco senso, visto che è Ruby a determinare gran parte del feeling di questo ambiente.

Per leggere questa tesi nel migliore dei modi e così afferrare i concetti non solo

dal punto di vista logico, ma anche da quello pratico del codice, è utile avere una conoscenza di base del linguaggio Ruby che in realtà dovrebbe essere più approfondita della semplice overview che si presenterà nel prossimo capitolo. Inoltre sarebbe opportuno procedere di pari passo con lo svolgimento di piccoli esercizi o la creazione di un progettino per mettere in pratica tutto ciò di cui viene discusso.

2 Panoramica sul linguaggio Ruby

Ruby è un linguaggio open source, general purpose e interpretato. È un linguaggio orientato agli oggetti puro, nel senso che, come nello Smalltalk, ogni cosa è un oggetto con i propri metodi. Tuttavia presenta però anche aspetti del paradigma imperativo e di quello funzionale.

Ruby di per sè non ha niente di estremamente nuovo ma prende il meglio dai più rivoluzionari linguaggi, primi su tutti lo Smalltalk e il Perl, ma anche Pyhton, LISP e Dylan. Il risultato è estremamente potente e sintatticamente gradevole, infatti uno dei primi slogan del Ruby fu: *Ruby > (Smalltalk + Perl) / 2.*

Ruby possiede inoltre una sintassi pulita e lineare che permette ai nuovi sviluppatori di imparare il linguaggio ed essere produttivi in pochissimo tempo. È possibile infine estendere Ruby in C e in C++ in modo estremamente semplice se confrontato con i meccanismi di estensione degli altri linguaggi.

Vediamo ora una panoramica delle principali sintassi, tanto per riuscire a comprendere meglio il codice che verrà presentato in seguito.

2.1 Nomi

- Variabili locali, parametri dei metodi e nomi dei metodi iniziano con la lettera minuscola e i nomi composti vengono separati da underscore;
- nomi di classi, moduli e costanti iniziano con una lettera maiuscola e si usa la tecnica "gobba di cammello" per i nomi composti;
- rails usa *simboli* per identificare alcune cose, in particolare quando si specificano chiavi per parametri di metodi hash

  ```
  redirect_to :action => "edit", :id => params[:id]
  ```

  Un simbolo sembra il nome di una variabile, ma è preceduto da : . Si può pensare ai simboli come a stringhe di letterali, che sono magicamente trasformate in costanti. In alternativa, si possono considerare i due punti come la cosa nominata, pertanto :id è la cosa nominata id. Si può pensare a :id come al nome della variabile id, in altre parole come al suo valore.

2.2 Metodi

```
def say_goodnight(name)
   result = 'Goof night, ' + name
   return result
end
# Time for bed......
puts say_goodnight ('Mary-Ellen')
puts say_goodnight ('John-boy')
```

Un metodo ha un nome e viene definito fra le parole chiave def e end. Il return è opzionale e se non è presente viene restituito il valore dell'ultima espressione valutata all'interno del metodo.

Non serve il punto e virgola alla fine di un'istruzione se ognuna occupa una riga diversa. I commenti si realizzanno utilizzando # e l'indentazione non ha alcuna rilevanza sintattica.

2.3 Tipi di dati

Di seguito verrà presentata una breve descrizione delle classi che corrispondono ai principali tipi di dati negli altri linguaggi, e per maggiori dettagli (indispensabili) si rimanda ai relativi manuali.

2.3.1 Numeri

Il linguaggio Ruby prevede l'esistenza di numeri interi e di numeri in virgola mobile (floating-point). I primi sono oggetti delle classi Fixnum o Bignum mentre i secondi sono di tipo Float. I numeri che possono essere rappresentati in una word (meno un bit) sono oggetti della classe Fixnum, quelli che vanno oltre questo limite sono invece istanze della classe Bignum. Per rappresentare numeri non decimali va fatto precedere al numero vero e proprio un indicatore di base (0b per i binari, 0 per gli ottali, 0d per i decimali e 0x per gli esadecimali).

Essendo i numeri oggetti di una determinata classe, possiamo applicare ad essi i metodi previsti dalle rispettive classi (operazioni aritmetiche di base, operazioni sui bit, valore assoluto, conversioni, ecc). Altri metodi sono ereditati dalle classi Integer e Numeric poiché entrambe le classi derivano da Integer che a sua volta deriva da Numeric. Abbiamo ad esempio chr, floor, next, step e molti altri.

2.3.2 Stringhe

Una delle maggiori influenze del linguaggio Ruby è il Perl e da questo eredita, tra l'altro, una potente e avanzata gestione del testo. Le stringhe, sequenze di caratteri racchiusi tra virgolette (") o tra singoli apici ('), sono delle istanze della classe String. È previsto un avanzato sistema di accesso ai singoli caratteri delle stringhe o a sottostringhe attraverso il metodo [ ]. In generale comunque si hanno più o meno gli stessi metodi per la manipolazione delle stringhe che si hanno negli altri linguaggi ad oggetti. Una differenza sostanziale è invece la gestione delle stringhe create con apici singoli o doppi:

- singoli: qualsiasi sequenza di caratteri al loro interno diventa il valore della stringa;
- doppi: Ruby interpreta prima le eventuali sequenze che iniziano con il backslash e lo sostituisce con un valore binario (tipico esempio il \n che fa andare a capo); in secondo luogo Ruby esegue l'interpolazione di espressioni, perciò la sequenza # {*espressione*} viene sostituita dal valore di *espressione*.

2.3.3 Array

Un array può essere visto come una lista di oggetti non necessariamente dello stesso tipo, accessibili tramite una chiave costituita da un numero intero. Gli array sono istanze della classe Array e possono dunque essere creati nei seguenti modi: arr = Array.new, oppure utilizzando il metodo [ ] che elenca espressamente ad uno ad uno i vari elementi. Gli array, oltre a collezionare elementi eterogenei, hanno il plauso di essere molto flessibili rispetto a quelli di Java per esempio. Infatti la loro dimensione non è fissata in precedenza, ma si possono inserire o rimuovere elementi in qualsiasi posizione a piacimento attraverso una vasta gamma di metodi. Inoltre sono presenti vari iteratori applicabili alle istanze della classe Array che ne facilitano la gestione.

2.3.4 Hash

Gli hash, istanze della classe Hash, sono liste di coppie di chiave-valore. A differenza degli array, negli hash troviamo delle chiavi per l'accesso agli elementi che possono essere di tipo qualsiasi. In Ruby gli hash sono racchiusi tra parentesi

graffe e ogni coppia è separata dall'altra da una virgola, mentre tra le chiavi e i valori ci deve essere il simbolo =>. Un esempio di hash è il seguente

```
inst_section = {
   :cello      => 'string'
   :clarinet   => 'woodwind'
   :violin     => 'string'
}
```

Le chiavi devono essere uniche e di solito in Rails sono dei simboli.

Come gli array anche gli hash possono essere creati in vari modi, innanzitutto scrivendo direttamente le coppie chiave-valore tra parentesi graffe (come nell'esempio) oppure utilizzando il metodo [ ]. Per elencare invece tutte le chiavi e i valori di un hash si utilizzano i metodi keys e values. Infine, oltre a buona parte di iteratori dal funzionamento uguale a qello degli array, gli hash ne hanno di propri che permettono di sfruttare al meglio la loro struttura.

2.3.5 Espressioni regolari

Le espressioni regolari sono uno strumento fondamentale per la gestione del testo, e uno dei linguaggi che ha fatto di questo strumento un punto di forza è sicuramente il Perl. Brevemente, le espressioni regolari vengono utilizzate per controllare se una stringa verifica un certo schema (pattern). O in altre parole, le espressioni regolari forniscono dei modelli per ricercare all'interno di un testo non solo utilizzando espressioni letterali, ma anche particolari identificatori che rappresentano delle determinate classi di caratteri.

In Ruby sono oggetti della classe Regexp, che possono essere creati come gli Array e gli Hash in diversi modi. Una volta creata l'espressione regolare, è possibile confrontarla con qualsiasi stringa utilizzando il metodo match della classe Regexp oppure l'operatore =~e il suo negato !~. All'interno di una espressione regolare, oltre ai normali caratteri, è possibile usare delle sequenze che rappresentano delle determinate classi. In alternativa è possibile creare delle classi racchiudendo i caratteri tra parentesi quadre, ad esempio [A-Za-z] sta ad indicare tutte le lettere dalla a alla z sia maiuscole che minuscole. Oltre ai caratteri, in una espressione regolare è possibile anche utilizzare due particolari elementi: l'accento circonflesso, che va usato se i caratteri cercati si devono trovare all'inizio del testo e $ (dollaro) se si devono trovare alla fine.

2.4 Eccezioni

Le eccezioni sono oggetti della classe Exception o di sue sottoclassi. Il metodo raise causa un'eccezione che deve essere intercettata e questo interrompe il normale flusso del codice. Sia metodi che blocchi di codice situati tra le parole chiave begin e end intercettano certe classi di eccezioni usando la clausola rescue.

```
begin
   content = load_blog_data(file_name)
rescue BlogDataNotFound
   STDERR.puts "File not found"
rescue BlogDataFormatError
   STDERR.puts "Invalid blog data in the file"
end
```

Le clausole rescue possono essere piazzate direttamente all'estremo livello della definizione di un metodo, senza il bisogno di includere il contenuto all'interno di un blocco begin/end.

2.5 Strutture Organizzative

Ci sono due concetti in Ruby per organizzare metodi, cioè classi e moduli.

2.5.1 Classi

Un esempio sarà utile a capire i concetti fondamentali, comunque molto simili agli altri linguaggi ad oggetti e che differiscono solo per la sintassi con i quali sono espressi.

```
class Order < ActiveRecord::Base
       has_many :line_items

       def self.find_all_unpaid
          self.where('paid=0')
       end
       def total
          sum = 0
          line_items.each {| li | sum += li.total}
```

```
        sum
    end
end
```

La definizione inizia con la parola chiave class seguita dal nome della classe (con lettera maiuscola). In questo caso la classe Order è una sottoclasse della classe Base la quale si trova dentro il modulo ActiveRecord.

has_many è un metodo definito all'interno di ActiveRecord ed è richiamato nel momento in cui Order è definita. Normalmente questi tipi di metodi fanno asserzioni sulle classi e vengono chiamati *dichiarazioni*.

Dentro il corpo della classe si possono definire due tipi di metodi:

- di *classe*, che hanno il prefisso self. e possono essere chiamati sulla classe in generale ed in ogni punto dell'applicazione, ad esempio

  ```
  to_collect = Order.find_all_unpaid
  ```

- di *istanza*, ovvero metodi che hanno accesso alle variabili d'istanza e che possono essere chiamati solo dalle istanze di una classe. Ad esempio

  ```
  class Greeter
     def initialize(name)
        @name = name
     end
     def name
        @name
     end
  end
  g = Greeter.new("Diego")
  puts g.name   #=> Diego
  g.name = "Marco"
  puts g.name   #=> Marco
  ```

  Un metodo di istanza è pubblico per default e chiunque può chiamarlo. Essi però possono essere resi privati, quindi invocabili solo da dall'istanza della classe nella quale sono implementati; oppure protetti, cioè richiamabili da più istanze di una classe e delle sue sottoclassi. NB: Il metodo puts stampa a video.

2.5.2 Moduli

I moduli sono simili alle classi in quanto contengono una collezione di metodi, costanti e altri moduli o definizioni di classi. A differenza delle classi però, non si possono creare oggetti basati su di essi. Sostanzialmente i moduli hanno due scopi:

- agiscono come namespace, in modo da evitare collisioni di nomi tra metodi;
- permettono di condividere funzionalità fra le classi; più classi possono essere definite all'interno di un modulo per sfruttare le sue funzionalità senza usare l'ereditarietà. Oppure una classe mista ad un modulo può usare i metodi d'istanza di quest'ultimo, essendo disponibili come se fossero definiti nella classe. I metodi helper (verranno presi in esame) sono un esempio di come rails sfrutta i moduli.

YAML è un esempio di modulo molto usato in Rails per definire configurazioni di cose come database, dati di test e traslazioni (Ma verrà visto il suo impiego soprattutto con le migrazioni). Ecco un esempio

```
development:
   adapter sqlite3
   database: db/development.sql3
   pool: 5
   timeout: 5000
```

In YALM l'indentazione è importante e in questo caso viene definito development, avente un insieme di quattro coppie chiave-valoree separate da due punti.

2.6 Oggetti Marshaling

Ruby può convertire un oggetto in un flusso di byte che può essere usato in altri punti o anche al di fuori dell'applicazione. Questo processo si chiama *marshaling*. Tuttavia a volte la conversione non è possibile o comunque pone dei problemi da cui è bene fare attenzione. Rails usa il marshaling per salvare i dati di sessione (l'argomento non verrà trattato).

2.7 Idiomi

Alcune caratteristiche di Ruby possono essere combinate in modi interessanti, ma non immediatamente intuitivi. Ecco alcuni esempi che saranno usati in seguito:

- metodi come `empty!` e `empty?` vengono chiamati rispettivamente *bang method* e *predicate method*. Il primo di solito fa qualcosa di distruttivo al ricevente. Il secondo restituisce `true` o `false` a seconda di qualche condizione.

- `a || b`. L'espressione valuta `a`. Se essa non è falsa o nulla, tale valutazione si ferma e l'espressione restituisce `a`, altrimenti ritorna `b`. Questo è un modo comune per restituire un valore di default se il primo falore non è ancora stato impostato.

3 L'architettura di un'applicazione Rails

3.1 Model-View-Controller

Le applicazioni Rails utilizzano il noto approccio Model-View-Controller (o MVC), un design pattern molto noto per organizzare il codice nei software interattivi, diventato ormai comune anche per la realizzazione di applicazioni web. Introdotto nel 1979 dal professor Trygve Reenskaug, originariamente venne impiegato dal linguaggio Smalltalk ed è poi stato sposato da numerose tecnologie moderne, come framework basati su PHP, su Python, su Java (Swing, JSF e Struts), su Objective C o su .NET.

Esso divide un'applicazione in tre parti: modello, viste e controller, permettendo di raggiungere un ottimo isolamento tra il codice che gestisce i dati e quello che li presenta all'utente.

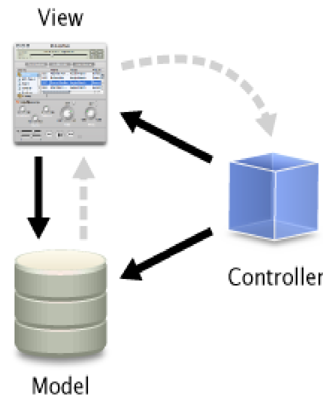
Il **modello** è il responsabile del mantenimento dello stato dell'applicazione, che a volte è transitorio, mentre a volte è permanente e viene salvato fuori dall'applicazione (spesso in un database). Il modello è qualcosa in più dei soli dati effettivi; esso comprende anche tutte le regole che devono essere applicate tra questi dati. Tale implementazione permette la loro consistenza poiché nessun altro nell'applicazione può renderli invalidi.

La **vista** si occupa di generare l'interfaccia grafica, normalmente basata sui dati del modello. Sebbene la vista si possa presentare all'utente in molti modi diversi a seconda delle richieste, essa stessa non gestisce mai i dati, ovvero il suo lavoro è soltanto quello di visualizzarli. Si potranno avere quindi molte viste basate sui medesimi contenuti del modello.

Il **controller** orchestra l'applicazione: riceve eventi dal mondo esterno (normalmente input dall'utente), interagisce con il modello e visualizza la vista appropriata.

Ruby on Rails impone una struttura all'applicazione nella quale si sviluppano modelli, viste e controller come pezzi separati ma funzionali, uniti poi al momento dell'esecuzione. La vera potenza del framework è però quella di eseguire questa unione basandosi sull'utilizzo di operazioni di default che facilitano il compito del programmatore, il quale può concentrarsi sul cuore dell'applicazione; non c'è bisogno quindi di scrivere alcuni metadati di configurazione esterni per far funzionare

il tutto. Questo è un esempio della filosofia principale di Rails: *convention over configuration*.



3.2 Il Modello

Nella maggior parte dei casi vogliamo mantenere le informazioni della nostra applicazione web all'interno di un database relazionale poiché ci offre un supporto completo e coerente. Sebbene sia difficilmente intuibile dal codice SQL, i database sono costruiti sulla base della teoria degli insiemi ed è per questo difficile collegarli ai concetti di programmazione ad oggetti: gli oggetti trattano dati e operazioni, mentre i database trattano insiemi di valori. Perciò operazioni facilmente esprimibili in termini relazionali sono difficilmente codificabili in oggetti e viceversa. Nel tempo si è lavorato molto per collegare questi due aspetti e Rails ha scelto le librerie di tipo *Object-Relational Mapping*.

3.2.1 Object-Relational Mapping (ORM)

Le librerie ORM:

- mappano tabelle in classi (tabella orders corrisponde alla classe Order);
- le righe di tale tabella corrispondono a oggetti della classe (un particolare ordine è rappresentato da un'istanza della classe Order);
- dentro all'oggetto istanziato gli attributi sono usati per impostare e recuperare i valori delle colonne (l'oggetto Order ha metodi per recuperare e impostare gli attributi prezzo, quantità, ecc);

- le classi che mappano tutto ciò forniscono anche un insieme di metodi che eseguono operazioni sulle tabelle (query), oppure sulle singole righe di una tabella (update, delete, ecc).

3.2.2 Active Record

Active Record è la libreria che implementa il livello ORM in Rails, e segue quasi fedelmente lo standard: tabelle mappate in classi, righe in oggetti e colonne in attributi di tali oggetti. Essa minimizza il numero di configurazioni che lo sviluppatore deve eseguire; questa libreria verrà esaminata in dettaglio nel capitolo 6.

3.3 La Vista

Sostanzialmente una vista è un pezzo di codice html che visualizza contenuti. Di solito però si vuole includere del contenuto dinamico che verrà creato dal metodo di azione nel controller. Esso può essere generato da un template, sostanzialmente in tre modi:

- il più comune chiamato Embedded Ruby (ERb) incastra pezzi di codice Ruby all'interno di una vista, in modo analogo a quello che viene fatto in altri framework come PHP o JSP. Sebbene questo approccio sia molto flessibile, molti sono convinti del fatto che violi lo spirito del MVC. Inserendo il codice nelle viste si rischia infatti di inserire concetti logici che andrebbero nel modello o nel controller.
- Si può usare ERb per costruire dei frammenti JavaScript che vengono eseguiti nel browser e questo è un buon metodo per creare interfacce dinamiche Ajax.
- Rails fornisce anche un XML Builder per costruire documenti XML usando codice Ruby e la struttura dell'XML generato seguirà automaticamente quella del codice.

3.4 Il Controller

Il controller è il centro logico dell'applicazione ed ha il compito di coordinare l'interazione fra utente, viste e modello. Tuttavia Rails gestisce molte di queste

interazioni dietro le quinte, lasciando concentrare il programmatore sul solo livello applicativo. Inoltre il controller è responsabile di altri servizi ausiliari quali:

- converte le richieste esterne in azioni interne utilizzando gli URL;
- fa caching, aumentando le performance dell'applicazione;
- gestisce i moduli *helper*, utili a estendere le capacità dei template delle viste senza appesantire il loro codice;
- gestisce le sessioni, dando l'impressione all'utente di un'interazione real time con l'applicazione.

4 Struttura di un'applicazione Rails

4.1 Le directory dell'applicazione

Le applicazioni sviluppate con Rails hanno una peculiarità, ovvero sono tutte organizzate secondo una struttura comune. Questo è una conseguenza del fatto che il comando rails genera una serie di directory e file che forniscono una certa linea guida nello sviluppo, linea che se rispettata permette a Rails di effettuare molte cose automaticamente (ad esempio caricare i file, generarli ed individuarli a runtime e molto altro). Questa struttura comune permette anche di comprendere con semplicità il codice di progetti realizzati da altri, in quanto sono organizzati nella stessa maniera. Vediamo nel dettaglio il significato delle varie directory e dei file che contengono, e per molti di questi ne verrà approfondita in seguito l'analisi.

App

È il cuore di un progetto, quella che contiene il codice specializzato per l'applicazione web e sulla quale il programmatore dovrà svolgere la maggior parte del lavoro. In questa directory si parlerà in dettaglio dei vari moduli di Rails Active Record, Action Controller e Action View.

Config

Inizialmente questa directory contiene informazioni relative a tre cose: gli environment, i dati di connessione al database e le route. Gli *environment* sono un'altra interessantissima funzionalità di Rails meriterebbe un'ampia trattazione. In breve, esistono tre ambienti che hanno caratteristiche predefinite differenti e possono essere configurati indipendentemente per usare ad esempio diversi database o abilitando e disabilitando l'autenticazione, attivando il caching delle pagine o dei file eccetera. Per configurare nel dettaglio ogni ambiente si deve intervenire sul file config/environment/nomeambiente.rb, mentre per configurazione condivise andrebbe modificato il file config/environment.rb. Gli ambienti citati sono:

- *development*, che si usa in fase di scrittura del codice quando si vogliono molti log, si ricaricano spesso i file sorgente, si notificano errori, eccetera;
- *test*, nel quale si vuole un sistema isolato per avere risultati ripetibili utili ad effettuare test;
- *production*, da utilizzare quando l'applicazione è aperta agli utenti finali.

Le opzioni di accesso al database vengono invece controllate da un singolo file, config/database.yml che è scritto in formato YAML.

Il file routes.rb contiene infine le associazioni tra un URL ed una determinata azione.

Db

In questa directory verranno mantenute informazioni sul database, listati SQL e codice ruby relativo alle migration. Le *migration* sono una potentissima funzionalità di Rails tramite la quale è possibile effettuare modifiche incrementali al proprio database facendolo evolvere nel tempo, con la possibilità aggiuntiva di poter ritornare a stati precedenti se lo si desidera.

Doc

Il luogo dove raccogliere la documentazione relativa al progetto, inizialmente contiene solo un file "README" di default. La cartella viene popolata attraverso vari task Rake (in seguito maggiori dettagli) che generano guide e documentazione.

Lib

Le funzionalità non prettamente legate al lato web dell'applicazione vanno inserite in questa directory. Ad esempio se si ha un modulo per convertire dei dati o per effettuare calcoli, o per interagire con sistemi esterni, questo è il posto in cui mantenerla. In generale se una cosa non è un modello, controller, helper o vista, va messa in lib.

Qui si tova anche una sottocartella vuota tasks dove poter inserire alcuni comandi rake scritti dal programmatore. Rake è un'utility scritta in Ruby per eseguire comandi predefiniti. Questa libreria è universalmente riconosciuta come la versione Ruby del celebre make. Rake permette di creare task completamente definiti in Ruby per eseguire qualsiasi tipo di comando, dalla compilazione di un programma alla manutenzione di un filesystem. Rails ne mette a disposizione un bel pò; per visualizzarne la lista si lanci il comando rake -T.

Log

Contiene i log del webserver in esecuzione, che raccolgono informazioni su statistiche di tempo, cache e database. Qui si possono trovare tre principali file, chiamati development.log, test.log, e production.log. Ognuno di questi è usato a seconda del tipo di enviroment con il quale si sta lavorando.

Public

Questa directory è la faccia esterna dell'applicazione, ovvero la root del webserver. Qui vanno messi file HTML statici, immagini, JavaScript, CSS e forse anche alcune pagine web. Per questi ultimi tre esistono delle specifiche sottodirectory. Anche in questo caso si capisce che questa piccola convenzione permette alle applicazioni Rails di essere molto omogenee tra loro.

Script

In questa cartella sono presenti alcuni piccoli script, eseguibili da riga di comando, che permettono di fare molte operazioni utili.

Test

Questa directory è destinata a contenere i test che si sviluppano per l'applicazione. In particolare essa conterrà test relativi ad ogni singola parte dell'applicazione (controller, modelli, viste) e test che invece attraversano tutti gli altri strati. Inoltre in questa directory si potranno mantenere dei dati di prova utili a far girare i test (le cosi dette *fixture*).

Tmp

Contiene file temporanei organizzati in sottocartelle per i contenuti di cache, sessioni, e socket. Generalmente questi file vengono cancellati automaticamente da Rails dopo un certo periodo.

Vendor

In un qualsiasi progetto ci si ritrova a utilizzare librerie di terze parti, ed esse vanno mantenute in questa directory. Un particolare tipo molto usato sono i plugin, ovvero delle piccole librerie che contengono funzionalità che estendono Rails e che possono essere installate automaticamente tramite il comando script/plugin e salvate dentro la cartella vendor/plugins.

4.2 Convenzioni sui nomi

Come è stato già detto, Rails preferisce le convenzioni alle configurazioni, e quindi cerca di evitare allo sviluppatore il peso di dover specificare l'associazione ad esempio tra tabelle e classi. Tutto ciò avviene grazie alla " conoscenza dell'inglese " da parte del framework.

Per l'associazione tra tabelle e classi del modello è sufficiente che le tabelle siano chiamate con il plurale del nome della classe e che siano scritte in minuscolo. La tabella messages sarà dunque mappata sulla classe Message, authors su Author e topics su Topic. Il meccanismo di ActiveRecord che si occupa di questa conversione è molto più intelligente di quel che si potrebbe pensare e ad esempio è in grado di capire che il plurale di "person" è "people" o che il plurale di "status" è "statuses". Se questa convenzione non piace, ovviamente può non essre seguita, a patto di specificare esplicitamente il nome delle tabelle all'interno della classe.

I controller di Rails hanno altre convenzioni; per ognuno di essi, ad esempio un "prova controller" valgono le seguenti regole:

- Rails assume che la classe sia chiamata ProvaController e sia implementata dentro il file prova_controller.rb nella cartella app/controllers.
- Inoltre ci sia un modulo helper chiamato ProvaHelper nel file prova_helper.rb nella cartella app/helpers.
- Rails allora guarda nella cartella app/views/prova se c'è una vista per questo controller.
- Di default prende l'output di queste viste e lo piazza nel modello del layout contenuto nel file prova.html.erb o prova.xml.erb nella directory app/views/layouts.

Denominazione Modelli

Tabella	products
File	app/models/product.rb
Classe	Product

Denominazione Controller

URL	http://../store/list
File	app/controllers/store_controller.rb
Classe	StoreController
Metodo	list
Layout	app/views/latouts/store.html.erb

Denominazione Viste

URL	http://../store/list
File	app/views/store/list.html.herb (o .builder o .rjs)
Helper	modulo StoreHelper
File	app/helpers/latouts/store_helper.rb

5 Creare una semplice applicazione

5.1 Creare un progetto

Per iniziare verrà creata una piccola applicazione con pochi elementi essenziali, utile a comprendere meglio gli aspetti più profondi che verranno presentati in seguito. Si tratta di un progetto chiamato `blog`, ovvero un semplicissimo weblog.

Per iniziare, aprire un terminale, posizionarsi dove si vuole creare il progetto e digitare il comando `rails new blog`. Questo comando genera la cartella `blog`, che avrà tutte le sottocartelle esaminate nel capitolo precedente.

5.2 Configurare il database

Praticamente ogni applicazione degna di questo nome utilizza un database, specificato nel file di configurazione `config/database.yml`. Se si apre il file in una nuova applicazione si noterà che il database di default è SQLite3. Il file contiene sezioni per tre differenti ambienti nei quali Rails può girare:

- ambiente di *sviluppo*, usato in fase di realizzazione dell'applicazione;
- ambiente di *test*, usato per eseguire test automatizzati;
- ambiente di *produzione*, quando l'applicazione è pronta ed è utilizzata dagli utenti nel mondo.

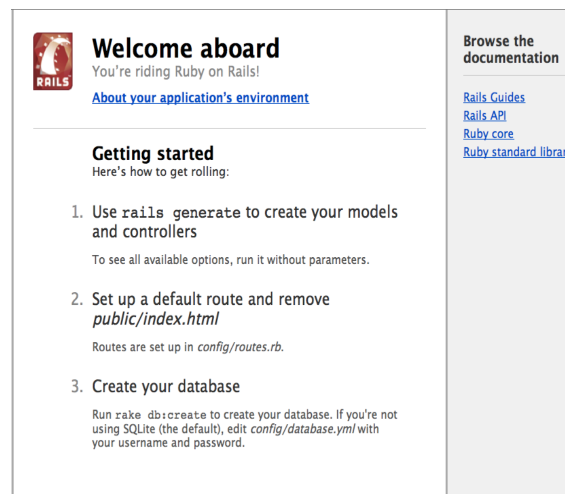
SQLite3 è appunto installato con Rails e non ci sono comandi da eseguire o account utenti da configurare; tuttavia esso può comunque essere cambiato con altri dbms (MySQL, PostgreSQL, Oracle, DB2, Firebird e SQL Server) e può richiedere diverse impostazioni. Ora si può creare il database vuoto con il comando `rake db:create` che si va a posizionare all'interno della cartella `db`.

5.3 Hello Rails!

Tanto per iniziare, come di consueto si fa con un nuovo linguaggio di programmazione, si vuole visualizzare del testo a schermo. Per farlo però bisogna prima far partire il web server con il comando `rails server`, il quale di default lancia un'istanza di WEBrick web server (ma Rails può usare molti altri tipi di server). Per vedere l'applicazione in azione bisogna aprire una finestra del browser e navigare

alla pagina http://localhost:3000. Infatti, come ogni altra applicazione web, anche quelle Rails appaiono all'utente associate ad un URL e quando si punta a quel determinato indirizzo si parla con il codice dell'applicazione, che a sua volta genera una risposta.

Se il sistema è configurato correttamente si dovrebbe visualizzare la seguente schermata di benvenuto; cliccando sul link About si può anche visualizzare un riassunto sull'ambiente dell'applicazione.



Per ottenere il classico "Hello" c'è bisogno di creare un minimo di un controller e una vista; fortunatamente si può fare con un singolo comando: rails generate controller Say hello. Rails creerà diversi file in automatico, inclusi i file

- app/controllers/say_controller.rb che contiene il codice che soddisferà la richiesta del browser;

- app/views/say/hello.html.erb che verrà usato per visualizzare il risultato dell'esecuzione del metodo action di hello nel controller Say; lo si apra e si digiti ¡h1¿Hello, Rails!¡/h1¿.

Se ora si punta come prima alla pagina http://localhost:3000 si visualizza ancora la schermata di benvenuto. Perchè?

Come ogni altra applicazione web, un'applicazione Rails appare all'utente associata ad un URL. Quando si punta il browser a quell'URL si sta parlando con il codice dell'applicazione, ovvero con un controllore, che genera una risposta all'utente. Poiché dall'URL attuale non si evince quale controllore debba incaricarsi di

dare una risposta, Rails visualizza la pagina di benvenuto. Se però ora si punta il browser a http://localhost:3000/say/hello, a schermo apparirà la scritta di saluto desiderata. Il metodo con cui Rails gestisce il routing interpreta gli URL e cioè di come viene data risposta all'utente verrà discusso nel capitolo x.

5.4 Flusso di un'applicazione

Dopo aver costruito un'applicazione semplicissima, è utile fin da subito capire come funzionano i suoi meccanismi basilari.

Una volta che il server è avviato, cosa è successo da quando è stata richiesta la pagina a quando è stata visualizzata la risposta?

- Il browser ha richiesto all'applicazione una pagina indicandone l'indirizzo URL http://localhost:3000/say/hello.
- L'applicazione ha ricevuto la richiesta interpretando l'indirizzo per decidere quale controller attivare; ha perciò creato una nuova istanza della classe SayController che si trova in app/controllers/say_controller.rb.
- Rails ha poi invocato il metodo action hello del controller, il quale esegue il proprio codice.
- Successivamente Rails cerca un template da visuailizzare nella directory app/views/say e la sua scelta ricade su hello.html.erb.
- Infine la pagina HTML è stata inviata al browser per la visualizzazione.

Naturalmente non è tutto qui, anzi Rails fornisce molte opportunità di intervenire in questo flusso di operazioni e modificare o sovrascrivere leggermente alcune azioni. In ogni caso verrà sicuramente preservato il mantra *convenction over configuration* di questo framework.

Una volta ricevuta una richiesta dal client, l'applicazione deve individuare il controller e la action da attivare. Per effettuare questo collegamento l'applicazione consulta le regole di routing che conosce estraendole dal file di configurazione config/routes.rb.

Per avere una panoramica delle richieste attualmente riconosciute dalla nostra applicazione Rails e delle action attivate per ognuna di queste inseriamo dal terminale (dopo esserci assicurati di aver disattivato la console Rails) il comando rake routes.

5.5 Lo scaffolding

Lo scaffolding (da scaffold: impalcatura) è un modo rapido per generare alcune fra le parti più importanti di un'applicazione. Se si desidera creare modello, viste e controller per una nuova risorsa in una sola operazione, è lo strumento adatto per risparmiare gran parte del lavoro.

Nel caso del weblog, se si vuole una risorsa che serva a rappresentare i post che faranno parte del blog, basta eseguire sul terminale il comando

```
rails generate scaffold Post name:string title:string content:text
```

Il generatore creerà una serie di file e cartelle, fra i quali forse il più importante è un file che si occupa della migrazione del database (il meccanismo delle migrazioni verrà esaminato in dettaglio nel prossimo capitolo). Ora non resta che eseguire la migrazione con il comando

```
rake db:migrate
```

6 Modelli

L'importanza di questa struttura è già stata sottolineata in precedenza ed ora verrà esaminato in dettaglio ogni aspetto di Active Record, il modulo di Ruby on Rails gestisce la persistenza dei dati, utilzzando un esempio che verrà via via arricchito per esporre le principali e fondamentali operazioni da svolgere per manipolare modello e database.

6.1 Definire le classi del modello

Tutto ciò che nell'applicazione ha a che vedere con un'entità del database ha bisogno di una classe nella cartella models.

Si supponga di avere la necessità di tenere traccia di una lista di prodotti per una applicazione. Bisognerà quindi inserire una classe Procuct con tutti i suoi attributi nella cartella models. Usando lo scaffolding visto nel capitolo x, Rails svolge tutto il lavoro per il programmatore:

```
rails generate scaffold Product \
title:String description:text; image_url:string price:decimal
```

Cosa ha prodotto questo comando?

- creato il file app/models/product.rb che rappresenta la classe nel modello;

  ```
  class Product < ActiveRecord::Base
     attr_accessible :description, :image_url, :price, :title
  end
  ```

- creato il file db/migrate/201107110000001_create_products.rb che riguarda le migrazioni e verrà analizzato nella prossima sezione;
- altri file in altre cartelle dell'applicazione, anch'essi approfonditi nei prossimi capitoli.

Cosa più importante di tutte, Active Record ha automaticamente mappato la classe nella sua corrispondente tabella dello schema relazionale del database. Tutti i suoi attributi diventano colonne della tabella stessa, secondo la convenzione dei nomi già citata in x.

Inoltre, com'è noto, ogni tabella deve anche avere una chiave primaria e Active Record fornisce una colonna chiamata id (che può comunque essere modificata manualmente) di tipo intero incrementale che serve a questo scopo. Essa non è la sola ad essere aggiunta, ecco la lista completa di quelle che hannno funzioni ausiliari ma importanti:

- created_at, created_on, updated_at, updated_on tengono traccia del *timestamp* di creazione e ultima modifica delle righe che andranno a comporre la tabella.
- lock_version memorizza la versione della riga.
- type : Active Record supporta le sottoclassi e in questo caso tutti gli attributi di tutte le sottoclassi sono tenute insieme nella stessa tabella. L'attributo type è usato per nominare la colonna che terrà traccia del tipo di una riga.
- xxx_id è il nome di default della chiave esterna riferita alla tabella chiamata con il plurale della forma xxx

6.2 Specificare relazioni fra modelli

Active Record supporta tre tipi di relazioni fra tabelle: uno-a-uno, uno-a-molti, molti-a-molti e vengono realizzate aggiungendo alcune dichiarazioni ai modelli coinvolti.

6.2.1 Relazione uno-a-molti

Una relazione di questo tipo permette di rappresentare una collezione di oggetti. Ad esempio, un ordine potrebbe avere una serie di articoli (line items) e quindi nel database tutte le righe di articolo di un particolare ordine contengono la chiave esterna riferita all'ordine. I modelli perciò vengono così modificati:

```
class Order < ActiveRecord::Base
has_many : line_items
........
end

class LineItem < ActiveRecord::Base
```

```
belongs_to : order
........
end
```

Nel definire la relazione uno-a-molti con has_many e belongs_to, è possibile specificare differenti opzioni che modificano il comportamento della relazione.

6.2.2 Relazione uno-a-uno

Questa relazione è implementata usando una chiave esterna in una riga di una tabella, per referenziare al massimo una sola riga in un'altra tabella. Ad esempio la tabella orders (corrispondente al modello Order) è legata alla tabella invoices da questo tipo di relazione, perciò si hanno le seguenti modifiche alle classi:

```
class Order < ActiveRecord::Base
has_one : invoice
........
end

class Invoice < ActiveRecord::Base
belongs_to : order
........
end
```

La regola contenuta implicitamente nel codice è che la classe per la tabella che contiene la chiave esterna deve sempre avere la dichiarazione belongs_to.

Se ad esempio le variabili @invoice e @order contenengono rispettivamente un'istanza del modello Invoice ed una del modello Order, possiamo utilizzare le operazioni:

- @order.invoice = @invoice o @invoice.order = @order per mettere in relazione uno-a-uno le due istanze.
- @order.invoice = nil per annullare la relazione.
- @order.invoice.nil? restituisce true se non è presente alcuna relazione, false altrimenti.

6.2.3 Relazione molti-a-molti

Seguendo il filo dell'esempio, si supponga di voler categorizzare alcuni ipotetici prodotti. Un prodotto può appartenere a molte categorie, e ogni categoria può contenere molti prodotti. Questa è una relazione molti-a-molti che si può esprimere con la dichiarazione has_and_belongs_to_many aggiunta ad entrambi i modelli:

```
class Product < ActiveRecord::Base
has_and_belongs_to_many : categories
........
end

class Category < ActiveRecord::Base
has_and_belongs_to_many : products
........
end
```

Rails implementa l'associazione usando una tabella intermedia di join contenente le coppie di chiavi esterne delle due tabelle coinvolte ed il nome assunto da questa nuova tabella è la concatenazione dei due nomi di quest'ultime in ordine alfabetico. In questo esempio si avrà la tabella di join categories_products.

Ora è possibile accedere alle categorie abbinate a un prodotto con @products.categories, e ai prodotti relativi ad una categoria con @category.products.

6.3 Operazioni con i modelli (CRUD)

Con Active Record si ha la mappatura tra classi e tabelle ma non è la sola, ce n'è un'altra: ogni nuova istanza della classe corrisponde ad una nuova riga della tabella dove tutti i valori degli attributi riempiono i valori delle colonne. Per agire su di esse bisognerebbe usare i costrutti del linguaggio SQL ma, ancora una volta, Rails corre in aiuto del programmatore attraverso metodi di Active Record. Essi sono semplici e intuitivi e resta comunque la possibilità di integrare pezzi o parti intere di codice SQL. Di seguito sono esposte le operazioni più comuni da applicare al database.

6.3.1 Inserire nuove righe: i metodi new e create

Nell'implementazione Rails la creazione di una nuova istanza della classe non crea automaticamente una nuova riga, ma si limita a preparare tutti i dati in memoria in attesa di una esplicita richiesta di salvataggio nel database con il metodo save.

Per creare una nuova istanza si utilizza il metodo new del modello al quale si è interessati. Per creare una nuova istanza di Order si lancia il comando

```
an_order = order.new(
     name : "Diego"
     email : "cdiego89@alice.it"
     address : "via Buratti 6"
     pay_type : "check")
an_order.save
```

Senza la chiamata al metodo save l'ordine esisterebbe solo in memoria locale. Il metodo create risparmia quest'ultima incombenza.

```
an_order = order.create(
     name : "Diego"
     email : "cdiego89@alice.it"
     address : "via Buratti 6"
     pay_type : "check")
```

Ad entrambi i metodi possono anche essere passati array di attributi hash; verranno create più righe e la variabile locale conterrà l'array delle corrispondenti istanze della classe:

```
orders = order.create([
   {  name : "Diego"
      email : "cdiego89@alice.it"
      address : "via Buratti 6"
      pay_type : "check")
   },  {.......}, {......}])
```

Si possono passare anche parametri di un form, così da rendere il lavoro molto snello!

```
@order = Order.new(params[:order])
```

Per quanto riguarda la chiave primaria, questi metodi la inseriscono automaticamente impostando ovviamente un valore unico al campo id di default.

6.3.2 Interrogare il database

Tutti i metodi che verranno presentati corrisponderanno ad un'interrogazione di tipo SELECT a livello SQL.

Il modo più semplice per individuare una riga in una tabella è quello di utilizzare il metodo find specificando la chiave primaria:

```
an_order = Order.find(27)
```

Il risultato di questa operazione è un oggetto corrispondente alla riga con tale chiave. Se venissero specificate più chiavi il metodo restituirebbe un numero di oggetti pari alle chiavi primarie trovate. Se non ci sono righe con le chiavi richieste, il metodo lancia l'eccezione ActiveRecord::RecordNotFound.

Se si vuole invece fare una ricerca in base al valore di un attributo (caso molto più frequente) Rails mette a disposizione varianti del metodo find. Eccone alcuni esempi:

```
order = Order.find_by_name("Carraro Diego")
orders = Order.find_all_by_name("Carraro Diego")
order = Order.find_by_name_and_email("Carraro Diego",
"cdiego89@alice.it")
```

Nel primo caso si reperisce solo la prima riga trovata che contiene il nome indicato, mentre nel secondo tutte quante. L'ultimo restituisce la prima riga che ha la combinazione di nome e mail evidenziata. Se nei tre casi non viene trovato alcun record, il metodo restituisce nil. Active Record infatti converte ciò che si trova dopo il by in nomi di colonne sulle quali effettuare la ricerca.

Ovviamente non è tutto qui, anzi. Rails fornisce molti altri metodi che non verranno trattati ma che sono comunque presenti nella documentazione ufficiale per effettuare praticamente ogni tipo di query. Si ricorda che in ogni caso è possibile affidarsi al classico codice SQL utilizzando il metodo find_by_sql

```
orders=LineItem.find_by_sql(query scritta in codice sql)
```

che naturalmente restituisce un array di oggetti (potenzialmente vuoto).

6.3.3 Modificare righe del database

L'operazione può essere svolta da tre tipi di metodi: save, update_attribute,update.

```
order=Order.find(89)
order.name="Diego"
order.save
```

Active Record modificherà l'eventuale record trovato con id 89, altrimenti il metodo save inserirà una nuova riga (se gli altri attributi possono essere nulli).

```
order=Order.find(89)
order.update=(:name,"Diego")
```

Fa la stessa cosa risparmiando la chiamata al metodo save: inoltre c'è anche una versione che agisce su più attributi contemporaneamente.

```
order=order.update(89, name: "Diego", email: "cdiego89@alice.it")
```

Combina le funzioni di lettura e update sulla riga con id 89. Come parametri possono essere passati anche un array di id e un array hash di valori di attributi ed il metodo farà l'update o l'inserimento di tutti i record.

6.3.4 Cancellare righe del database

Rails supporta due stili di cancellazioni, il primo fa uso di delete e delete_all.

```
Order.delete(123)
Order.delete([4,8,15,16,23,42])
```

cancellano le righe con gli id corrispondenti e ritornano il numero delle righe cancellate, mentre

```
Product.delete_all(["price > ?", @expensive_price])
```

cancella i prodotti che soddisfano la condizione indicata.

Il secondo stile fa uso del metodo destroy che cancella dal database la riga corrispondente ad un certo oggetto del modello:

```
order=Order.find_by_name("Diego")
order.destroy
```

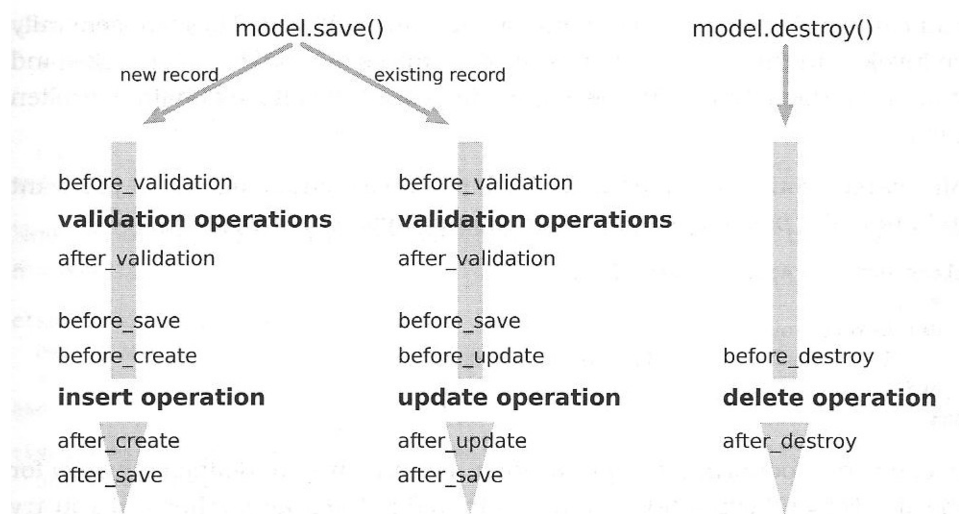
Il seguente metodo funziona concettualmente come il primo ma prende una condizione per evidenziare le righe e poi richiama il metodo destroy per eliminarle:

```
Order.destroy_all(["shipped_at < ?", 30.days.ago])
```

Sostanzialmente delete e destroy si equivalgono, con la differenza che il primo bypassa le varie chiamate di Active Record e alcune funzioni di validazione (mantengono coerente il database con le regole definite nelle classi del modello) , mentre il secondo le richiama tutte. In conclusione quindi il metodo destroy sembra essere il più adatto.

6.3.5 Ciclo di vita dei modelli

Active Record controlla il ciclo di vita degli oggetti del modello: li crea, li salva, li aggiorna e li cancella. Usando le *chiamate*, essa permette al codice del programmatore di partecipare al monitoraggio di queste operazioni. Si può infatti scrivere codice che venga invocato ad ogni evento significativo della vita di un oggetto e che esegua delle validazioni complesse o mappature sui valori. In figura sono indicate sedici chiamate di questo tipo:



Per usare codice durante le chiamate ci sono due modi.

Il preferito è dichiarare un handler, che può essere un metodo o un blocco. Si associa un handler con un particolare evento usando i metodi della classe che vuole essere monitorata. Per associare un metodo bisogna dichiararlo privato o

protetto e specificare il suo nome come simbolo per la dichiarazione dell'handler. Se si tratta di un blocco invece, lo si aggiunge dopo la dichiarazione. L'esempio comprende entrambi le scelte.

```
class Order < ActiveRecord::Base
  before_validation : normalize_credit_card_number
  after_create do |order|
    logger.info ........
  end
  protected
  normalize_credit_card_number
    self.cc_number.gsub!(.......
  end
end
```

Alternativamente si può definire la chiamata direttamente con un metodo d'istanza. Ad esempio

```
class Order < ActiveRecord::Base
  #......
  def before_save
    self.payment_due || = Time.now + 30.days
  end
end
```

Sono presenti ulteriori tecniche per potenziare il meccanismo delle chiamate, ma non verranno trattate.

6.4 Transazioni

Una transazione raggruppa una serie di modifiche al database in modo che esse o vengano applicate tutte con successo, oppure non ne venga applicata alcuna. Cioè se anche solamente una di queste non va a buon fine, l'intera transazione fallisce.

In Active Record si usa il metodo `transaction` invocato su un modello per realizzare una transazione; alla fine del blocco di operazioni la transazione è completa e il database è aggiornato, a meno che non si verifichi un'eccezione: in questo caso il database annulla i cambiamenti già effettuati. Ecco un esempio:

```
Account.transaction do
  account1.deposit(100)
  account2.withdraw(100)
end
```

La transazione protegge il database dal diventare inconsistente, ma cosa succede agli oggetti del modello? Se infatti una transazione fallisce, essi comunque saranno aggiornati e quindi rispecchierebbero una situazione errata. La soluzione è quella di intercettare le eccezioni all'interno del codice e sistemare " manualmente" lo stato degli oggetti.

6.5 Le migrazioni

Nella realizzazione di un'applicazione ci si trova di fronte ad una serie di scelte che portano a commettere errori tali da dover cambiare le cose di volta in volta. In particolare, per quanto riguarda lo schema del database, esso viene costantemente riveduto aggiungendo tabelle, rinominando colonne, eccetera. Perciò il monitoraggio degli stati in cui si trova di volta in volta il database e la gestione di queste modifiche è possibile grazie alle *migrazioni*. Esse sono in grado, in pratica, di ricotruire la storia dell'evoluzione del database.

6.5.1 Creare una migrazione

Fisicamente, una migrazione è un file nella cartella db/migrate dell'applicazione, che può essere creato manualmente oppure utilizzando un generatore. Di solito si preferisce usare il secondo metodo in quanto risparmia tempo ed evita possibili errori. Attualmente ci sono due generatori che svolgono questo compito:

- Il *generatore di modelli* crea una migrazione associata con il modello. Ad esempio, creando un modello chiamato discount si crea anche una migrazione chiamata yyyyMMddmmss_create_discount.rb:

  ```
  project> rails generate model discount
          invoke     active_record
          create       db/migrate/20110608133549_create_discounts.rb
          create       app/models/discount.rb
  ```

- Oppure si può crearne una indipendente dal modello:

```
project> rails generate migration add_price_column
        invoke     active_record
        create      db/migrate/20110412133549_create_discounts.rb
```

Analizzando il nome del file di migrazione creato, si può notare è composto da una sequenza di quattordici cifre, un underscore e una stringa (dipendente dal modello nel primo caso o dal nome scelto nel secondo).

Le quattrodici cifre indicano il timestamp in anno, mese, giorno, ora, minuto, secondo del momento in cui la migrazione viene creata e praticamente costituiscono il *numero di versione* della migrazione.

6.5.2 Eseguire una migrazione

Le migrazioni sono eseguite lanciando il Rake task db:migrate :

```
project> rake db:migrate
```

Il codice di migrazione mantiene una tabella chiamata schema_migrations in ogni database Rails. Questa tabella ha solo una colonna chiamata version che ha una riga per ogni migrazione applicata con successo. Le seguenti fasi si susseguono quando viene lanciato il comando scritto sopra:

- Viene verificata la presenza della tabella schema_migrations che se non esiste viene creata.
- Si esaminano tutti i file di migrazione all'interno della cartella db/migrate in base al numero di versione, non prendendo in considerazione quelli che sono già presenti nel database. Quindi per ognuno di quelli non presenti si applica la migrazione aggiungendo una riga alla tabella schema_migrations.
- Se a questo punto si rilancia il comando Rake non succede nulla poichè tutte le versioni dei file di migrazione corrispondono ad una riga della tabella.

Grazie a questa ingegnosa struttura, si può forzare il database ad una specifica versione con il comando

```
project> rake db:migrate VERSION=20110412133549
```

Se la versione è più grande di tutte le migrazioni finora applicate, tutte le migrazioni non ancora eseguite vengono applicate ora (praticamente equivale al comando rake db:migrate). Se invece la versione è compresa fra quelle della tabella schema_migrations, Rails riporta il database a quella versione. In questo modo è possibile passare da una versione all'altra del database mantenendo comunque la consistenza dei dati.

6.5.3 Anatomia di una migrazione

Una migrazione è una sottoclasse di ActiveRecord::Migration che implementa due metod: up e down.

```
class AddEmailToOrders < ActiveRecord::Migration
  def up
    add_column :orders, :e_mail, :string
  end

  def down
    remove_column :orders, :e_mail
  end
end
```

è la classe all'interno del file 20110711000017_add_email_to_orders.rb. Il metodo up si occupa di applicare i cambiamenti al database per questa migrazione, mentre down li annulla.

Ancora meglio, al posto di implementare entrambi i metodi, si può scegliere di usare il metodo change, con il quale Rails riesce a riconoscere se aggiungere o rimuovere una colonna a seconda dell'azione fatta in precedenza.

```
class AddEmailToOrders < ActiveRecord::Migration
  def change
    add_column :orders, :e_mail, :string
  end
end
```

Un'altra potenzialità di Rails è data dalla gestione dei tipi di dati di una colonna del database (che corrisponde ad un attributo del modello o della classe che rappresenta la migrazione). Nell'esempio il campo email è di tipo string, anche se

nessun database supporta questo tipo. Rails prova a rendere le proprie applicazioni indipendenti dal database sottostante e riesce a mappare il tipo string nel corrispondente tipo adatto al database che si sta usando. Per esempio, il tipo string crea una colonna di tipo varchar(255) se si lavora con SQLite3 mentre di tipo char varying se si lavora con Postgres.

Si possono specificare ulteriori opzioni sulle colonne di una migrazione, che corrispondono a vincoli sulle colonne del database:

- null: true o false permette o meno all'attributo di essere nullo;
- limit: size impone un limite alla taglia del valore del campo;
- default: value setta il valore di default di una colonna.

Ecco alcuni esempi:

```
add_column :orders, :attnl, :string, limit:100
add_column :orders, :ship_class, :string, null:false,
default: 'priority'
add_column :orders, :order_type, :integer
```

6.5.4 Gestire le tabelle

Per ora si è visto come usare migrazioni per manipolare le tabelle esistenti, ma ovviamente le migrazioni possono aggiungere o rimuovere tabelle.

Creare e rimuovere tabelle

```
class CreateOrderHistories < ActiveRecord::Migration
  def change
    create_table :order_histories do |t|
      t.integer :order_id, null: false
      t.text :notes

      t.timestamps
    end
  end
end
```

create_table prende il nome della tabella che deve essere creata e un blocco che viene usato per definire le colonne. Se il metodo viene chiamato all'interno del metodo change, esso potrebbe eliminare la tabella passata come parametro, ovvero come se si chiamasse il metodo drop_table. Come al solito se non viene specificata la chiave primaria Rails aggiunge il campo id. Il timestamp crea le colonne created_at e updated_at.

Come di consueto, la chiamata di create_table può prendere ulteriori parametri che non verranno trattati.

Tutti i metodi descritti in questo capitolo sono disponibili come metodi degli oggetti Active Record e quindi accessibili dentro modelli, viste e controller dell'intera applicazione.

7 Action Pack

Ora verrà analizzato il processo di funzionamento dell'Action Pack, che si trova nel cuore delle applicazioni Rails. Esso si compone di tre moduli: *Action Dispatch*, *Action Controller* e *Action View*. Il primo instrada richieste ai controller, il secondo converte tali richieste in risposte e il terzo le formatta per renderle visibili all'utente.

7.1 Action Dispatch

Da un punto di vista semplice, un'applicazione web accetta richieste in ingresso da un browser, le processa, e ritorna una risposta. Rails codifica le informazioni nella richiesta URL e usa il sottosistema chiamato *Action Dispatch* per determinare cosa fare con questa richiesta.

Rails adotta due strategie per fare routing di richieste:

- basato sulle risorse, ovvero adottando il paradigma REST;
- basato su pattern matching, requisiti e condizioni.

In entrambi i casi il processo determina il controller e l'azione preposti alla risposta. Di seguito verrà presentato solo il primo dei due approcci, poiché esso è il più interessante e tuttora il più utilizzato dai programmatori.

7.1.1 Routing basato su REST

REST (Rapresentational State Transfer) è un paradigma che definisce un insieme di principi architetturali per la progettazione di un sistema, cioè non si riferisce ad un sistema concreto e ben definito né si tratta di uno standard stabilito da un organismo di standardizzazione. La sua definizione è apparsa per la prima volta nel 2000 nella tesi di Roy Fielding, "Architectural Styles and the Design of Network-based Software Architectures", discussa presso l'Università della California, ad Irvine. È bene precisare che i principi REST non sono necessariamente legati al Web, nel senso che si tratta di principi astratti di cui però il World Wide Web ne risulta essere un esempio concreto.

L'intero paradigma ruota attorno al concetto di *risorse*, che nel caso si stia usando il protocollo HTTP sono rappresentate dagli URL. La loro manipolazione

avviene per mezzo dei metodi GET, POST, PUT e DELETE del protocollo, detti *verbi*.

In Rails, il routing basato sulle risorse fornisce una mappatura fra i verbi HTTP e URL con le azioni dei controller. Per convenzione, ogni azione mappa a sua volta una particolare operazione CRUD nel database. Una singola entry nel file di routing config/routes.rb crea sette diverse rotte nell'applicazione, tutte mappate allo specifico controller. Di seguito il file routes.rb con le rotte e il codice del controller per l'esempio dei prodotti, ovvero il file /app/controllers/product_controller.rb.

```
Project::Application.routes.draw do |map|
  resources :products
end
```

Verbo HTTP	Path	Azione	Usato per
GET	/products	index	restituisce una lista di risorse
GET	/products/new	new	costruisce una nuova risorsa e la passa al client. Essa non viene salvata nel server. Questa azione può essere anche pensata come la creazione di in form vuoto da far riempire al client
POST	/products/new	create	crea una nuova risorsa dai dati, aggiungendola alla collezione
GET	/products/:id	show	restituisce il contenuto della risorsa identificata
GET	/products/:id/edit	edit	restituisce il contenuto della risorsa identificata in un form modificabile
PUT	/products/:id	update	aggiorna il contenuto della risorsa identificata
DELETE	/products/:id	destroy	distrugge la risorsa identificata

```
class ProductsController < ApplicationController
  # GET /products
  # GET /products.json
```

```
def index
  @products = Product.all

  respond_to do |format|
    format.html # index.html.erb
    format.xml
    format.json { render json: @products }
  end
end

# GET /products/1
# GET /products/1.json
def show
  @product = Product.find(params[:id])

  respond_to do |format|
    format.html # show.html.erb
    format.json { render json: @product }
  end
end

# GET /products/new
# GET /products/new.json
def new
  @product = Product.new

  respond_to do |format|
    format.html # new.html.erb
    format.json { render json: @product }
  end
end

# GET /products/1/edit
def edit
  @product = Product.find(params[:id])
```

```
end

# POST /products
# POST /products.json
def create
  @product = Product.new(params[:product])

  respond_to do |format|
    if @product.save
      format.html { redirect_to @product,
        notice: 'Product was successfully created.' }
      format.json { render json: @product, status: :created,
        location: @product }
    else
      format.html { render action: "new" }
      format.json { render json: @product.errors,
        status: :unprocessable_entity }
    end
  end
end

# PUT /products/1
# PUT /products/1.json
def update
  @product = Product.find(params[:id])

  respond_to do |format|
    if @product.update_attributes(params[:product])
      format.html { redirect_to @product,
        notice: 'Product was successfully updated.' }
      format.json { head :no_content }
    else
      format.html { render action: "edit" }
      format.json { render json: @product.errors,
        status: :unprocessable_entity }
```

```
      end
    end
  end

  # DELETE /products/1
  # DELETE /products/1.json
  def destroy
    @product = Product.find(params[:id])
    @product.destroy

    respond_to do |format|
      format.html { redirect_to products_url }
      format.json { head :no_content }
    end
  end

  def who_bought
    @product = Product.find(params[:id])
    respond_to do |format|
      format.html
      format.xml
      format.atom
      format.json { render json: @product.to_json(include: :orders) }
    end
  end
end
```

Da notare come nel controller si abbia un'azione per ogni corrispondente azione REST. I commenti prima di ognuna mostrano il formato dell'URL che l'ha invocata. Inoltre molte azioni contengono anche il blocco respond_to usato per determinare il tipo di contenuto da mandare in risposta.

Come ulteriore esempio, quando l'applicazione riceve la richiesta

```
DELETE /products/17
```

questa viene mappata in un'azione del controller. Se il primo matching nel file config/routes.rb è resources :products, Rails instrada questa richiesta al metodo destroy nel controller dei prodotti usando l'id 17.

Le risorse Rails forniscono un set iniziale di sette azioni come si è visto, ma c'è la possibilità di aggiungerne altre usando un'estensione alla chiamata resources:

```
Project::Application.routes.draw do |map|
  resources :products do
    get :who_bought, on: :member
  end
end
```

Con questa sintassi si dice di voler aggiungere una nuova azione chiamata who_bought, invocata con un HTTP GET. Essa è applicata ad ogni membro della collezione dei prodotti. Se si specifica :collection al posto di :member, l'instradamento è applicato all'intera collezione.

Spesso le risorse stesse contengono ulteriori collezioni di risorse. Ad esempio si potrebbe voler salvare la recensione di ogni utente su ogni prodotto. Ogni recensione è perciò una risorsa, e collezioni di recensioni sono associate con ogni singolo prodotto. Ancora una volta, Rails fornisce un modo semplice e intuitivo per l'instradamento per questo tipo di situazioni:

```
Project::Application.routes.draw do |map|
  resources :products do
    resources :reviews
  end
end
```

Poiché la risorsa review appare dentro il blocco dei prodotti, essa deve essere referenziata da una risorsa prodotto. Questo significa che il path di una review deve avere come prefisso uno specifico prodotto. Per identificare ad esempio la review con id 4 per il prodotto con id 99 si deve usare il path /products/99/review/4.

Per vedere l'insieme di rotte presenti in un dato momento si può lanciare il comando rake routes da terminale.

7.2 Action Controller

Nella sezione precedente si è visto come l'Action Dispatcher instrada una richiesta in ingresso nell'appropriato codice. Ora si analizzerà cosa succede dentro al codice.

7.2.1 Metodi e Azioni

Un controller è una sottoclasse di ApplicationController e ha metodi come qualsiasi altra classe. Quando l'applicazione riceve una richiesta, il routing determina quale controller e quale azione eseguire, quindi crea un'istanza di tale controller ed invoca il metodo con lo stesso nome dell'azione. Se il metodo non viene trovato e il controller implementa method_missing, quest'ultimo viene invocato. Ancora, se nemmeno questo esiste il controller cerca un template chiamato come il controller e l'azione correnti. Se nessuna di queste cose accade viene lanciato l'errore AbstractControllller::ActionNotFound.

Parte del lavoro di un controller è quindi rispondere all'utente e questo può essere fatto sostanzialmente in quattro modi:

1. Quello più comune è di visualizzare un template (detto *rendering* di un template). In termini dell'architettura MVC esso è una vista che prende informazioni fornite dal controller e le usa per generare una risposta all'utente.

2. Il controller può restituire una stringa al browser senza invocare una vista, di solito per notificare errori.

3. Il controller può mandare altri tipi di dati al client, tipicamente diversi dall'HTML (magari PDF o altre cose).

4. Infine può anche non essere restituito nulla. In questo caso comunque vengono inviati degli header HTTP perché una qualche risposta è sempre attesa.

I primi tre casi sono i più comuni e verranno presi in considerazione.

7.2.2 Visualizzare un template

Un *template* è un file che definisce il contenuto di una risposta per l'applicazione. Rails supporta tre formati per un template:

- *erb*, che è codice Ruby incorporato (tipicamente con HTML);
- *builder*, un modo più programmatico per costruire contenuti XML;
- *RJS*, che genera JavaScript.

Per convenzione, il template per l'azione *azione* del controller *controller* si troverà nel file app/views/controller/action.type.xxx, dove *type* è il tipo di file (html, atom o js), e *xxx* è uno fra erb, builder o js.

Il metodo render è il cuore di tutto il rendering in Rails. Esso prende un hash di opzioni che specificano cosa visualizzare e come farlo. Vedremo ora, per mezzo di esempi, le opzioni di rendering nei casi 1 e 2 sopra citati.

- render(): senza alcun parametro richiama il template di default per il controller e l'azione correnti. Il seguente codice visualizza il template app/views/blog/index.html.erb in tutti e tre i casi, coerentemente con ciò che è stato spiegato all'inizio della sezione 7.2.1.

```
class BlogController < ApplicationController
  def index
    render
  end
end

class BlogController < ApplicationController
  def index
  end
end

class BlogController < ApplicationController
end
```

- render(:text => string) manda al client la stringa specificata e non viene richiamato alcun template.

```
class HappyController < ApplicationController
  def index
    render(:text => "Hello there!")
  end
end
```

- render(:action => action_name) visualizza il template di default per l'azione specificata (naturalmente del controller in cui si trova), ma non invoca il metodo azione.

```
def display_cart
  if @cart.empty?
    render(:action => :index)
  else
    #...
  end
end
```

- render(:template => name, [:locals => hash]) visualizza il template specificato e organizza il testo da mandare al client (opzionale). Il valore :template deve contenere i nomi sia del controller che dell'azione, separati da un /. Il seguente codice visualizza il template app/views/blog/short_list.

```
class BlogController < ApplicationController
  def index
    render(:template => "blog/short_list")
  end
end
```

7.2.3 Mandare file e altri dati

send_data(data,option...) invia al client una stringa contenente dati binari. Tipicamente il browser usa una combinazione di tipo contenuti e disposizioni, entrambi specificati nelle opzioni, per determinare cosa fare con questi dati.

```
def sales_graph
  png_data = Sales.plot_for(Date.today.month)
  send_data(png_data, :type => "image/png", :disposition => "inline")
end
```

send_file(path,option...) invia al client il contenuto di un file.

```
def send_secret_file
  send_file("/files/secret_list")
  headers["Content-description"] = "Top Secret"
end
```

7.3 Action View

Il modulo ActionView incapsula tutte le funzionalità necessarie per visualizzare all'utente i template, di solito HTML, XML o JavaScript.

7.3.1 Utilizzare template

Quando si scrive una vista, si sta scrivendo un template. Per capire esattamente come lavorano, bisogna porre attenzione a tre aree:

- dove vengono collocati;
- l'ambiente nel quale vengono eseguiti;
- cosa ci va dentro.

Si è già visto che il metodo render si aspetta di trovare i template nella cartella app/views dell'applicazione. Al suo interno la convenzione è di avere una sotto-cartella per tutte le viste correlate ad ogni controller. Ognuna di queste cartelle contiene template chiamati con il nome dell'azione nel corrispondente controller. Se però si vuole salvare template in una cartella qualsiasi del filesystem, si può utilizzare la chiamata render(file: 'dir/template') . Tutto ciò è molto utile per condividere template tra diverse applicazioni.

I template contengono un misto fra testo e codice. Quest'ultimo aggiunge del contenuto dinamico alla risposta e viene eseguito in un ambiente che gli permette di avere accesso a informazioni impostate dal controller:

- tutte le variabili d'istanza del controller sono disponibili anche al template. Questo è il modo che hanno le azioni per comunicare con il template.
- L'istannza del controller corrente è accessibile usando l'attributo controller. Questo permette al template di chiamare qualsiasi metodo pubblico del controller.

- Infine il percorso dalla cartella radice dei template è salvato nell'attributo base_path.

Come è già stato spiegato nella precedente sezione, Rails supporta tre tipologie di template: *erb*, *builder* e *RJS*.

7.3.2 Utilizzare form

HTML fornisce un numero di elementi, attributi e valori corrispondenti, che controllano come gli input dell'utente vengono raccolti. Si può certamente inserire un form direttamente dentro il template; tuttavia Rails rende tutto ciò non necessario grazie all'utilizzo di *helper*. Di seguito un esempio di template usato per generare un tipico form.

```
<%= form_for(:model) do |form| %>

<p>
  <%= form.label :input %> <!-- <label id="helper.label"/> -->
  <%= form.text_field :input, :placeholder => 'Enter text here..' %>
            <!-- <label id="helper.text"/> -->
</p>

<p>
  <%= form.label :address, :style => 'float: left' %>
  <%= form.text_area :address, :rows => 3, :cols => 40 %>
            <!-- <label id="helper.textarea"/> -->
</p>

<p>
  <%= form.label :color %>:
  <%= form.radio_button :color, 'red' %>
            <!-- <label id="helper.radio"/> -->
  <%= form.label :red %>
  <%= form.radio_button :color, 'yellow' %>
  <%= form.label :yellow %>
  <%= form.radio_button :color, 'green' %>
```

```
  <%= form.label :green %>
</p>

<p>
  <%= form.label 'condiment' %>:
  <%= form.check_box :ketchup %>
            <!-- <label id="helper.checkbox"/> -->
  <%= form.label :ketchup %>
  <%= form.check_box :mustard %>
  <%= form.label :mustard %>
  <%= form.check_box :mayonnaise %>
  <%= form.label :mayonnaise %>
</p>

<p>
  <%= form.label :priority %>:
  <%= form.select :priority, (1..10) %>
            <!-- <label id="helper.select"/> -->
</p>

<p>
  <%= form.label :start %>:
  <%= form.date_select :start %>
            <!-- <label id="helper.date"/> -->
</p>

<p>
  <%= form.label :alarm %>:
  <%= form.time_select :alarm %>
            <!-- <label id="helper.time"/> -->
</p>
<% end %>
```

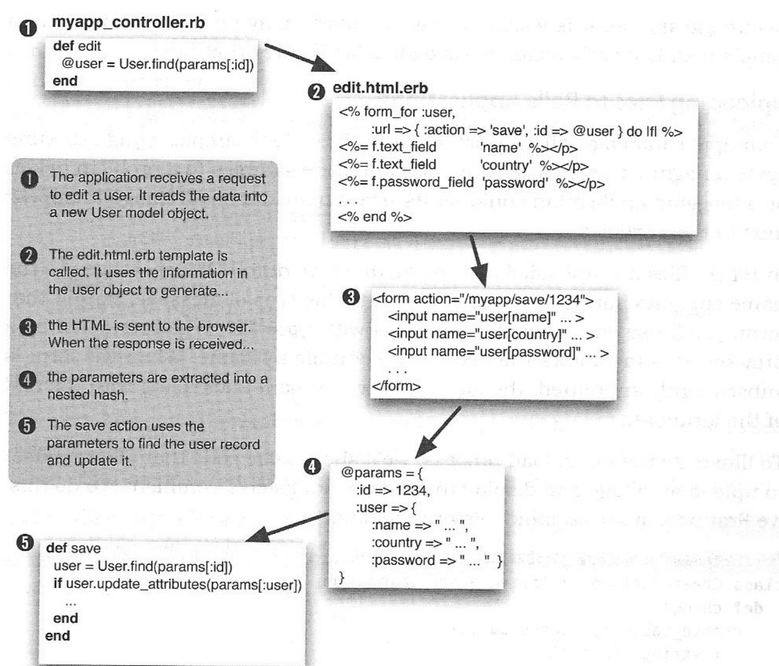
In questo template vengono usati gli helper:

- **text_field** e **text_area** per raccogliere campi a linea singola e multilinea;

- **search_field**, **telephone_field**, **url_field**, **email_field**, **number_field** e **range_field** per raccogliere specifici tipi di input;
- **radio_button**, **check_box** e **select** per fornire un insieme di opzioni da scegliere;
- **date_select** e **time_select** per mostrare data e ora.

Inoltre, se non si è soddisfatti, si possono reperire helper o plugin da integrare in Rails per potenziare alcune funzionalità dei form.

In figura, nel processo in cui si interagisce con un form, possiamo vedere come i vari attributi nel modello passino attraverso il controller, la vista, la pagina HTML e ancora indietro al modello.



L'oggetto del modello ha attributi come **name**, **country** e **password**; il template usa i metodi helper per costruire un form HTML per permettere all'utente di inserire dati nel modello. Da notare come sono nominati i campi del form; ad esempio l'attributo **contry** mappa un campo di input HTML con il nome **user**[**country**].

Quando l'utente conferma il form, i dati vengono inviati tramite POST all'applicazione. Rails estrae i campi dal form e costruisce l'hash **params**.

Nella parte finale di questo processo, gli oggetti del modello possono accettare nuovi valori per i propri attributi e quindi eseguire

```
user.update_attributes(params[:user])
```

7.3.3 Utilizzare Helper

Nella precedente sezione si è detto che è possibile integrare del codice Ruby nell'HTML anche se in realtà non è proprio una buona abitudine per vari motivi, primo fra tutti il fatto di mantenere bene separati i tre elementi concettuali dell'architettura MVC. Rails perciò fornisce un buon compromesso con l'impiego degli helper.

Un *helper* è semplicemente un modulo contenente metodi che assistono una vista ed esistono per generare HTML (o XML o JavaScript) e quindi estendere il comportamento di un template.

Di default, ogni controller ottiene il suo modulo helper; in aggiunta c'è un helper per l'intera applicazione chiamato **application_helper.rb**. Non sarà di certo una sorpresa scoprire che Rails fa certe assunzioni per aiutare a collegare gli helper nei controller e nelle loro viste. Anche se tutti gli helper sono disponibili a tutti i controller, è spesso buona pratica organizzarli per evitare confusione. Infatti, riferendosi come al solito all'esempio dei prodotti, gli helper che sono univoci per le viste associate a **ProductController** tendono ad essere collocati in un modulo helper chiamato **ProductHelper** nel file **product_helper.rb** della cartella **app/helpers**. Per fortuna il comando script **rails generate controller** crea e piazza il modulo automaticamente.

Di seguito un semplicissimo esempio per capire come si potrebbe fare per utilizzare un helper: supponiamo di avere una vista **index.html.erb** nella cartella **app/views/store**, che imposta dinamicamente il titolo della pagina.

```
<h3><%= @page_title || "Pragmatic Store" %></h3>
```

Spostiamo il codice all'interno del metodo dell'apposito helper; poiché siamo nello store controller, modifichiamo il file **store_helper.rb** all'interno della cartella **app/helpers**.

```
Module StoreHelper
  def page_title
    @page_title || "Pragmatic Store"
  end
end
```

Ora il codice della vista chiama semplicemente il metodo dell'helper:

```
<h3><%= page_title %></h3>
```

Rails fornisce una vasta gamma di metodi helper preconfezionati e disponibili per tutte le viste, i più importanti usati per formattare dati o per collegare tra loro pagine e risorse.

7.3.4 Layout e Partial

In questo capitolo si è visto che che i template sono pezzi di codice HTML isolati e la loro struttura fa si che a volte ci siano molti elementi duplicati, violando così il principio DRY (Don't Repeat Yourself). Infatti mediamente un website ha le seguenti ridondanze:

- molte pagine condividono intestazioni e sidebar;
- molte altre contengono gli stessi frammenti di HTML (un blog ad esempio visualizza un articolo in varie sezioni);
- le stesse funzionalità vengono usate spesso (alcuni siti ad esempio hanno un componente di ricerca standard che appare in molte sidebar).

Layout

Rails permette di visualizzare pagine nidificate dentro altre pagine. Tipicamente questa caratteristica è usata per mettere il contenuto di un'azione dentro una pagina HTML standard (titolo, piè di pagina e sidebar). Quando Rails prende in carico una richiesta di presentare un template da un controller, ne presenta due. Ovviamente quello che viene chiesto esplicitamente (o quello di default se l'azione è vuota), ma non solo; Rails tenta anche di trovare e presentare un layout che, se individuato, inserisce l'output dell'azione specifica nell'HTML prodotto da questo layout. Un possibile layout può essere il seguente:

```
<html>
<head>
  <title>Form: <%= controller.action_name %></title>
  <%= stylesheet_link_tag    'scaffold' %>
</head>
<body>

<%= yield :layout %>

</body>
</html>
```

Il layout imposta una pagina HTML standard con le sezioni head e body, usa il nome dell'azione corrente come titolo della pagina e include un file CSS.

Nel body c'è una chiamata a yield ed è qui che avviene la magia: quando il template dell'azione viene presentato, Rails ne salva il contenuto etichettandolo :layout. Dentro al layout, chiamando yield si recupera tale testo. Infatti :layout è il contenuto di default restituito durante la presentazione.

Se ad esempio il template my_action.html.erb contiene

```
<h1><%= @msg %></h1>
```

il browser vede il seguente HTML:

```
<html>
<head>
  <title>Form: my_action</title>
  <link href="/stylesheets/scaffold.css" media="screen"
rel="Stylesheet" type="test/css" />
</head>
<body>

<h1>Hello, World!</h1>

</body>
</html>
```

Partial

Mentre i layout servono a condividere il codice che fa da cornice alla pagina, i partial sono particolarmente utili nel caso in cui si vogliano riutilizzare degli elementi tra viste differenti e ancora di più quando ci sono elementi multipli.

Un *partial* non è altro che un ritaglio di codice, una sorta di subroutine che rappresenta un template. Esso può essere invocato una o più volte all'interno di un altro template passandogli come parametri oggetti da visualizzare. Quando il rendering finisce, il controllo ritorna al template chiamante. Il nome del file contenente questo codice deve cominciare con il carattere underscore, che serve a differenziarlo dagli altri tipi di template.

Per esempio, il partial che visualizza un ipotetico blog potrebbe essere salvato nel file _article.html.erb nella directory app/views/blog:

```
<div class="article">
  <div class="articleheader">
    <h3><%= article.title %></h3>
  </div>
  <div class="articlebody">
    <% article.body %>
  </div>
</div>
```

Altri template usano il metodo render(partial:) per invocarlo:

```
<%= render(partial:  "article", object: @an_article) %>
<h3> Add Comment </h3>
.............
```

Il parametro :partial del metodo render è chiaramente il nome del template da visualizzare (senza l'underscore). Il parametro :object invece identifica un oggetto che può essere passato nel partial. Questo oggetto sarà disponibile all'interno del template attraverso una variabile locale con lo stesso nome del template. In questo esempio @an_article è passato al template e quest'ultimo potrà accedervi usando la variabile locale article. Per questo è stato possibile scrivere article.title nel partial.

8 Il caching

Si è visto che Rails offre un potenza ed un'espressività enormi, ma al mondo non c'è niente che sia gratis. Sebbene Ruby sia un un interprete piuttosto veloce, Rails è composto da un numero notevole di astrazioni sulle funzionalità di base e per alcune applicazioni particolari questo potrebbe degradare troppo la velocità del framework, rendendolo inadatto. Ovviamente questo non è un problema specifico di Rails, ma di qualunque ambiente di livello molto alto; la presenza di un framework però può non essere solo una cosa negativa, in quanto è possibile fornire un meccanismo per affrontare questi problemi.

In particolare, Rails include un sistema potente di caching, che permette di raggiungere ottime prestazioni con una fatica ridotta. Esistono diverse applicazioni web scritte con questa piattaforma che sono in grado di gestire milioni di accessi al giorno senza avere setup hardware eccessivi, e anche se non sempre ci si trova di fronte a problemi di questo tipo, è importante sapere che esiste un modo provato per affrontarli.

Il sottosistema di caching di Rails lavora a tre livelli di granularità:

- caching delle pagine;
- caching delle azioni;
- caching dei frammenti.

8.1 Caching delle pagine

Il caching delle pagine è la più semplice ed efficiente forma di caching in Rails. La prima volta che un utente richiede un particolare URL, l'applicazione invoca e genera una pagina HTML e ne salva il contenuto in cache. Essa è accessibile dal web server, in modo che nelle richieste seguenti venga servito un file statico piuttosto che servire nuovamente la stessa richiesta, con operazioni che potrebbero avere un impatto prestazionale notevole.

Sebbene questo modello di caching non sia adatto a tutte le situazioni, esso permette un aumento di prestazioni impressionante, in quanto l'applicazione è in grado di servire le pagine alla stessa velocità che il server impiega a servire qualsiasi altro contenuto statico.

Nel seguente esempio, applichiamo questa ottimizzazione ai feed. Per far si che sia effettuato il caching della pagina relative ad rss è sufficiente aggiungere una riga al controller

```
class FeedController < ApplicationController
caches_page :rss
```

Come è evidente, un meccanismo di caching come questo è straordinariamente facile da applicare, ed è perfettamente ragionevole cercare di far si che alcune pagine particolarmente visitate siano riproducibili staticamente in questo modo. La cosa ad esempio è quasi sempre perfetta per i feed, in quanto questi file tendono ad essere oggetto di moltissime richieste pur non avendo sostanzialmente nessuna interazione con l'utente.

Tuttavia, dopo aver salvato una pagina, è necessario far sì che la cache venga in qualche modo invalidata quando i dati non sono più aggiornati. Rails fa si che sia possibile raccogliere queste funzionalità di pulitura della cache in apposite classi, gli *sweeper*. Continuando l'esempio, sarà dunque presente una classe FeedSweeper in app/models/feed_sweeper.rb, fatta in questo modo:

```
class FeedSweeper < ActionController::Caching::Sweeper
  observe Message

  def after_save(message)
    expire_page(:controller=>'feed', :action=>'rss')
  end
end
```

Il metodo observe serve a dire allo sweeper che deve attivarsi quando ci sono interventi su determinati oggetti. Si possono attivare diversi metodi di callback, che cioè verranno richiamati quando succede qualcosa relativo ai vari momenti della vita di un oggetto, come la creazione, la distruzione o la scrittura nel database.

Il metodo expire_page serve a cancellare uno dei file della cache, e prende in input il solito insieme di argomenti per costruire un url. Ma uno sweeper deve essere associato a determinate azioni, in modo che possa intervenire solo quando vengono effettuate determinate azioni, quindi bisogna usare la direttiva cache_sweeper in un controller, come in questo caso:

```
class FeedController < ApplicationController
cache_sweeper :feed_sweeper, :only=>[:add_message, :add_topic]
```

Qui si trovano gli argomenti :only ed :except, che permettono appunto di associare le operazioni dello sweeper a determinate azioni. Non sarebbe utile effettuare un controllo per azioni come index, o show, dove gli oggetti non possono essere creati. Per quel che riguarda add_topic, invece, lo sweeper verrà attivato, ma non sarà eseguita nessuna callback se l'oggetto non viene creato, il che farà si che non venga invalidata la cache esattamente come si vuole.

8.2 Caching delle azioni

L'action caching è sostanzialmente identico al page caching e le richieste vengono ancora fatte passare attraverso il controller, il che fa si che ad esse possano essere applicati filtri ed altro. Il meccanismo d'uso è pero praticamente immutato, semplicemente si userà caches_action nei controller, e negli sweeper si potrà usare expire_action. Il controller HomeController di una ipotetica home page, ad esempio, può far uso di un comodo caching di questo tipo:

```
caches_action :index, :add_topic, :show
```

Ovviamente, per gestire l'invalidazione della cache si dovrà far uso di altre regole di sweeping, creando una classe TopicSweeper ex novo o aggiungendo delle regole allo sweeper esistente, e facendo si che sia controllata anche la classe Topic. Se si sceglie di gestire diverse classi in un unico sweeper, bigognerà stare attenti alla gestione delle callback, in quanto l'oggetto potrà essere di qualsiasi tipo, e quindi sarebbe necessaria una soluzione come questa:

```
def after_save(thing)
  case thing
    when Topic
      expire_action :controller=>'home',:action=>'index'
    when Message
      expire_action :controller=>'home',
        :action=>'show', :id=>thing.topic.id
  end
end
```

Il codice dovrebbe essere autoesplicativo, ma si può notare una cosa interessante nella gestione relativa al caso in cui venga creato un nuovo messaggio: poiché il caching è collegato ad una azione con un parametro specifico, l'identificativo del

topic, si utilizza l'informazione salvata nel messaggio per costruire un url specifico, in modo da invalidare la cache soltanto per quell'oggetto.

8.3 Caching dei frammenti

L'ultimo tipo di caching è il caching dei fragment, frammenti, ed è quello pensato per gestire pezzetti statici all'interno di pagine dinamiche; sostanzialmente è necessario soltanto usare il metodo cache, ed un blocco.

```
<%= codice variabile %>

<% cache(:action=>"azione", :parametro=>"qualcosa") do %>
    <p> codice statico </p>
<% end %>
```

Per cancellare i dati si possono utilizzare expire_fragment(:controller=>mycontroller, :action=>azione, :parametro=>qualcosa), se non vengono specificati argomenti per il metodo cache il frammento verrà associato univocamente alla coppia controller-azione. Il fragment caching permette di raggiungere una flessibilità estrema, ed in effetti l'action caching è implementato usando i frammenti, ma ovviamente è quello che offre il minore incremento prestazionale, quindi andrà sfruttato solo quando gli altri metodi non sono possibili.

9 Altro da sapere su Rails

Ovviamente Rails non è solo ciò che che è stato visto, ma la sua completa e approfondita trattazione esula dagli scopi di questa tesi. Tuttavia in questo capitolo verrà presentata una breve overview di altre interessanti caratteristiche.

Anzitutto è presente un'ottima integrazione della tecnologia AJAX, ovvero la possibilità di aggiornare una pagina o parti di essa utilizzando JavaScript senza doverla caricare interamente da zero.

Utilizzando Rails è possibile realizzare web service tramite le tecnologie SOAP o XML-RPC in modo molto semplice, sfruttando i controller ed i modelli definiti precedentemente e permettendo uno sviluppo estremamente rapido.

ActionMailer è invece un modulo pensato per tutte quelle situazioni in cui è necessario inviare email, come può essere la verifica della validità di un account in fase di registrazione. Così le email sono usate creando mailers con il comando rails generate mailer nome_mailer. Le classi generate sono sottoclassi di ActionMailer::Base e vivono nella cartella app/mailers.

Rails offre degli helper appositi che permettono di applicare effetti dinamici e trasformazioni assortite, di creare caselle di testo con ricerca live, editor in place e molto altro, utilizzando una sola riga di codice. Inoltre un particolare tipo di viste, le viste .rjs sono state appositamente pensate per questo scopo, e grazie ad esse è possibile realizzare trasformazioni anche molto complesse utilizzando solamente ruby e lasciando che il codice javascript sia generato automaticamente.

Un altro aspetto interessante è che si possono creare applicazioni che non girino necessariamente sul browser, magari utili a caricare e sincronizzare periodicamente in background un database, oppure ad accedere direttamente ai dati di un'altra applicazione. Per realizzare tutto ciò si può comunque utilizzare il modulo Active Record e anche uno strumento più potente: *Active Support*. Quest'ultimo è un insieme di librerie condivise da tutti i componenti Rails che estendono le funzionalità delle classi con nuovi metodi e moduli.

Rails offre due differenti funzionalità per il riuso di codice tra applicazioni differenti. Il primo sono i componenti, ovvero interi pacchetti di funzionalità che possono essere trasportati da un'applicazione all'altra semplicemente copiandoli in una directory. Il secondo è il meccanismo dei plugin, che permette di andare a modificare funzionalità anche basilari del framework permettendo un'integrazione perfetta di nuovi strumenti nella base esistente. Esistono dozzine di plugin per gli usi più differenti, dai pacchetti per aggiungere un wiki all'applicazione alla

creazione di mappe con google maps, dall'indicizzazione full text del database alla realizzazione di wizard. La disponibilità di una comunità molto estesa e vitale che rilascia questi plugin liberamente garantisce la possibilità di includere funzionalità complesse in pochi attimi, ed è un vantaggio immenso per lo sviluppo di applicazioni in tempo molto ridotto.

Concludendo, ogni sviluppatore può entrare a far parte del continuo sviluppo di RoR sfruttando il repository GitHub per segnalare e aiutare a risolver bug e problemi, offrendosi disponibili ad effettuare test sul codice oppure a partecipare per ampliare la documentazione.

10 LMS Canvas

Un esempio concreto che sta avendo un discreto successo nel mondo dell'e-elearning è il Learning Management System **Canvas**, una piattaforma open-source rilasciata sotto licenza AGPLv3 da Instructure Inc. nel 2011.

Cos'è un LMS?

Un *Learning Management System* (LMS) è la piattaforma applicativa (o insieme di programmi) che permette l'erogazione di corsi in modalità e-learning al fine di contribuire a realizzare le finalità previste dal progetto educativo dell'istituzione proponente. Il learning management system presidia la distribuzione dei corsi on-line, l'iscrizione degli studenti, il tracciamento delle attività on-line. Gli LMS spesso operano in associazione con gli LCMS (learning content management system) che gestiscono direttamente i contenuti, mentre all'LMS resta la gestione degli utenti e l'analisi delle statistiche.

10.1 Keywords del progetto

Canvas è scritto in Ruby on Rails ed il codice è scaricabile da GitHub. Tuttavia non è nessario istallare il sistema poichè esiste una versione più leggera nella Canvas Cloud, dove basta registrare un account e utilizzare un'istanza della piattaforma.

I progettisti di questo LMS hanno individuato cinque punti cardine sui quali si sono ispirati.

Canvas è un Hub di comunicazione.

L'apprendimento significativo può verificarsi nelle conversazioni che abbiamo con gli altri, sia formali che informali, e Canvas è progettata per facilitare comunicazioni studente-studente e studente-istruttore. Essa dovrebbe essere una funzione automatizzata dei LMS, ed è proprio ciò che fa Canvas eliminando la necessità di e-mail manuali a singoli studenti per i vari avvisi. Quando viene apportata una modifica all'interno di un corso, le notifiche vengono automaticamente attivate e inviate agli utenti appropriati. Una volta che i messaggi sono inviati, gli studenti e gli insegnanti possono riceverli nel modo che preferiscono. Ci sono infatti molteplici canali di comunicazione, compresi SMS, notifiche di Facebook ed e-mail e la frequenza di tali notifiche è impostabile scegliendo fra 'subito', 'settimanale', 'tutti i giorni,' o 'mai'.

Canvas integra contenuti multimediali.

Canvas è progettato per rendere facile l'integrazione di contenuti multimediali da Internet, il più grande repository nel mondo. I corsi stessi sono disegnati per essere parte del web: con pochi click è possibile rendere pubblico il corso e quindi consultabile da molte persone. Ogni cambiamento effettuato verrà prontamente riflettuto online. Gli *ospiti* avranno accesso in sola lettura alle informazioni da qualsiasi parte del mondo.

Il *Rich Content Editor* in Canvas rende molto facile inserire testo, immagini, equazioni matematiche, o video tutti insieme in una sola pagina. Sono incorporabili ad esempio video di Youtube incollando semplicemente il link, oppure collegare presentazioni PowerPoint o documenti di Word, per i quali Canvas interagisce con Scribd per creare un'anteprima si possa navigare senza doverla scaricare sul proprio computer. Questo tool rende anche più facile registrare e inserire commenti audio o video quasi ovunque in un corso; si possono lasciare registrazioni audio all'interno di Conversations, lo strumento di messaggistica. La codifica di questi file media (e moltissimi altri non citati) è realizzata in modo tale da permettere ai lettori di accedervi senza problemi da qualsiasi dispositivo informatico.

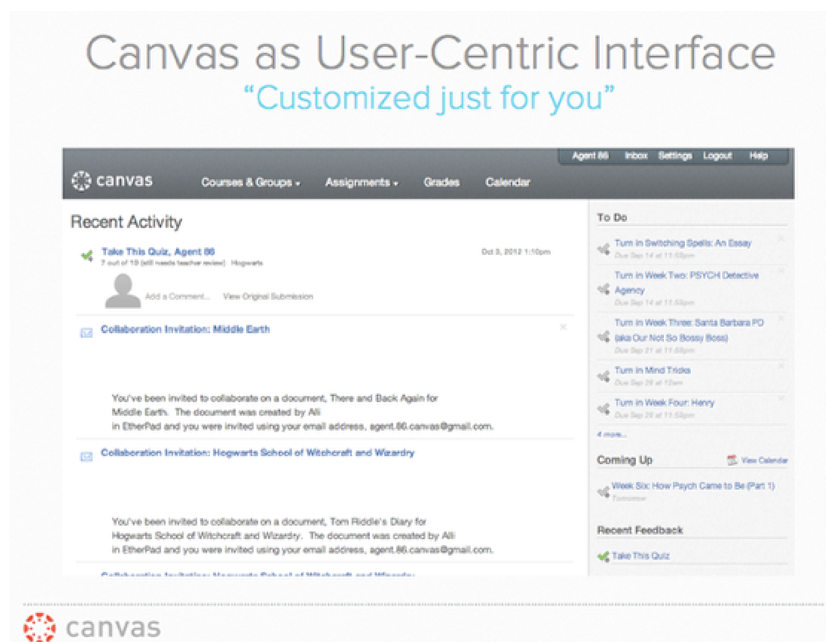
L'atomo costituente Canvas è l'HTML e ciò significa che si possano creare link ad altre pagine o risorse all'interno del corso tessendo una ragnatela. Inoltre il Rich Content Editor incoraggia l'utente a costruire pagine dai contenuti accessibili (ad esempio la formattazione del testo viene eseguita utilizzando gli stili) e di grafica semplice e intuitiva. Canvas è conforme con il W3C's Web Accessibility Initiative Web Content Accessibility Guidelines (WAI WCAG) 2.0.

User Interface di Canvas.

I fondatori hanno fatto una scelta molto consapevole per garantire che l'interfaccia utente sembrasse la stessa per tutti: studenti, insegnanti e amministratori di sistema. I dati visualizzati all'interno di tale interfaccia cambiano a seconda dell'utente, ma l'interfaccia di base no. Questo aspetto di Canvas rende molto facile la formazione di nuovi utilizzatori, perché si può fare riferimento alla stessa interfaccia utente. Un corso Canvas può essere suddiviso in sei parti:

- il Global Navigation menu situato in alto, che mostra tutti i corsi e i gruppi di appartenenza, le prove di verifica da sostenere, i voti e il calendario per tutte le classi;
- il Course Navigation menu sulla sinistra, che collega l'utente alla varie aree del corso;

- il Breadcrumb Navigation menu all'inizio del corpo della pagina, che consente di spostarsi verso l'alto e verso il basso nella gerarchia corso;
- la Sidebar sulla destra, che ha tutti i link e i pulsanti necessari per ottenere gestire il corpo della pagina principale;
- l'Help Corner in alto a destra, dove si può ottenere supporto, controllare la casella di posta o modificare il proprio profilo.



Naturalmente esistono anche delle features avanzate che soddisfano esigenze più particolari, tuttavia ne è sconsigliato l'utilizzo ai principianti; infatti Canvas è pensato esso stesso come un corso di apprendimento nel quale un passo alla volta si arriva alla piena conoscenza del sistema.

Un corso, molti punti di vista.

Il contenuto costruito all'interno del corso è lo stesso, ma ci sono molti modi diversi di vedere tali dati. È possibile visualizzarli attraverso il calendario, il programma, la pagina dei materiali didattici o il registro di classe. Si tratta sempre degli stessi dati, solo visti in modo diverso.

Ci sono quattro schermate nell'immagine qui sotto: il Calendario [1], il Syllabus [2], la pagina dei materiali didattici [3], e il registro di classe [4].

- Il calendario è una lente che guarda al corso e mostra solo le cose che hanno una data ad esse associate.
- Il Syllabus è una lente che illustra all'utente quali sono gli obiettivi del corso e come raggiungerli.
- La pagina dei materiali didattici è una lente che visualizza i contenuti forniti da docente all'interno del corso.
- Il Libretto è una lente che mostra solo le sezioni di valutazione, con i risultati dei test e il relativo peso assegnato.

Canvas semplifica l'insegnamento

Canvas rende estremamente facile per l'utente includere risorse per assemblare i corsi, se tali risorse provengono da un corso precedentemente presentato, un corso che qualcun altro ha tenuto in Canvas, o proveniente da altri LMS.

Quando arriva il momento di costruire un nuovo corso o copiare i materiali didattici da un semestre all'altro, si può infatti risparmiare un sacco di tempo selezionando solo il contenuto necessario e far partire da lì la costruzione. In questo modo, ogni corso diventa un template da usare o copiare per intero o in parte.

Sostanzialmente Canvas offre tutte le funzionalità degli altri lms, quindi perché dovrebbe riuscire competere con software largamente utilizzato e testato e con un insieme vasto di plugin, ad esempio come moodle?

La risposta rientra nella prospettiva diversa in cui è stato sviluppato. Il core di canvas è implementato per favorire il social learning e l'integrazione con i social network (Facebook, Google Docs). Inoltre Canvas dà molto peso all'accessibilità dei contenuti e alla loro fruibilità, favorendo la user experience. L'aspetto mobile e tablet è molto ben curato e permette l'utilizzo dell'lms in tutte le sue feature. Questo ha un'importanza enorme ora che la connettività non è prerogativa solo dei pc, ma di una moltitudine di dispositivi eterogenei.

Inoltre è stata dedicata particolare attenzione alla keyword che sta rivoluzionando i servizi informatici, ovvero il cloud computing, dato che canvas è sviluppato per sfruttare a pieno queste potenzialità.

Nonostante le indubbie qualità, Canvas dovrà affrontare una sfida molto complicata: quella di creare attorno a se una community attiva e numerosa di developer che continuino a far progredire il core e sviluppino numerose componenti aggiuntive e plugin. Anche se questo non è facilitato dalla mancanza di una developer guide adeguata che, nonostante il sorgente sia su github, scoraggia i programmatori ad avvicinarsi al prodotto.

Conclusioni

Questa tesi non aveva la pretesa di essere una guida per Ruby on Rails, e non lo è stata. Il lavoro svolto però è servito a tracciare una sorta di percorso alla conoscenza del framework, con l'intento di darne un'idea sui punti forti e su cosa si può o non si può fare. Gli argomenti presentati dovrebbero far capire ad un programmatore interessato a sviluppare in Rails se questo strumento fa al caso suo. Di conseguenza per ottenere una applicazione che faccia qualcosa di discreto bisognerà sicuramente approfondire la conoscenza utilizzando le apposite guide forum e quant'altro, ma soprattutto bisognerà fare esperienza attraverso molte prove ed esperimenti.

D'altronde non bisogna cadere nell'errore di pensare che Rails sia un ambiente perfetto. La community di sviluppatori che ci sta intorno sta crescendo di giorno in giorno e il framework ha un riscontro positivo nel fatto che stanno spuntando nuove applicazioni.

Per quanto riguarda Canvas, le considerazioni sono state fatte nel relativo capitolo.

Riferimenti bibliografici e sitografici

- Libro: Sam Ruby, Dave Thomas, David Heinemeier Hansson, Agile Web Development with Rails, Fourth Edition, The Pragmatics Bookshelf.
- Sito web: Ruby on Rails Guides (v3.2.11) http://guides.rubyonrails.org.
- Sito web: Guida Ruby On Rails http://www.html.it/guide/guida-ruby-on-rails.
- Sito web: Guida Ruby On Rails 2 http://www.html.it/guide/guida-ruby-on-rails-2.
- Sito web: Guida Ruby http://www.html.it/guide/guida-ruby.
- Sito web: Help and documentation for the Ruby programming language http://ruby-doc.org.
- Sito web: Canvas by Instructure http://www.instructure.com/
- Sito web: Canvas Guides http://guides.instructure.com/