Introduzione a Rails

Rails è un framework per lo sviluppo di applicazioni web basato sul pattern ***MVC***. Rails favorisce la metodologia di sviluppo detta **convention over configuration**, in altri termini il framework fa delle assunzioni su cosa vogliamo sviluppare e su come lo vogliamo sviluppare, piuttosto che costringerci a definire innumerevoli file di configurazione per descrivere il comportamento di ogni singolo meccanismo del framework.

Rails ha inoltre una visione del pattern MVC leggermente differente rispetto altri frameworks come ad esempio Struts. In particolare per quanto rigurda il MODEL lo strato del CONTROLLER. Infatti nel MODEL troveremo nella stragrande maggioranza dei casi delle classi ActiveRecord, invece per quanto riguarda lo strato del CONTROLLER, pur rimanendo un layer di connessione tra MODEL e VIEW, ospiterà gran parte della business-logic che vorremmo implementare nell'applicazione.

Componenti fondamentali

Rails viene distribuito in una librebria gem , che è costituita da questi componenti fondamentali:

1)ActionPack
   (a) ActionController
   (b) ActionDispatch
   (d) ActionView

2)ActionMailer

3)ActiveModel

4)ActiveRecord

5)ActiveResource

6)ActiveSupport

In questa piccola guida cercherò di far capire il ruolo di questi componenti, sviluppando una serie di esempi, dedotti durante lo sviluppo di un'applicazione di social-networking (twitter-clone) seguendo una guida all'URL : http://www.railstutorial.org

MODEL

Nello strato del MODEL, vanno a finire tutte le classi del nostro dominio di business, le cosidette classi entity, mentre il più delle volte la business-logic viene eseguita nelle azioni del controllore. Nessuno comunque ci impedisce di creare un layer business-logic per i casi d'uso più complessi che coinvolgono diverse classi entity.

In ogni caso lo strato del MODEL è fortemente influenzato dal ***pattern Active Record*** e dalle ***Migrations.*** Rails favorisce il progetto del dominio di business e del modello dei dati come un processo unico, supportato dalle Migrations, che ci permettono di definire lo schema dei dati senza conoscere DDL-SQL, e di gestire l'evoluzione dello schema senza ricorrere a istruzioni SQL, ma solo gestendo degli script Ruby.

Fowler da questa definizione di **Active Record**: *"An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on the data"*.
Questo è esattamente quello che fa l'implementazione Ruby di ActiveRecord.

Le classi che modelliamo come ActiveRecord sono responsabili delle operazioni CRUD dei propri dati, e anche di eseguire la logica di business richiesta su questi dati, come ad esempio la ***VALIDAZIONE*** degli stessi. Ma andiamo a creare la classe che modella la risorsa principale della nostra applicazione che è User:

rails generate model User name:string email:string

questa riga produrrà oltre che la nostra classe User con i due attributi String, anche una ***Migration*** che è il seguente script Ruby:

```
class CreateUsers < ActiveRecord::Migration

  def self.up
    create_table :users do |t|
      t.string :name
      t.string :email
     t.timestamps
    end
  end

  def self.down
    drop_table :users
  end
end
```

Analizzando lo script, ci accorgiamo che si tratta di una classe CreateUsers che estende da ***ActiveRecord::Migration.*** L'istanza di questa classe presenta un paio di metodi di classe **up()** e **down()** che verranno utilizzati per creare e distruggere la tabella users relativa a User.

In particolare per creare la tabella si utilizza il metodo di ActiveRecord ***create_table()*** e per eliminarla si utilizzerà ***drop_table().*** Si noti poi anche, che up() creerà anche le colonne **name, email, created_at** e **updated_at.** Non dobbiamo istanziare la classe direttamente, ma utilizziamo rake (un duale di make però per ruby)

***rake* db:migrate**

mentre per distruggere la tabella:

rake db:rollback

supponiamo ora ad esempio di voler aggiungere un vincolo d'univocità alla colonna email e magari anche un indice; si tratta di evolvere lo schema dei dati, e questo viene gestito con una migrazione. In altri termini andiamo a creare una nuova migrazione:

rails generate migration add_email_unique_index

questo genera la migrazione : ***<time_stamp>_add_email_unique_index.rb*** in cui aggiungiamo il codice necessario per la nostra migrazione:

```
  def self.up

    add_index :users, :email, :unique=>true

  end
```

```
def self.down
  remove_index :email
end
```

Per implementare un'autenticazione (vedi dopo) abbiamo bisogno di una colonna encrypted_password che manterrà la versione criptata della password, nel database. Questo rappresenta un nuovo requisito, e per gestirlo ancora una migrazione :

rails generate migration add_column_encrypted_to_users encrypted_password:string

rake db:migrate

A questo punto andiamo a vedere come è fatta la classe User.

Classi Del Model E Validazioni

La classe User contenuta in ***app/models/user.rb*** è :

```
class User < ActiveRecord::Base


end
```

Da ActiveRecord::Base ereditiamo un costruttore, che durante la creazione dell'istanza, come prima cosa va a controllare se esiste la tabella per questa classe e poi va a vedere quali colonne sono definite per essa, e da questa lettura crea gli attributi e i rispettivi metodi accessors per la classe.

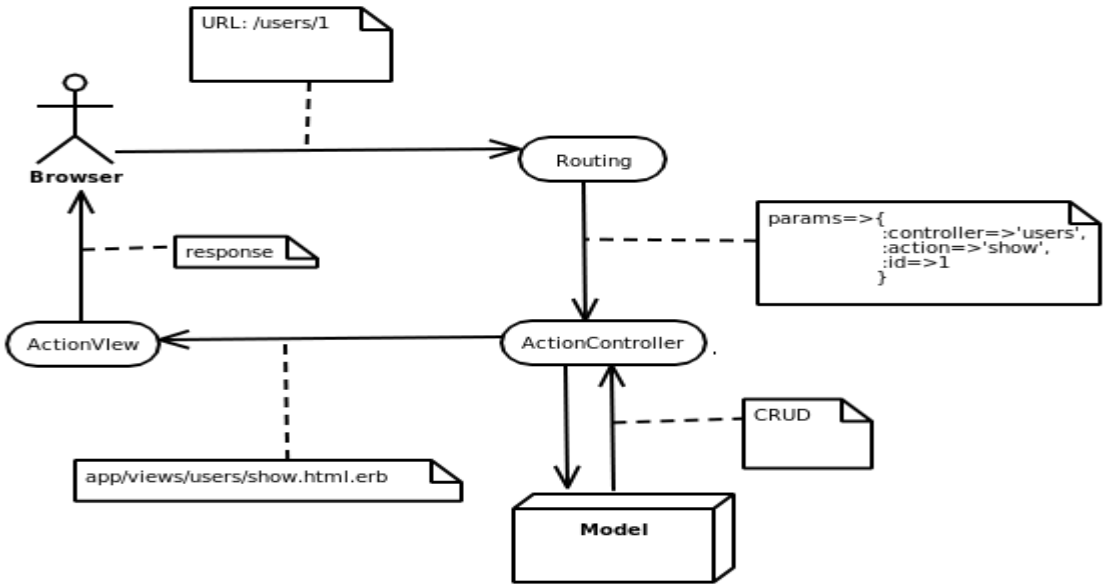
In questo caso ***:name***, ***:email*** e :***encrypted_password***. Prima di Rails 3 le validazioni venivano eseguite tramite metodi di ActiveRecord. Ora questi metodi e altri che hanno lo scopo di aumentare le capacità delle classi del MODEL, sono stai inglobati in un'altro modulo che è ***ActiveModel*** che a sua volta è incluso in ActiveRecord.

```
class User < ActiveRecord::Base

  regex=/\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i

  attr_accessor :password #password_confirmation inserita in automatico
  attr_accessible :name, :email, :password,:password_confirmation

 validates :name,
        :presence=>true,
        :length=>{:within=>3..40}

 validates :email,
        :presence=>true,
        :uniqueness=>{:case_sensitive=>false},
        :format    => { :with => regex }

 validates :password,
        :presence=>true,
        :confirmation=>true,
        :length=>{:within=>6..40}

 validates :password_confirmation,
        :presence=>true
end
```

Il codice qui sopra parla da se. Come vediamo ereditando da ActiveRecord, gli oggetti della nostra classe User hanno già innata le capacità di validazione più comuni. Possiamo controllare la validità di un oggetto utilizzando il metodo ***is_valid?().*** Ovviamente gli oggetti per essere propagati dovranno essere validi.

VIEW & CONTROLLER

Il ciclo di richiesta di Action Pack



Nella figura qui sopra, si sta Supponendo che, da un browser, arrivi la richiesta rappresentata dall'URL /users/1 alla nostra applicazione Rails. Ricevuta la richiesta, Rails istanzia il componente ***ActiveDispatch*** che ha il compito di analizzare l'URL e ricavare il controllore e l'azione da invocare. *ActiveDispatch* può fare ciò basandosi sul file **config/routes.rb** che analizzeremo a breve. Inoltre *ActiveDispatch* prepara una variabile globale, un'istanza di *Hash*, chiamata ***params***, dove le azioni del controller possono eventualmente recuperare tutti i parametri passati dalla richiesta, e dove sono inserite anche delle informazioni di routing sull'azione attivata; in questo modo Rails implementa in maniera semplice il ***meccanismo di trasferimento dei dati***.

Continuando ad analizzare la figura, vediamo che l'esecuzione va avanti con l'invocazione dell'azione corretta per la richiesta. Le azioni interagiscono con gli oggetti dello strato del MODEL, dove nella maggior parte dei casi di applicazioni Rails, troveremo istanze di *ActiveRecord.* Eseguita la logica di business, se non specificato diversamente dall'azione, Rails istanzierà un oggetto di tipo *ActionView* che ha il compito di generare la presentazione per l'utente.

Routing , REST , ActionController e ActionView : una visione d'insieme.

Rails ha un supporto naturale per l'implementazione di web services basati su REST. Abilitare questo supporto è immediato. Ad esempio nella nostra applicazione abbiamo individuato una risorsa che è un utente rappresentato dalla classe User. Allora nel file **config/routes.rb** andiamo ad inserire una linea:

resources :users

e basterà l'inserimento di questa linea, per abilitare le sette operazioni REST fondamentali che possiamo eseguire sulla risorsa, nonché a portare alla creazione di altrettanti metodi helpers, per generare gli opportuni path e url che utilizzeremo nella view:

HTTP VERB	**URL**	**CONTROLLER#ACTION**	**HELPER**	**PURPOSE**
GET	/users/	users#index	users_path	Visualizza tutti gli utenti
GET	/users/id	users#show	user_path(id)	Visualizza l'utente con l'id specificato, il parametro id è passato via params[:id]
GET	/users/new	users#new	new_user_path	Richiede di voler creare un nuovo utente.
POST	/users	users#create	users_path	Crea l'utente fisicamente. I dati che arrivano dal client per popolare la risorsa sono piazzati in params
GET	/ users/id/edit	users#edit	edit_user_path(id)	Richiede il meccanismo di editing dell'utente con l'id specificato. Il parametro id è inserito in params
PUT	/users/id	users#update	user_path(id)	Aggiorna l'utente con questo id, secondo i dati passati dal client. Sia i dati che l'id sono inseriti in params
DELETE	/users/id	users#destroy	user_path(id)	Cancella l'utente con questo id

Dalla tabella si vede come le decisioni riguardo al routing, vengono prese in base al verbo HTTP utilizzato e all'URL richiesto; ma vediamo come questo meccanismo di routing è collegato con le azioni del controllore con un esempio pratico. Andiamo a creare quindi il controllore adatto alla

risorsa User:

rails generate controller users index show new create edit update destroy

andrà a generare seguendo le convenzioni sulla pluralizzazione dei nomi la seguente classe :

```
class UsersController < ApplicationController
  def index
  end
  def new
  end
  def create
  end
  def edit
  end
  def update
  end
  def destroy
  end
 def show
 end
end
```

come vediamo una classe controllore deve estendere sempre la classe ***ApplicationController,*** e andando ad analizzare ***app/controllers/application_controller.rb*** troviamo:

```
class ApplicationController < ActionController::Base
end
```

Quindi un controllore sarà un'istanza di ***ActionController::Base***. Questa è una classe che contiene tutti metodi che sono utilizzabili dentro un controllore, e che quindi stabilisce tutto quello che un controllore può o non può fare in Rails. Andiamo di seguito ad analizzare e implementare le varie azioni RESTful che possiamo eseguire sulle risorse.

users#index

L'azione users#index ha lo scopo di visualizzare tutti gli utenti della nostra applicazione, e spesso quando si deve visualizzare un elenco piuttosto consistente di elementi, si desidera implementare la paginazione, per questo motivo abbiamo installato una libreria di supporto di nome *will_paginate*. (vedi bundle) Con questo gem installato nel bundle, ci ritroviamo dentro la classe *User* e dentro la classe *Array*, il metodo *paginate()* che lavora come *User.all* ma invece di ritornare un *Array* con tutti gli elementi della tabella users, ritorna una struttura dati *WillPaginate::Collection* adeguata per essere utilizzata durante la paginazione eseguita dall' helper *will_paginate*.

```
def index
    @users = User.paginate(:page=>params[:page])
    @title=visualizza tutti gli utenti
```

```
end
```

Notiamo come il controllore abbia il compito di trasportare i dati del dominio di business che dobbiamo manipolare. In quest'azione infatti stiamo creando un paio di variabili d'istanza del controllore, e in particolare ***@users*** contiene un insieme di oggetti User recuperati dalla omonima tabella. Queste variabili d'istanza saranno automaticamente disponibili all'oggetto ***ActionView*** istanziato da Rails per presentare lo stato dell'applicazione dopo l'esecuzione dell'azione. In questo modo si realizza il trasporto dei dati dal dominio di business alla *VIEW*.

Una volta eseguita l'azione se non si specifica altrimenti, Rails seguendo una convenzione, va a cercare un template del tipo **nome_azione.html.erb** e in questo caso **index.html.erb.** Il rendering effettivo è eseguito da Rails creando l'istanza di una sottoclasse di ***ActionView::TemplateHandlers***, a seconda dell'estensione del file che determina il template da utilizzare. Da Rails 2 in poi, le estensioni sono:

1) **.erb** (HTML con embedded Ruby
2) **.rjs** (javascript con embedded Ruby)
3) **.builder** (per i generatori XML)

In questo caso andiamo a vedere come potrebbe essere il template index.html.erb :

```
[...]
<%=will_paginate @users%>
<ul class=”users”>
  <%@users.each do |user| %>
  <li>
     <%=gravatar_for user, :size=>40 %>
     <%=link_to user.name, user_path(user.id)%>
  </li>

</ul>
<%=will_paginate @users%>
```

Si vede che si tratta di *HTM*L, con dentro codice Ruby. In effetti per integrare Ruby nel documento HTML si utilizzano i tag <% ..%> e <%= ..%>. La cosa più importante da notare è che Rails, istanziato l'oggetto ***ActionView::TemplateHandler*** adeguato al parsing di questo tipo di file, andrà a eseguire il codice Ruby tra i tag e a sostituire al tag stesso <%= … %> il risultato dell'elaborazione nel punto esatto dove si trova il tag. Si noti inoltre come *@users* provenga da *ActionController* e anche qui, vi possiamo accedere come variabili di istanza di *ActionView*. In particolare stiamo iterando sugli elementi della collezione @users, estraendo i signoli elementi e inserendo informazioni in una lista <ul>; in ogni <li> inseriamo un'immagine (tramite l' helper gravatar_for che sfrutta la gem gravatar che fornisce la gestione delle immagine di profilo) e un link:

```
<%=link_to user.name, user_path(user.id)%>
```

Questo metodo ***link_to*** proviene da uno dei moduli inclusi in *ActionView*, e ci serve a generare i links in maniera consistente nell'utilizzo con Rails. In questo caso il primo argomento è il testo da visualizzare, e ci mettiamo user.name, mentre il secondo argomento è il link vero e proprio, cioè l'URL che generiamo con uno dei metodi helper iniettati da *ActionDispatch* durante il routing dell'azione. Infatti come detto sopra, nel routing sono stati abilitati :

1) ***GET /users/id*** <=> azione show()
2) ***PUT /users/id*** <=> azione update()

id è l'id dell'utente che individua in maniera univoca la risorsa; utilizzando ***user_path(user.id)*** andiamo a generare proprio i vari URLs ***/users/1, /users/2, /users/3*** e così via per tutti gli utenti presenti nella lista. Quindi in questo modo ogni utente della lista è collegato ad un'azione show()

poiché se non specificato altrimenti, i link utilizzano il verbo HTTP GET. (si inizi a notare che il browser supporta solo i verbi GET e POST in maniera nativa, ma come vedremo dopo, Rails raggira questo deficit anche grazie a link_to) Il ruolo della linea

<%=will_paginate @users %>

è quello di realizzare la paginazione e visualizzarla in maniera opportuna. Si noti infine come quest'azione sia accesa sempre da una GET HTTP, e sia spesso collegata con una READ di tutti i record nella tabella, o di un sottoinsieme di essi.

ActionController: Rappresentazioni multiple delle risorse e loro negoziazione

Nel caso appena esaminato riguardo l'azione *index()* abbiamo visto come Rails può fornire una certa rappresentazione di una risorsa. In altri termini un client, invocando ***GET /users*** va a richiedere la rappresentazione di una risorsa del sistema che è una lista di utenti, e in questi termini possiamo anche pensare al controllore *users* come ad una risorsa del sistema.

Ma **Rails supporta e può servire diverse rappresentazioni per la medesima risorsa.** Ad esempio un utente davanti ad un browser che richieda ***GET /users/*** vuole probabilmente visualizzare tutti gli utenti del sistema in pagine html come abbiamo mostrato. Però un client della nostra applicazione *RESTful* può essere ad esempio un feed-reader che si aspetta una diversa rappresentazione della risorsa come ad esempio un documento *RSS*, oppure potrebbe essere un'applicazione *AJAX* che sta richiedendo un oggetto *JSON* come rappresentazione della lista di utenti.

Quando lasciamo decidere a Rails quale formato (rappresentazione) della view si deve presentare dopo un azione, come ad esempio index(), Rails segue come al solito una convenzione, va a vedere dentro la directory /app/views/users/ se trova un file del tipo :

nome_azione.formato.template_engine

quindi se costruiamo index.html.erb potrà servire una rappresentazione HTML, se inseriamo i***ndex.xml.builder***, potrà servire documenti *XML*, se invece costruiamo ***index.atom.builder*** oppure ***index.rss.builder*** potrà servire richieste *Atom* o *RSS e così via.*

In generale quindi possiamo creare e mantenere le nostre diverse rappresentazioni in questa maniera. Però possiamo anche sfruttare le potenzialità di un oggetto del MODEL che estendi *ActiveRecord(ActiveModel)*, e generare delle rappresentazioni *XML* oppure *JSON* al run-time direttamente da metodi dell'oggetto stesso. Andiamo allora ad aggiungere funzionalità a index() in modo che possa supportare *JSON*, *XML* e anche una nostra view in formato *RSS*:

```
def index
    respond_to do |format|

        format.html{
            @users = User.paginate(:page=>params[:page])
            @title = "Visualizza tutti gli utenti"
        }

        format.atom{

       }

        format.rss{
```

```
        }

        format.xml{
          users=User.all
          render :xml=>users
          return
        }


        format.json{
            users=User.all
            render :json=>users
            return
         }
end
```

Dal codice qui sopra si evince che possiamo decidere quale formato presentare con il metodo di ActionController ***respond_to(),*** e sfruttando le capacità di ActiveRecord (ActiveModel), che aggiunge dei metodi di supporto per varie rappresentazioni come ad esempio il metodo ***to_xml()***, per produrre rappresentazioni *XML* e con ***to_json()*** per *JSON*.

Il metodo ***render()*** è un metodo disponibile ai controllori, con cui possiamo decidere quale view presentare al client. In questo caso gli abbiamo passato :xml=>users il che equivale a dire ***render users.to_xml().*** Per supportare invece *RSS* abbiamo scelto di costruire e mantenere una rappresentazione aggiungendo rispettivamente ***index.rss.builder*** in /app/views/users/, che ha lo scopo di generare un canale *RSS* con un item per ogni utente iscritto alla nostra applicazione.

index.rss.builder

```
xml.instruct! :xml, :version => "1.0"
xml.rss :version => "2.0" do
   xml.channel do
      xml.title "Ruby On Rails Demo Application news"
      xml.description "Le news esportate da Ruby On Rails Demo Application"
      xml.link users_url(:rss)

   for user in @users
     xml.item do
        xml.title user.name
        xml.description "Anche l'utente #{user.name} si è iscritto a Ruby On Rails Demo Application"
        xml.pubDate user.created_at.to_s(:rfc822)
        xml.link user_url(user, :rss)
        xml.guid user_url(user, :rss)
     end
  end
 end
end
```

Quando Rails trova index.rss.builder instanzia una sottoclasse di ***ActionView::TemplateHandler*** adeguata al parsing del file. ***In particolare questo template-engine andrà ad istanziare a sua volta un oggetto con nome xml, di una sottoclasse di Builder::XmlMarkup*** adeguata a gestire il markup necessario per *RSS, e* Nel file index.rss.builder abbiamo subito a disposizione l'oggetto xml. Si noti che nella descrizione dell'URL del canale, e nella descrizione dell'URL dell'item *RSS* abbiamo utilizzato gli helepers generati dal routing di Rails. In particolare si noti che:

users_url() genera: http://localhost:3000/users
users_url(:rss) genera: http://localhost:3000/users.rss
user_url(user) genera: http://localhost:3000/users/:id
user_url(user,:rss) genera: http://localhost:3000/users/:id.rss

ma a cosa servono le estensioni? E qui introduciamo il discorso della negoziazione delle

rappresentazioni. In generale vi sono tre modi principali adottati nel tempo come best-practices convenzionali nella stragrande maggioranza dei casi, per negoziare la rappresentazione desiderata:

1)Utilizzo di un parametro nella query string come ad esempio il parametro ***format***
ad esempio GET /users?***format=rss*** oppure GET /users?***format=xml***

2)Utilizzo del ***suffisso d'estensione*** : GET /users***.rss*** oppure GET /users***.xml***

3)***HTTP Content Negotiation***

Rails supporta tutti e tre questi metodi e li prova nell'ordine suddetto. Per introdurre altri concetti, consideriamo adesso l'implementazione di altre azioni del controller.

users#new()

Questo metodo va sempre in coppia con il metodo *users#create()* nel senso che in generale ad ogni richiesta d'azione *new()* segue una richiesta create(). L'azione *new* richiede in effetti la rappresentazione di una nuova risorsa virtuale, cioè la rappresentazione di una risorsa che attende di essere creata fisicamente con create. Questo metodo è attivato sempre da una *GET* ed in particolare come visto nella tabella ***GET /users/new.*** La maggiorparte delle volte, questa richiesta spedirà una pagina *HTML* con la form necessaria a raccogliere i dati per creare la risorsa. Quindi possiamo affermare che la rappresentazione ritornata a questa richiesta è la form. Andiamo a vedere l'implementazione del metodo:

```
def new

   @title="Nuovo Utente"

   @user = User.new() # serve per l'helper form_for

end
```

e quindi il template ***app/views/users/new.html.erb***

```
<%=form_for @user do |f| %>

<div class="field">
     <%= f.label(:name, "Nome")%> <br />
     <%= f.text_field(:name)  %> <br />
</div>

 <div class="field">
     <%= f.label :email%> <br />
     <%= f.text_field :email  %> <br />
</div>

<div class="field">
     <%= f.label :password%> <br />
     <%= f.password_field :password  %> <br />
</div>

<div class="field">
     <%= f.label :password_confirmation, "Conferma Password"%> <br />
     <%= f.password_field :password_confirmation  %> <br />
</div>

<div class="actions">

<%= f.submit "Registrami!" %>

</div>

<%end%>
```

Nella applicazioni web è un task molto comune quello di creare oppure editare le risorse della nostra applicazione, e questo di solito viene eseguito tramite delle form. Rails possiede una vasta libreria di metodi helper per la costruzione delle form. In particolare in questo esempio stiamo utilizzando tre helpers molto potenti che in caso sia stato abilitato REST possono sfruttare il meccanismo del record identification per costruire la form, contribuendo a mantenere le nostre view concise e pulite. In particolare ponendo l'attenzione su :

```
<%=form_for @user do |f| %>

   <div class="field">
       <%= f.label(:name, "Nome")%> <br />
       <%= f.text_field(:name)  %> <br />
   </div>
```

notiamo che all'helper ***form_for*** viene passata l'istanza di User @user. In output il markup prodotto sarà:

```
<form accept-charset="UTF-8"
        action="/users"
        method="post">
    <div style="margin:0;padding:0;display:inline">

                <input name="utf8" type="hidden" value="&#x2713;" />

                 <input name="authenticity_token"
                        type="hidden
                        value="Penc/G2K0oZMcniZ6k1QzG2jGnzuKQbCj7nucIBewbA=" />
     </div>

    <div class="field">
        <div class="field"><label for="user_name">Nome</label></div> <br />
        <div class="field"><input id="user_name" name="user[name]" size="30" type="text" value="" /></div><br />
     </div>
```

Quindi come si vede, Rails è abbastanza intelligente da determinare che l'istanza è nuova (transient) e a generare l'opportuno URL ***action="/users"*** e anche con l'opportuno metodo HTTP ***method="post"*** e inoltre gli attributi name dei vari campi sono impostati in modo che Rails ne potrà ricavare un Hash user={:name=>...,:email=>... } da inserire in params, in modo che tale Hash sarà poi accessibile da params in questa maniera: params[:user]. Quando si preme il pulsante submit della form, viene dunque inviata una richiesta ***POST /users*** che accende la create() del controllore users.

users#create

```
  def create

    @user = User.new params[:user]

    if !@user.valid?
      @title="Registrazione | Errore"
      render 'new'
    else
      @user.save!()
      flash[:success]="Registrazione riuscita!
                  per convalidare la registrazione
                  segui il link che ti abbiamo
                  spedito a #{@user.email}"
      redirect_to root_path
    end
  end
```

Quest'azione del controllore ha dunque lo scopo di creare fisicamente la risorsa, il che, la maggior parte delle volte, significa associare la risorsa al meccanismo di persistenza e cioè propagare l'istanza in un record del database, via ***ActiveRecord::Base.save(),*** cioè quest'azione corrisponde ad una ***CREATE*** di ***CRUD***. Si noti poi che abbiamo utilizzato il metodo ***redirect_to***. Questo metodo

serve ai controllori per muoversi da un'azione ad un'altra. In questo caso una volta registrato l'utente, l'azione esegue **redirect_to /** per accendere l'azione ***home()*** del controllore ***pages*** che ha lo scopo di preparare e mostrare la pagina home; infatti nel file ***routes.rb*** abbiamo inserito la seguente linea:

:root :to=>'pages#home'

Si noti che redirect_to quando avvia una nuova azione, deve istanziare un nuovo controllore e invocarne il metodo adeguato e quindi non si trasferiscono le variabili d'istanza eventualmente create nel controllore di partenza. In questo caso se si desidera passare argomenti, bisogna sfruttare la sessione. (vedremo dopo). A questo punto ci potremmo chiedere: è possibile spedire una diversa rappresentazione dei dati, popolati dal client, alla nostra applicazione Rails? In altri termini posso richiedere ad esempio la registrazione di un nuovo utente non attraverso una form, ma da un'applicazione client remota che spedisce i dati relativi all'utente codificati ad esempio in file XML? Dal punto di vista di Rails non dobbiamo fare assolutamente nulla se creiamo il giusto file XML. Quando Rails riceve una richiesta XML, converte il documento XML in arrivo nell'opportuno Hash. Ad esempio quando spediamo un file XML di questo tipo:

```
<?xml version='1.0' standalone='yes'?>
<user>
  <name>Pinco Pallino</name>
  <email>pinco@pallino.com</email>
  <password>valid password</password>
  <password_confirmation>valid password</password_confirmation>
</user>
```

invocando ***POST /users.xml,*** se nel metodo create() è presente :

```
def create()

@user = User.new params[:user]

   format.xml{
          if @user.save
            render  :xml=>@user , :status=>:created, :location=>user_path(@user)
            else render :xml=>"<?xml version='1.0' standalone='yes'?>
                        <response>406 Not Acceptable</response>",:status=>:not_acceptable
          end
end
```

Allora come si vede da ***@user = User.new params[:user]*** non dobbiamo cambiare nulla, per ricevere i dati dal meccanismo di trasferimento, in quanto params[:user] conterrà l'Hash: **user**={**:name**=>'Pinco Pallino',**:email**=>'pinco@pallino.com',:password=> …}.

users#edit

Questo in generale, va considerato accoppiato con users#update, nel senso che ad ogni azione *users#edit()* segue solitamente *users#update()*. Quest'azione richiede di preparare una risorsa virtuale che rappresenti un aggiornamento. Questo porterà a distribuire una rappresentazione che solitamente sarà una form pre-popolata con i dati relativi alla risorsa da aggiornare. L'azione viene avviata da una *GET*, e in particolare:
GET /users/id/edit dove l'id è come al solito, l'id dell'istanza di User da aggiornare. In generale quest'azione è collegata all'azione di READ dal databse, per recuperare e popolare la risorsa da aggiornare.

```
 def edit

      @user  = User.find_by_id(params[:id])
      @title = "Impostazioni utente"

end
```

Come si vede, spesso l'azione di edit è associata ad una READ dal database; Per quanto riguarda invece app/views/users/edit.html.erb ha praticamente lo stesso contenuto di prima, con l'unica differenza che ***form_fo***r ha ora la capacità di capire che *@use*r è un'istanza persistente, e quindi il markup di output dovrà tenere conto che l'azione da richiamare alla submit della form è PUT /users/id; ecco un frammento di output:

```
<form accept-charset="UTF-8"
      action="/users/1"
      class="edit_user"
      id="edit_user_1"
      method="post">

 <div style="margin:0;padding:0;display:inline">
    <input name="utf8" type="hidden" value="&#x2713;" />
    <input name="_method" type="hidden" value="put" />
    <input name="authenticity_token"
           type="hidden" value="85UdB9Vu2ihma8f6X7TKt44quNKruugUksbVBxfDRx8=" />
 </div>
```

Per raggirare questa situazione, Rails utilizza un campo nascosto:
<input name="_method" type="hidden" value="put" />

Questo fa arrivare a Rails _method='put' che gli dice che ci stiamo riferendo ad un'azione PUT. A questo punto alla submit verrà richiesta l'azione users#update()

users#update

```
def update

    @user = User.find_by_id( params[:id] )

    if @user.update_attributes(params[:user])
      flash[:success]="Impostazioni utente aggiornate!"
      redirect_to user_path(@user)
    else
       @title="Impostazioni utente | Errore"
       render 'edit'
    end

 end
```

L'azione update che segue una di edit, ha dunque il compito id aggiornare la risorsa. Quest'azione è accesa da ***PUT /users/id*** ed è spesso collegata ad ***UPDATE*** (***CRUD***) nel database. Vediamo che nell'esempio si sta utilizzando il metodo di ***ActiveRecord::Base update_attributes()*** che come argomento riceve un Hash con gli attributi che devono essere aggiornati. Il metodo esegue l'aggiornamento degli attributi e quindi esegue una save() (ovviamente la save è abbastanza intelligente da eseguire l'aggiornamento delle colonne specificate). Se nel MODEL abbiamo specificato attr_accesible allora solo gli attributi specificati possono essere aggiornati. attr_accessible ci serve per prevenire il mass-assignment degli attributi che non dovrebbero essere editabili dagli utenti direttamente.

users#show

Quest'azione ha lo scopo di visulizzare una rappresentazione della risorsa. Non di tutte le risorse ma di una risorsa in particolare. L'azione viene accesa da ***GET /users/id*** e si riferirà appunto all'azione di visualizzazione della risorsa con quell'id. Ovviamente valgono le cose dette in precedenza, per quanto rigurada le varie rappresentazioni. L'azione è collegata quasi sempre ad una ***READ*** (***CRUD***) nel database.

```
def show

    @user = User.find_by_id(params[:id])

    if @user.nil?
       flash[:notice]="L'utente selezionato non esiste."
       redirect_to root_path()
    end

 end
```

e il rispettivo ***app/views/users/show.html.erb:***

```
  h1 class="alt"><%= put_message(:show) %></h1>

<table class="profile">
 <tr>

  <td class="profile_identity">
    <h2 class="alt">
      <%=gravatar_for @user%>
      <%=@user.name%>
    </h2>
  </td>

  <td class="round profile_sidebar">
    <strong>Name</strong> <%= @user.name %><br />
    <strong>URL</strong>  <%= link_to user_path, @user%><br />
    <strong>Post</strong> <%=%>
  </td>

 </tr>

</table>
```

Su questo file non c'è nulla da riferire che sia particolare dell'azione show. Quindi ne approfittiamo per parlare un po del layout dell'applicazione e dell'utilizzo alternativo di render nella view, per il rendering delle parti che costistuiscono il layout dell'applicazione. Il layout di tutte le pagine della nostra applicazione si trova in ***app/layouts/application.html.erb*** e contiene:

```
<!DOCTYPE html>
<html>
<head>

  <title><%=fill_title_tag%></title>
  <%=render 'layouts/stylesheets.html.erb'%>
  <%=render 'layouts/fixing_script.html.erb'%>

</head>

<body>
    <div class="container">
       <%= render  'layouts/header'%>
        <%= render  'layouts/flash_messages'%>


<!-- Finestra Contenuto principale -->

  <section id="main" class="round span-19 colborder">

     <%= yield %>

  </section>

<!--                          -->


  <%=render 'layouts/sidebar' %>

  <%=render 'layouts/footer' %>


</div>
```

```
</body>
</html>
```

Come vediamo render è utilizzato per includere nel rendering finale delle parti(***partials***). Ad esempio ***<%=render 'layouts/sidebar' %>*** andrà a cercare in app/views/layouts il file ***_sidebar.html.erb*** cioè una parte della presentazione che verrà inclusa prioprio in quel punto. Inoltre si noti l'utilizza di ***<%=yield%>:*** se ad esempio Rails sta producendo la presentazione show.html.erb, la produce inserendo il risultato dell'elaborazione di questo template, dentro il layout principale nel punto esatto indicato dal metodo ***yield().***

users#destroy

Infine arriviamo all'utlima azione REST, che è destroy. Quest'azione dichiede ovviamente l'eliminazione fisica della risorsa, e il più delle volte significa eseguire una ***DELETE*** di (***CRUD***) dentro il database. Solitamente la rappresentazione associata a questa richiesta, è una pagina html con un link che si riferisce alla risorsa da cancellare. L'azione viene accesa dal verbo DELETE di HTTP e in particolare da ***DELETE /users/id,*** quindi poiché i browsers non supportano questo varbi in maniera nativa, vediamo come Rails può ovviare a questo inconveniente, grazie a link_to e un po di javascript. Ad esempio per un utente che si sia autenticato come admin, (vedi gestione delle sessioni), la pagina index.html.erb apparirà un po differente. Infatti accanto a ciascun utente visualizzato apparirà un pulstante per l'eliminazione. Vediamo qui un frammento di index.html.erb modificata:

```
 <% if current_user.admin?%>

<span>|</span><%= link_to "Cancella",user_path(user.id),
                        :method=>:delete,
                        :confirm=>"Sei sicuro di voler eliminare #{user.name}?",
                        :title=>"Cancella #{user.name}"%>
    <%end%>
```

Vediamo come link_to oltre che i soliti 2 argomenti, qui riceva un Hash di opzioni: {:method=>:delete, :confirm=>”...”,:title=>”Cancella#{user.name}”} e il markup in output sarà :

```
<span>|</span><a href="/users/1" data-confirm="Sei sicuro di voler eliminare Pinco Pallino?" data-method="delete" rel="nofollow" title="Cancella Domenico D'Egidio">Cancella</a>
```

Se clicchiamo su cancella, viene avviata l'esecuzione AJAX con il supporto di Prototype per inserire una POST con un'opportuno parametro_method='delete'. Inoltre verrà visualizzata una finestra di conferma per l'operazione. Per introdurra ancora qualche funzionalità interessante di ActionController andiamo adesso ad implementare la gestione della sessione.

Gestione della sessione e cookies

Ogni qual volta si interagisce con un'applicazione Rails, il framework genera un id di sessione univoco. Internamente Rails esegue ***sid=ActiveSupport::SecureRandom.hex(16).*** Indipendentemente dal meccanismo di storing che abbiamo scelto, la sessione viene acceduta sempre nella stessa maniera, e cioè attraverso il metodo **ActionController::Base**.***session()*** che ritorna un *Hash* in cui possiamo recuperare/registrare tutti i nostri oggetti purchè serializzabili. Quindi ad esempio se volessimo inserire nella sessione un oggetto del model come *@user*, basterà fare:

session[:user]=@user

Ma dove va a finire *@user?* Dove è registrato? Possiamo scegliere diversi meccanismi di storing in

Rails:

1) **CookieStore** (default)
2) **RDBStore**
3) **MemCacheStore**
4) **ActiveRecordStore**

Il meccanismo di default è quello utilizzato nella nostra applicazione d'esempio; utilizzando questo meccanismo, l'oggetto ***@user verrà memorizzato in un cookie***, cioè il server non mantiene alcuna informazione in memoria. Rails segue questo algoritmo implicito:

1) ***sid=ActiveSupport::SecureRandom.hex(16)***

2) ***h={ :user=>@user, :session_id=>sid, 'flash'=>{} }***

quindi procede con la codifica base 64 della serializzazione dell'Hash:

3) ***data = ActiveSupport::Base64.encode64s(Marshal.dump(h))***

4) ***escaped_data_value = Rack::Utils.escape(data)***

questa operazione genera una stringa, che contiene codificati il sid, l'oggetto @user e anche eventuali messaggi della variabile flash. Perchè questi contenuti non possano essere corrotti ,Rails aggiunge un ***DIGEST*** criptato utilizzando una chiave generata da Rails e messa in ***/initializers/secret_token.rb.***

5) ***digest = 'SHA1'***

6) ***digest_value=OpenSSL::HMAC.hexdigest(OpenSSL::Digest::Digest.new(digest), secret, data)***

infine il digest viene aggiunto alla stringa prodotta prima:

7) ***final_output = "#{escaped_data_value}--#{digest_value}"***

final_output è il contenuto che possiamo osservare nel cookie. Si noti che nei cookies non possiamo inserire più di 4kb di dati. Il che non è male, visto che solitamente i dati da inserire in sessione non dovrebbero essere troppo complessi. Si noti però che questo cookie messo in automatico muore alla chiusura del browser. Nella nostra applicazione vogliamo utilizzere un cookie che non viene cancellato fin quando non lo decidiamo noi. Questo cookie contiene un remember token, cioè una coppia email e password criptati, che il browser rimanderà indietro, ad ogni nuova richiesta al server. La sessione, qui viene modellata come una risorsa dell'applicazione, e vogliamo in particolare che venga creata e distrutta. Pertanto nel meccanismo di routing andiamo ad abilitare le azioni REST new, create, destroy:

```
resource :session, :only=>[:new,:create,:destroy]
```

Quindi andiamo a generare il nostro controllore :

rails generate controller session new create destroy

e andiamone a implementare le azioni

Evoluzione del model: aggiunta di un'associazione ONE-TO_MANY

Andiamo avanti con lo sviluppo dell'applicazione, e modelliamo un'altra risorsa della nostra applicazione. In questo caso andiamo a generare una classe del MODEL che chiamiamo ***Micropost***. Questa è in relazione ***MANY->TO->ONE*** con la classe User. Con ***ActiveRecord*** è facile modellare questo tipo di relazioni; inoltre vedremo come sia facile inserire un ordinamento durante il recupero di oggetti di questo tipo dal database, e vedremo anche come stabilire una relazione ***parent/child*** con la classe User. Andiamo allora a generare come prima cosa il controllore adeguato alla nostra risorsa, ***microposts***

rails generate controller microposts

e quindi andiamo a generare la classe del model, e la relativa ***Migration:***

rails generate model Micropost content:string user_id:integer

In particolare quest'ultimo comando produrrà questa migrazione:

```
class CreateMicroposts < ActiveRecord::Migration
  def self.up
    create_table :microposts do |t|
      t.string  :content
      t.integer :user_id
      t.timestamps
    end
  end

  def self.down
    drop_table :microposts
  end
end
```

Vediamo che a livello di tabelle, la relazione verrà stabilita tramite una foreign key e quindi la relativa colonna va creata, al lato many. Questo risulterà ad un attributo shadow nella classe Micropost, che è user_id; eseguiamo adesso la migrazione:

rake db:migrate

prima di andare avanti, dobbiamo parlare di ***attr_accessible()*** è un metodo di ActiveRecord che ha lo scopo di evitare il *mass-assignment-attack*, cioè di evitare che dal web un utente possa impostare via apposite richieste, degli attributi che non dovrebbero essere settabili dall'esterno. Ad esempio tramite un client a linea di comando, un utente potrebbe richiedere …..... in ActiveRecord la relazione con navigabilità *MANY->TO->ONE* si chiama ***belongs_to*** e la possiamo impostare tramite il metodo ***belongs_to()*** :

```
class Micropost < ActiveRecord::Base

  belongs_to :user

end
```

questo metodo, crea un attributo ***user*** dentro Micropost e il relativo accessor che ritorna su richiesta un oggetto User. Dall'altro lato, la relazione *ONE->TO->MANY* va espressa con il metodo

has_many()

def class User < ActiveRecord::Base
[…]

has_many :microposts

end

questo metodo, va a creare l'*Array* ***microposts*** dentro la classe, che funziona da collezione di oggetti *Micropost*, e va quindi ad aggiungere all'array i seguenti metodi: (user è di tipo User)

> ***user.microposts.create(arg)*** che aggiunge un oggetto di tipo Micropost alla collezione, imposta ***user_id*** == ***user.id*** dove user è un'istanza di User, e quindi crea nel database il record relativo all'oggetto Micropost. Si noti che arg rappresenta gli argomenti in Hash per la costruzione dell'oggetto Micropost, e quindi quelli specificati in attr_accessible(), in questo caso quindi ***:content=>...***
>
> In caso di failure ritorna false.
>
> ***user.microposts.create!(arg)*** come sopra ma in caso di failure ritorna un'eccezione.
>
> ***user.microposts.build(arg)*** Ritorna un nuovo oggetto Micropost, con ***user.id*** == ***arg.user_id***

In particolare l'ultimo metodo non crea nulla nel database, ma mi serve a generare un'istanza di Micropost a partire da un'istanza di User, raggirando attr_accessible.

I18n
ActionMailer
ActiveResource
Web 2.0 Prototype, Scriptaculous, .RJS
BDD (Rspec per il testing)
(DEBUGGING)