Introducció al Ruby on Rails

Vicent Moncho Mas

PID_00194068

Universitat Oberta de Catalunya

Índex

Objectius

Amb l'estudi d'aquest mòdul, aconseguireu els objectius següents:

1. Conèixer què és el Ruby on Rails.

2. Entendre l'arquitectura bàsica d'una aplicació Ruby on Rails.

3. Conèixer els components del patró de disseny model-vista-controlador (MVC).

4. Aprendre a fer aplicacions senzilles amb el Ruby on Rails.

1. Què és el Ruby on Rails

El Ruby on Rails (RoR[1]) és un entorn de programació[2] que té com a objectiu facilitar el desenvolupament i el manteniment d'aplicacions web. Per a això es combinen un conjunt de característiques que ho fan possible:

(1)*RoR* és la sigla de Ruby on Rails.

(2)En anglès, *framework*.

- Totes les aplicacions es desenvolupen utilitzant l'arquitectura MVC. Això fa que cada peça de codi tingui el seu lloc definit, per la qual cosa és senzill saber on es troba i on l'hem d'introduir.

- El Rails facilita la creació de test unitaris, funcionals i d'integració sobre l'aplicació que es va creant. Aquesta característica representa un estalvi molt important de temps per al programador professional i un augment en la qualitat del programari desenvolupat.

- L'ús del Ruby com a llenguatge de programació modern i orientat a objectes és explotat amb l'objectiu de simplificar el codi final. La filosofia DRY[3] ('no et repeteixis') és implementada a partir de l'ús de classes de Ruby que defineixen els objectes necessaris per al patró de disseny model-vista-controlador (MVC[4]).

- L'ús de conveni en lloc de configuració; per a això s'utilitzen els valors definits per defecte, la qual cosa simplifica la necessitat de definir diferents configuracions.

(3)*DRY* és la sigla de l'expressió anglesa *don't repeat yourself*.

(4)*MVC* és la sigla de *patró de disseny model-vista-controlador*.

Són diversos els factors que fan que RoR sigui una bona elecció per al programador d'aplicacions web, però cal destacar que la corba d'aprenentatge és més petita que en altres tecnologies si es disposa d'experiència en el món de la programació web.

1.1. Instal·lació de l'RoR

El Ruby on Rails és un entorn de programació format per diversos components i cadascun d'aquests ha de ser instal·lat en el lloc de treball. En els subapartats següents s'explicaran els passos per a procedir amb la instal·lació en l'entorn Windows.

L'entorn Ruby on Rails necessita el programari següent:

- L'intèrpret del llenguatge de programació Ruby. Es recomana la versió 1.8.7 o 1.9.2, ja que aquestes dues són suportades pel Rails 3.0, que és la versió sobre la qual hem desenvolupat aquest mòdul.

- El sistema de paquets de Ruby, més conegut com a RubyGems. S'ha d'utilitzar la versió 1.3.6, que és la que utilitzem en el mòdul.

- L'entorn de desenvolupament Rails. En el mòdul s'utilitza la versió 3.0.

- S'instal·laran biblioteques[5] de programari específiques per a certes tasques.

- Es necessita un sistema gestor de base de dades. En aquest mòdul utilitzarem SQL-lite, ja que és nadiu i totalment adaptat al Ruby on Rails.

- Un editor de text per a la manipulació del codi. En aquest sentit es recomana el NotePad++, encara que hi ha moltes alternatives vàlides i sempre depèn del gust del programador.

(5)En anglès, *libraries*.

1.1.1. Instal·lació en el Microsoft Windows

1) Instal·lació del Ruby

El primer pas és la instal·lació del Ruby. Per a això s'abaixarà de l'espai de fitxers de l'aula el paquet següent d'instal·lació del Ruby per a Windows:

```
rubyinstaller-1.8.7-p302.exe
```

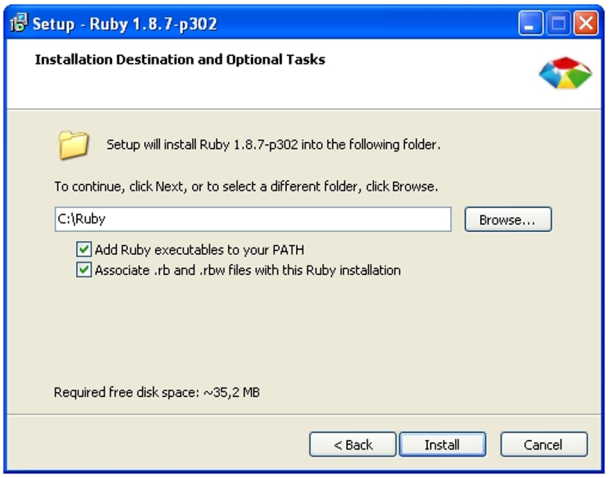
En executar el fitxer, s'ha d'acceptar la llicència i seleccionar les caselles de selecció[6] ; es canviarà el directori a `C:\Ruby` i es premerà *Install*, tal com s'observa en la figura 1.

Reflexió

Els exemples del mòdul s'han desenvolupat amb la versió 1.8.7, que és la versió recomanada. En cas d'utilitzar una versió diferent, és possible que aquests no funcionin.

(6)En anglès, *check-boxes*.

Figura 1. Finestra d'instal·lació del Ruby

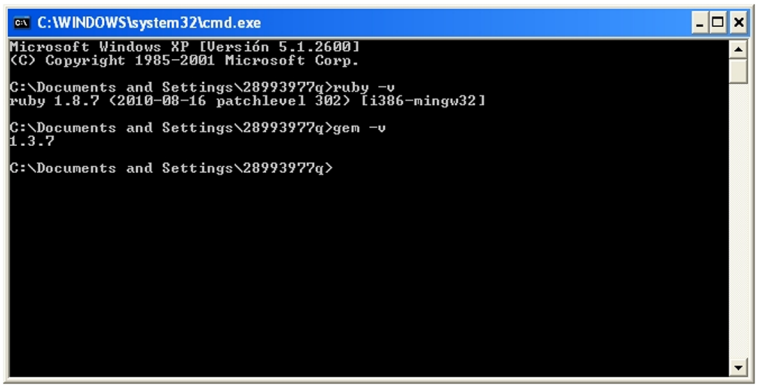


Per a comprovar la instal·lació s'executen les sentències següents en una finestra d'ordres de Windows:

```
ruby -v
gem - v
```

El resultat serà el que s'observa en la figura 2.

Figura 2. Finestra de confirmació de la versió del Ruby i del RubyGems



Si no s'utilitza el fitxer instal·lable de l'aula, és imprescindible que la versió del RubyGems sigui superior a la 1.3.5. L'actualització a l'última versió es fa amb les ordres següents:

```
gem update --system
gem uninstall rubygems-update
```

2) Instal·lació de l'SQLite

La instal·lació de l'SQLite requereix els fitxers següents (disponibles en l'aula):

```
sqlite-dll-win32-x86-3070701.zip
sqlite-shell-win32-x86-3070701.zip
```

El primer fitxer conté l'aplicació que permet l'accés i la modificació de bases de dades SQLite i el segon les biblioteques DLL del sistema (`sqlite3.def`, `sqlite3.dll` i `sqlite3.exe`).

El procés d'instal·lació es basa en la descompressió dels fitxers en la carpeta `C:\Ruby\bin`. S'ha de tenir cura especial que aquests fitxers es trobin en el directori mateix i un subdirectori. Fet el punt anterior, s'instal·len els enllaços entre el Ruby on Rails a l'SQLite3 mitjançant l'ordre següent:

```
gem install sqlite3-ruby
```

L'execució de l'ordre anterior requereix una connexió a Internet activa. En cas d'haver-hi un servidor intermediari[7] , s'ha de modificar la sintaxi per a introduir les característiques d'aquest últim.

(7)En anglès, *proxy*.

3) Instal·lació del Rails

La preparació de l'entorn finalitza amb la instal·lació del Rails. En el mòdul s'utilitza la versió 3.0.0. Aquesta s'instal·la amb l'ordre següent:

```
gem install rails –v 3.0.0
```

Es comprova la versió de Rails instal·lada amb l'ordre següent:

```
rails -v
```

Que retornarà el valor 3.0.0.

1.1.2. Instal·lació en altres OS

Encara que el Ruby on Rails és independent dels sistemes operatius, la instal·lació de l'entorn en OS X o Linux té gran dependència de la versió dels sistemes operatius, per la qual cosa els procediments d'instal·lació depenen de la versió sobre la qual s'instal·la i, per tant, és molt complex definir un únic procediment.

Per això no s'explicaran els procediments d'instal·lació per a aquests sistemes operatius, i en cas que es necessiti, es donaran les pautes o recomanacions especifiques en cada versió de sistema operatiu, tant basat en Linux com en OS X.

1.1.3. L'editor de codi

El desenvolupament d'aplicacions Ruby on Rails no necessita complexos entorns de desenvolupament integrats (IDE[8]) tal com s'esdevé en altres llenguatges de programació. La majoria de programadors actuals de Ruby on Rails utilitzen un simple editor de text que compleix una sèrie de requisits bàsics com els següents:

(8)*IDE* és la sigla de l'expressió anglesa *integrated development environment*.

- L'editor hauria de suportar codi Ruby i HTML per a facilitar el treball amb fitxers erb.

- És recomanable que aquest pugui crear l'esquelet de les estructures estàndard de Ruby.

- És una ajuda que la navegació pels fitxers sigui senzilla (això s'entendrà més endavant quan s'expliqui l'estructura del Rails).

- A causa que en el Ruby on Rails els noms tendeixen a ser llargs, és interessant que l'editor suggereixi noms a partir de la introducció de diversos caràcters.

Però cada programador té els seus propis costums, preferències o manies, per la qual cosa l'elecció final és responsabilitat d'aquest. En el cas que no es tingui un editor habitual, una recomanació avalada en la Xarxa és la del TextMate en l'entorn OS X, l'equivalent del qual en Windows és l'E-TextEditor, encara que un simple editor com el NotePad++ és suficient per a seguir aquest mòdul.

En el moment de creació d'aquest mòdul hi ha complexos entorns IDE amb suport per a Ruby on Rails, com l'Aptana, el Komodo o el NetBeans, però no es recomanen per al seguiment d'aquest mòdul ja que, a més d'aprendre Ruby on Rails, s'ha d'aprendre a utilitzar aquests entorns, la qual cosa afegeix una dificultat innecessària als objectius inicials.

1.2. El llenguatge de programació Ruby

El Ruby és un llenguatge de programació interpretat i orientat a objectes inspirat en altres llenguatges com Python, Perl i Smalltalk. El resultat és un llenguatge de programació concís, llegible i potent.

Ruby

El Ruby va ser creat per Yukihiro "Matz" Matsumoto, programador japonès. En va iniciar el desenvolupament el 1993 i el va presentar l'any 1995.

L'entorn de Matsumoto anomenava aquest llenguatge *robí*, ateses les similituds amb el llenguatge Perl, que significa *perla*.

En aquest apartat es plantejaran les característiques bàsiques del llenguatge que permeten seguir sense problema els algorismes que s'explicaran en els apartats següents del mòdul. Per tant, no es farà un estudi detallat del llenguatge de programació Ruby; ja que aquest donaria per a un mòdul complet, o, possiblement, per a una assignatura completa.

1.2.1. Principals característiques

Les característiques següents defineixen el Ruby com un llenguatge orientat a objectes, modern i senzill:

a) Està orientat a objectes: tots els seus tipus de dades són objectes, fins i tot els simples com els numèrics, els booleans, els valors nuls, etc. Per tant, en Ruby no hi ha un tipus de dades nadiu com en llenguatges com el JavaScript.

Sense tipus de dades natius

El Ruby és un llenguatge de codi obert i gratuït. Pot ser usat, copiat, modificat i distribuït lliurement.

b) És un llenguatge interpretat: disposa d'intèrprets per a diferents arquitectures i sistemes operatius i aquests permeten l'execució de programes escrits en Ruby sense compilació prèvia. Aquesta característica té grans avantatges, com la independència de plataforma i la reflexió. S'utilitzarà l'intèrpret de Ruby per a provar els conceptes que es presentaran en l'apartat següent.

Independència de la plataforma

Una aplicació Ruby és fàcilment portable a altres sistemes operatius com GNU/Linux, UNIX, Mac OS X i tota la família de sistemes operatius Windows.

c) És un llenguatge reflexiu: es pot generar i executar codi creat de manera dinàmica. Per exemple, és possible executar codi Ruby contingut dins d'una cadena de caràcters en temps d'execució. Això implica que podem modificar el codi que s'executarà durant el procés d'execució.

d) Disposa d'una biblioteca estàndard: disposa d'un conjunt de classes i funcions escrites en Ruby que proporcionen suport per a fer determinades operacions: tipus abstractes de dades, connectors[9] , gestió de dates, etc.

[9] En anglès, *sockets*.

e) Disposa d'un *garbage collector*: igual que en llenguatges com Java o JavaScript, la destrucció i alliberament de memòria es gestiona de manera automàtica sense intervenció del programador.

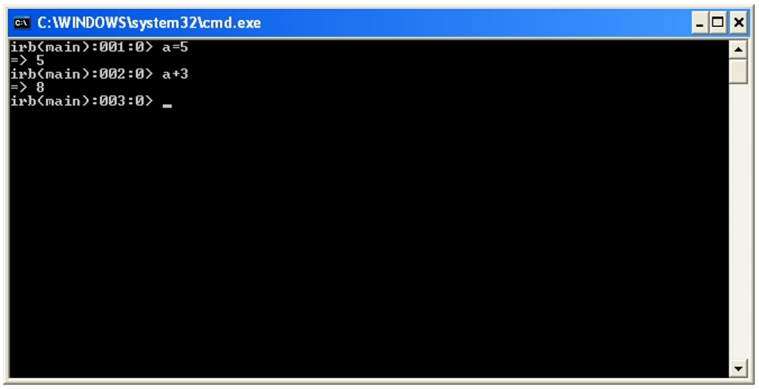
1.2.2. Tipus de dades

Tal com s'ha comentat, una de les característiques del Ruby on Rails és la convenció enfront de la configuració. Això afecta també com s'anomenen les variables, els objectes, els mètodes, etc. En particular, les regles següents són "de compliment obligat":

- Els noms de les variables, mètodes i els paràmetres d'aquests últims han de començar amb una lletra minúscula: `m_valor`, `s100`.
- Les instàncies de les classes han de començar amb el símbol `@`: `@registre`, `@menu`.
- Els noms compostos se separen utilitzant el símbol "_": `valida_valor()`, `desa_menu`.
- Els noms de les classes, mòduls i constants comencen amb una lletra majúscula i en aquest cas la separació de noms compostos s'implementa amb una lletra majúscula: `MenuPrincipal`, `Comensal`.

L'execució de les sentències i estructures es fa obrint una consola de Windows i executant l'ordre `irb`. Aquesta ordre executa l'intèrpret de Ruby i permet a l'usuari executar codi (figura 3).

Figura 3. Exemple d'execució d'ordres en l'intèrpret IRB



L'assignació de variables i els diferents operadors (`+`, `-`, `*`, `/`, `**`) es comporten de la mateixa manera que en la resta de llenguatges de programació. Es pot utilitzar l'intèrpret per a practicar amb aquests operadors.

El fet que cada valor és un objecte fa que quan s'assigni un valor a una variable, realment s'està creant una instància d'una classe (un objecte), i per tant es disposa de mètodes i propietats que es poden utilitzar. Per exemple, una variable amb un valor enter (per tant, un objecte) disposa de mètodes com `to_s` (que converteix el valor a *string*), `to_f` (que converteix el valor a coma flotant) i `next` (que proporciona l'enter següent).

Reflexió

El detall de cadascuna de les classes que defineixen tipus de dades no és objectiu d'aquesta assignatura, però es pot consultar en línia quan es necessiti en Index of Classes & Methods in Ruby 1.9.3.

Cadenes

Les cadenes de text es defineixen delimitant el text mitjançant l'ús de cometes, simples o dobles.

```
cadena1 = "aquesta és una cadena de caràcters\n"
cadena2 = 'i aquesta és una altra'
```

La diferència entre les dues definicions anteriors és que les cadenes definides mitjançant cometes dobles poden incloure caràcters especials com `\t` (tabulador), `\n` (retorn), i nombres en diferents representacions (octals, `\061`, hexadecimal, etc.).

Una característica interessant és que és possible introduir en una cadena el valor emmagatzemat en una variable utilitzant la notació `#{}`. Per exemple, en el codi següent, s'assigna a la variable `cadena` el text "L'edat és 25".

```
edat = 25
cadena = "L'edat és #{edat}"
```

Igual que en els tipus numèrics, la classe `string` disposa d'un conjunt ampli de mètodes, entre els quals destaquen les funcions següents relacionades amb lletres minúscules i majúscules.

```
"hotel".upcase          #-->   "HOTEL"
"Motocicleta".downcase  #-->   "motocicleta"
"Az".swapcase           #-->   "aZ"
"sam".capitalize        #-->   "Sam"
```

S'accedeix a les lletres d'una cadena a partir de l'índex que en defineix la posició, introduint aquest entre claudàtors `[]` (en Ruby el primer caràcter té l'índex zero).

```
y = "Tomate"
y[0]                    #--> "T"
y[1]                    #--> "o"
y[0] = "z"              #--> "zomate"
```

Els mètodes `delete`, `insert` i `reverse` insereixen, eliminen i inverteixen la posició de les lletres d'una cadena.

```
y.delete("T")           #-->   "omate"
y.insert(2,"c")         #-->   "Tocmate"
y.reverse               #-->   "etamcoT"
```

Per finalitzar amb les cadenes, se selecciona una secció d'una cadena especificant el rang d'aquesta.

```
n = "0123456789"
n[5...n.length] # subcadena des del cinquè element fins al final"56789"
```

Arrays

L'estructura *array* en Ruby és molt similar a la que hi ha en altres llenguatges de programació. Consisteix en un conjunt de dades discretes que són accessibles per l'índex que en defineix la posició.

Estructura *array*

```
a = ["Menú", "de ", "9", "euros"]
a.size                          #--> 4 el
```

`a[i]` ens retorna el (*i* – 1)-èsim element de l'*array*.

```
a[3]                            #--> "eur
```

El mètode `split` converteix una cadena en un *array* i `concat` afegeix, al final de l'*array*, l'*array* indicat entre parèntesis.

Mètode *split*

```
b= "i 50 cèntims".split #converte

a.concat(b)       #--> ["Menú", "d
```

Les funcions següents són bàsiques en el maneig d'*arrays*:

```
a.first    #--> "Menú" retorna el primer element de l'array

a.last     #--> "cèntims" retorna l'últim element de l'array

a.empty?   #--> false pregunta si l'array està buit
```

Un *array* pot contenir al seu torn altres *arrays* (i convertir-se en un *array* multidimensional):

Array multidimensional

```
c = [1, 2, 3, [4, 5, [6, 7, 8], 9
```

El mètode `flatten` converteix l'*array* multidimensional en unidimensional:

Mètode *flatten*

```
c.flatten                    #--> [1
```

Per a inserir elements, s'utilitza el mètode `insert(índex, valor)`:

Mètode `insert`

```
c.insert(3, 99)              #--> [1
```

El mètode `reverse` permet invertir l'ordre dels elements de l'*array*:

Mètode `reverse`

```
["a ", "b", "c"].reverse  #--> ["
```

És possible ordenar els elements alfabèticament utilitzant el mètode *sort*:

Mètode `sort`

```
["p", "z", "b", "m"].sort #--> ["
```

Hash

Es tracta d'un *array* en el qual els valors estan indexats per cadenes en lloc de nombres enters.

Normalment s'usa un *hash* per a emmagatzemar parells de valors relacionats entre si, com les paraules d'un diccionari i les seves definicions, les vegades que apareix cada paraula en un text o les propietats d'un objecte i els seus valors respectius.

Exemple

A continuació es crea un *hash* buit al qual es van afegint elements:

```
ingredients = Hash.new
ingredients["arròs"] = "Bomba"

ingredients["oli"] = "Oliva verge
```

Un *hash* es pot inicialitzar amb tots els valors des del principi:

```
llenguatge= {                          # declara un hash en la variable llenguatge
'Perl' => 'Larry',
'Python' => 'Guido',                   # el signe "=>" separa clau de valor
'Ruby' => 'Matsumoto'
}
```

A continuació es mostren diferents mètodes de la classe *hash*:

```
llenguatge['Ruby']                     # retorna el valor de la clau
llenguatge.has_key?('Java')            # retorna false ja que no existeix la clau
llenguatge.has_value?('McCarthy')      # retorna false ja que no existeix valor
llenguatge.sort                        # ordena per ordre de la clau
llenguatge.values_at('Python','Ruby') # retorna els valors ['Guido', 'Matsumoto']
llenguatge.values                      # retorna els valors ['Larry', 'Guido', 'Matsumoto']
llenguatge.keys                        # retorna les claus ['Perl', 'Python', 'Ruby']
llenguatge.length                      # retorna el nombre d'elements
```

Per a reemplaçar els valors existents, es fan simplement assignacions amb l'índex i el valor:

```
llenguatge['Ruby'] = 'Matz'           # reemplaça 'Matsumoto' per ' Matz'
llenguatge['COBOL'] = 'CODASYL'       # agrega un nou element al hash
```

Per a eliminar parells que ja no es necessiten, es disposa de la funció `delete`:

```
llenguatge.delete('COBOL')            # elimina el parell 'COBOL'=>'CODASYL'
```

Dates i hores

El Ruby disposa d'un mòdul anomenat `Time` que s'utilitza per a la manipulació de dates i d'hores. La data i hora actual s'obté amb el mètode `now`:

```
t = Time.now# "Mon Sep 10 23:11:06 +1000 2011"
```

El resultat és una cadena formada per lletres i nombres, als quals és possible accedir individualment mitjançant els següents mètodes:

```
t.day               # 10
t.month             # 9
t.year              # 2011
t.hour              # 23
t.min               # 11
t.sec               # 6
t.wday              # 1 primer dia de la setmana
t.yday              # 253 dia de l'any
t.strftime("%B")    # "September" nom del mes complet
t.strftime("%b")    # "Sep" idem abreujat
t.strftime("%A ")   # "Monday" dia de la setmana
t.strftime("%a ")   # "Mon" idem, abreujat
t.strftime("%p")    # "PM"  AM o PM
```

Una aplicació pràctica dels mètodes anteriors és el càlcul de la diferència entre dos moments de temps.

Aplicació pràctica

La creació d'un objecte del tipus data es fa amb el mètode `mktime()`:

```
vacances = Time.mktime( 2011, "de
```

D'altra banda, la classe `Date` s'utilitza per a especificar dates que no requereixen l'ús del temps.

```
require 'date'
data = Date.new(2011, 08, 10)
data.to_s                                    # "2011-08-10"
avui = Date.today
puts "#{avui.day}/#{avui.month}/#{avui.year}"  # "12/08/2011"
```

Aquest subapartat finalitza amb l'accés als elements d'un *array* o un *hash* de manera seqüencial. La funció `each` s'aplica a un *array*, a un *hash* o a un *string* de diverses línies amb la sintaxi següent:

```
variable.each {bloc}
```

La sentència provoca l'aplicació de les instruccions que es troben dins del bloc en cadascun dels elements de la variable. És a dir, es produeix una enumeració dins d'un cicle repetitiu en què en cada iteració s'avalua un dels elements de l'array, *hash o string*.

Exemple amb un *array*

L'exemple següent utilitza un *array* i mostra cada element de l'*array* amb una sentència molt simple:

```
a = [2,4,13,8]
a.each {|i| puts i}
```

En lloc de les claus `{}` de l'`each`, es poden usar les paraules clau `do` i `end`. Aquesta sintaxi permet escriure el bloc de codi en diverses línies, i és recomanable quan el contingut del bloc tingui una lògica més elaborada, que sigui difícil expressar en una línia de codi.

Exemple amb un *hash*

En l'exemple següent s'utilitza un *hash*:

```
b = {'sol'=>'dia', 'lluna'=>'nit'
b.each {|k,v| puts k + " : " + v}
```

```
linies = "En un lugar de la Mancha, de cuyo nombre ...Fi\n".split
num = 0
linies.each do |linea|        #usat al principi del bloc
   num += 1                   # el contingut del bloc va pel mig
   print "Línia #{num}:       #{linia}"
                              # s'estén sobre diverses línies
end
```

Per tant, es tracta d'un mecanisme que permet fer iteracions sobre estructures complexes, encara que no substitueix els mecanismes clàssics d'iteració, que s'estudien en el subapartat següent.

1.2.3. Estructures de control i iteratives

Estructures de control

Les estructures condicionals són similars en la majoria dels llenguatges de programació. En Ruby, l'estructura de control `if/elsif/else/end` té la sintaxi següent:

```
if expr1 [then|:]
   bloc1
[elsif expr2 [then|:]
   bloc2]
[else
   bloc3]
end
```

S'executarà el bloc de codi en el qual l'`expr1` o `expr2` adquireix el valor *cert*.

La sentència `then` es pot ometre, i en lloc d'aquesta es poden usar dos punts, "`:`", encara que aquests també són opcionals.

El Ruby disposa d'una sintaxi compacta que s'utilitza per a implementar estructures condicionals més simples:

```
x = a > b ? 1 : 0
```

Que assigna el valor 1 a x si $a > b$, i 0 en cas contrari.

Exemple

A continuació es presenta un exemple clàssic de l'ús de l'estructura, en el qual es comparen dos nombres:

```
   a = 5
   b = 3
if a > b
   puts "a és més gran que b"
elsif a == b
   puts "a és igual que b "
else
   puts "b és més gran que a ".
end
```

Quan es vol comparar una sola variable amb una quantitat de valors s'utilitza l'estructura `case`. La sintaxi d'aquesta estructura és la següent:

```
case variable
when valor1
   bloc1
[when valor2
   bloc2]
[else
   bloc3]
end
```

Quan la variable té un valor igual que algun dels examinats, llavors s'executa el bloc associat a aquest valor i la resta es descarten. En cas que la variable no coincideixi amb cap dels valors examinats, llavors s'executa la secció `else` i el `bloc3` (aquest últim cas és convenient per a atrapar dades que estan fora dels rangs esperats). En qualsevol cas, solament s'executa un dels blocs, i els altres són descartats.

En l'expressió anterior, els blocs `when` i `else` són opcionals; els blocs `when` poden ocórrer almenys una vegada i tantes com es necessitin. Si els valors comparats són numèrics, l'expressió `when` pot tenir valors puntuals (discrets), rangs o expressions regulars.

Exemple

A continuació es planteja un exemple numèric de l'estructura:

```
i = 8
case i
when 1, 2..5
   print "i està entre 1..5"
when 6..10
   print "i està entre 6..10"
else
   print "Error, dada fora de ran
end
```

Estructures repetitives

En el subapartat anterior es va presentar un cas especial de repetició que utilitza la funció `each`, amb la qual s'itera sobre elements d'un *array*.

```
a = ["a","b","c"]
a.each { |v| puts v}
```

Es pot aconseguir el mateix resultat amb l'estructura `for/in`.

```
for e in a
   puts e
```

```
end
```

Quan es vol repetir un cicle un nombre definit de vegades, el rang s'indica amb els límits de la iteració separats per dos punts.

```
for i in 5..10
   puts i
end
```

En el cas que no es vulgui iterar fins a l'últim valor, sinó que la iteració es detingui en el valor penúltim (del 5 al 9 en l'exemple anterior) es pot indicar afegint un tercer punt. D'aquesta manera quan s'utilitzen tres punts s'exclou l'últim valor del rang, i aquesta sintaxi és molt útil quan s'itera sobre un *array* (l'últim índex del qual és $n - 1$).

```
a = ["a","b","c"]
for i in 0...a.size
   puts "#{i}:#{a[i]}"
end
```

En el cas que el nombre de repeticions sigui conegut, es pot utilitzar la sentència `times`:

```
n.times do
   puts "hola"
end
```

Repetiria n vegades la sentència `puts 'hola'`.

Una alternativa és l'ús del mètode `upto` en un nombre enter:

```
1.upto(5) do|i|
   puts "Hola #{i}"
end
```

De la mateixa manera es pot utilitzar el mètode `downto`:

```
5.downto(1) do |i|
   puts "#{i}:Hola"
end
```

Per a implementar iteracions amb passos diferents d'un, s'utilitza el mètode `step`, en què el primer paràmetre és el nombre final i el segon paràmetre indica l'increment que es produeix en cada iteració.

```
2.step(10, 2) do |i|
   puts i
```

```
end
```

És interessant l'increment que produeix `step` quan s'utilitzen nombres reals, ja que els increments són fraccionaris:

```
2.step(10, 0.5) do |r|
   puts r
end
```

L'estructura `while` avalua una expressió el resultat de la qual és un valor booleà i repeteix la iteració tantes vegades com l'expressió sigui certa.

```
compte = 0
while (compte < 5) do
   puts compte
   compte += 1
end
```

En l'estructura `while` l'ús de la paraula `do` és opcional.

De la mateixa manera l'estructura `until` és similar a `while`, excepte que l'expressió avaluada té una lògica negativa, és a dir, el bloc es repeteix mentre la condició sigui falsa.

```
compte = 0
until compte >= 5 do
   puts compte
   compte += 1
end
```

L'estructura `loop` crea un bucle inicialment infinit però per a sortir-ne s'utilitza la instrucció `break`, juntament amb una condició que l'executa. L'exemple anterior es podria programar de la manera següent:

```
compte = 0
loop do
breakif compte >= 5
puts compte
compte += 1
end
```

Un avantatge d'aquesta estructura és el fet que la iteració pot finalitzar en qualsevol moment sense necessitat d'executar tot el codi inclòs en el cos.

1.2.4. Classes i mòduls

Classes

La definició de classes es fa de manera similar que en la resta de llenguatges de programació orientats a objectes. En Ruby s'han de tenir en compte alguns aspectes sobre les variables i els mètodes de la classe, que solament s'usaran dins de la classe, mentre que altres seran visibles des de l'exterior de la classe:

- Les variables internes les anomenarem *variables de la instància*, i per a distingir-les de la resta de variables, van precedides del signe `@`.

- Les variables visibles des de l'exterior, es coneixen com a *atributs*. Són propietats de l'objecte i es poden llegir i modificar. Els atributs són substantius i els seus valors són adjectius o nombres. Per exemple: `gos.color`, `planeta.gravetat`, `motor.cilindres`, `pacient.edat`.

Les funcions que es defineixen en les classes es coneixen com a **mètodes** de la classe i s'utilitzen verbs per a descriure el tipus d'accions que fan. Per exemple: `gos.bordar`, `motor.arrencar`, `pacient.crear`.

Exemple

A continuació es defineix una classe que defineix un gos que té nom i borda:

```
class Gos                  # nom de la classe

def initialize(nom)        # mètode per inicialitzar
   @nom = nom              # @nom variable interna
   @lladruc = "bup"        # @lladruc variable interna
end

def to_s                   # representació textual
   "#{@nom}: #{@lladruc}" # retorna valors interns
end

def nom                    # exporta nom com a propietat pública
   @nom                    # retorna el valor de @nom
end

def nom=(nomNou)           # mètode que permet modificar el nom
   @nom = nomNou           # assigna el valor a la variable interna
end

def<=>(gos)                # operador de comparació per a ordenar
   @nom <=> gos.nom        # compara per nom
end

def bordar                 # defineix mètode bordar
   @lladruc                # retorna el lladruc
end

end
```

En la definició de la classe anterior es tenen en compte els aspectes següents:

- El nom de la classe sempre comença en majúscula.

- Les variables que es declaren dins de la classe tenen abast local per defecte i van precedides pel signe `@`.

- El mètode `initialize()` es coneix com a **constructor**, ja que és el primer mètode que s'executa quan es crea una instància de la classe. El constructor és opcional, però se sol utilitzar per a efectuar qualsevol tipus d'inicialització dins de la classe. Aquest mètode s'executa automàticament (mai no s'invoca explícitament) en crear una instància de la classe amb el mètode `new`.

- És convenient també definir el mètode `to_s` en què s'indica què s'ha de retornar quan se sol·licita la representació textual de la classe.

- El mètode de comparació <=> es defineix per a fer que la classe sigui comparable, és a dir, que se li puguin aplicar operacions com `sort()`. Consisteix a declarar quina variable interna s'utilitzarà per a implementar comparacions. En l'exemple anterior, les comparacions s'executen per nom.

- El mètode `nom()` retorna el nom de l'objecte.

- El mètode `bordar()` fa que el gos bordi (solament retorna el valor de la variable `@lladruc`).

Però una vegada està definida la classe, es crearà una instància i això s'implementa indicant el nom de la classe seguit de la paraula `new` i els possibles paràmetres que s'hagin definit en el mètode `initialize`.

Exemple

```
f = Gos.new("fifi")
m = Gos.new("milu")
puts f.bordar
puts m.bordar
puts f.to_s
puts m.to_s
 m <=> f
```

En la classe `Gos` no es pot accedir al lladruc directament, ja que és una variable interna. Ara bé, es pot fer visible un atribut intern a partir de la definició d'un mètode que retorni el valor d'aquest.

Exemple

```
class Gos

def initialize(nom)
   @nom = nom
...
end

def nom      # propietat pública llegible
   @nom      # retorna el nom
end
...
end
```

A continuació s'utilitza el mètode públic definit per a accedir al valor del nom de la classe:

```
f.nom
```

Però perquè la propietat pugui ser modificada és necessari definir un mètode que en modifiqui el valor:

```
class Gos
...
def nom=(nomNou)
   @nom = nomNou
end
...
end
```

A continuació s'utilitza el mètode anterior per a modificar el nom de `fifi`.

```
f = Gos.new("fifi")
f.nom = "fifirucho"
```

El Ruby permet implementar aquesta característica de manera més senzilla, encara que no és una manera gaire ortodoxa en el món de la programació orientada a objectes. Es poden declarar atributs llegibles utilitzant la paraula clau `attr_reader`.

```
class Gos
   attr_reader :nom, :data_de_naixement, :color
...
end
```

Però s'ha de modificar el constructor perquè aquest accepti els valors inicials:

```
class Gos
...
def initialize(nom, naixement, color)
   @nom = nom
   @data_de_naixement = naixement
   @color = color
end
...
end
```

Amb el codi anterior es pot accedeix a les propietats de la manera següent:

```
f = Gos.new("fifi", "20000621","gris")
f.color
```

És possible definir un atribut com a modificable afegint `attr_writter` a la declaració d'aquest. Amb la sintaxi anterior, l'atribut serà actualitzable directament, tal com es pot observar en els exemples següents:

```
class Gos
   attr_writer :alies
...
```

```
end
```

I llavors seria modificable utilitzant la sintaxi següent:

```
f = Gos.new("fifi", "20000621","gris")
f.alies = "fifirucho"
```

Però si es necessita tant poder accedir als atributs com modificar-los, s'utilitza `attr` en la declaració.

```
attr :nom, true
```

El valor booleà especifica que a més d'accessible sigui modificable.

Les variables precedides per dos punts es coneixen com a **símbols** i són valors constants, el valor dels quals és una cadena amb el mateix nom, i no poden ser modificats o actualitzats.

Mòduls

> Els mòduls proporcionen un mecanisme per a agrupar codi que pugui ser reutilitzat.

La creació d'un mòdul és molt simple ja que, com es pot observar a continuació, s'agrupen funcions i classes. Utilitzen la sintaxi següent:

```
module ElMeuModul
   PHI = 1.61803398874989    #el quocient auri

def lamevafuncio
   puts "Una salutació des de lamevafuncio"
end

class LaMevaClasse

def elmeumetode
   puts "Una salutació des de LaMevaClasse.elmeumetode"
end

end

end
```

Per a utilitzar el mòdul es necessita una sentència `require` que indiqui el nom del mòdul que es vol utilitzar.

```
require 'ElMeuModul'
quocient = ElMeuModul::PHI
c = ElMeuModul::LaMevaClasse.new
c.elmeumetode
```

En una classe es poden incloure funcions definides en un mòdul (utilitzant la sintaxi `include`), de manera que se n'amplia la funcionalitat:

```
require 'ElMeuModul'

class LaMevaClasse1
   include ElMeuModul
end

class LaMevaClasse2
end

m1 = LaMevaClasse1.new
m1.lamevafuncio
m2 = LaMevaClasse2.new
m2.extend(ElMeuModul)   # estén la instància
m2.lamevafuncio         #"Una salutació des de lamevafuncio"
```

La tècnica anterior es coneix com a **polimorfisme d'interfícies** i permet ampliar classes i assignar-los un comportament especial compartit.

Un mòdul predefinit que s'utilitza normalment d'aquesta manera és `Enumerable`. Es defineixen els mètodes `initialize` i `each` i, en implementar el polimorfisme, mètodes com `collect`, `detect`, `map`, `each_with_index` són heretats i, per tant, es poden utilitzar en la instància de la classe.

Exemple de polimorfisme d'interfícies

En l'exemple següent s'observa aquesta potent tècnica.

```
class MultiArray
include Enumerable

def initialize(*arrays) #accepta una col•lecció d'arrays
   @arrays = arrays
end

def each                #per cadascun, n'itera el contingut
   @arrays.each { |a| a.each { |x| yield x } }
end

end
```

A continuació utilitzem la classe anterior:

```
ma = MultiArray.new([1, 2], [3], [4])
ma.collect                  # [1, 2, 3, 4]
ma.detect { |x| x > 3 }     # 4
ma.map { |x| x ** 2 }       # [1, 4, 9, 16]
```

```
ma.each_with_index { |x, i| puts "L'element #{i} és #{x}" }
```

1.3. L'entorn de programació Rails

En aquest subapartat s'expliquen els conceptes bàsics que defineixen el Rails i que provoquen que aquest entorn de programació sigui considerat pels programadors com un entorn simple, potent i de manteniment fàcil.

1.3.1. La filosofia "convenció enfront de configuració"

Per a simplificar el procés de desenvolupament és fonamental simplificar al màxim les decisions que ha de prendre el programador. El Rails pren aquestes decisions i les defineix per defecte. Aquesta tècnica es coneix com a **valors per convenció**.

La filosofia anterior és la base de l'entorn de programació. Amb la configuració per defecte es pot crear una aplicació totalment funcional en un temps rècord (tal com es veurà en el subapartat següent).

Les opcions definides per defecte de Rails impliquen les opcions següents:

- Configuració complexa d'accés a bases de dades.
- Configuració de la comunicació entre l'aplicació del servidor i la de l'usuari.
- Organització dels directoris i del seu contingut.

Tota l'estructura per defecte compleix el patró model-vista-controlador.

1.3.2. El patró model-vista-controlador

El patró model-vista-controlador es basa en la separació de la interfície d'usuari, les dades i la lògica de control, en tres components completament diferenciats.

El model especifica a més el mecanisme que defineix la relació entre aquests:

a) Model: gestiona les dades de l'aplicació; s'encarrega de la càrrega, persistència i integritat, i fa la connexió amb la base de dades, els fitxers binaris, XML, etc. Tal com es veurà més endavant, en el model no solament es gestiona l'accés a les dades sinó també les regles de validació d'aquestes, i separa aquestes dels altres dos components.

b) Vista: és el component responsable de generar les interfícies d'usuari. Part de les dades mostrades en les interfícies es basen generalment en les dades obtingudes a partir del model. En una aplicació web, les vistes representen el conjunt de pàgines HTML (i les generades de manera dinàmica), codi JavaScript i estils CSS associats.

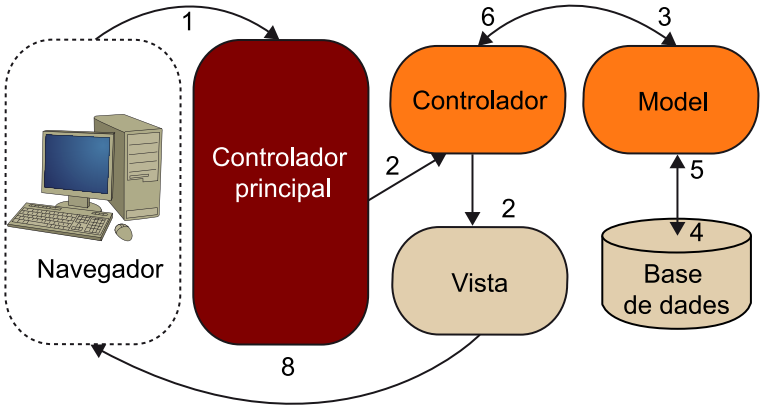
c) Controlador: rep les peticions provinents de la interfície d'usuari i coordina les respostes. Cada petició es delega a un controlador que s'encarrega d'executar les accions requerides fins a generar una resposta.

La utilitat del controlador

Es disposa d'una aplicació web per a la venda de llibres. Si l'usuari fa clic sobre el títol d'un llibre per visualitzar-ne les característiques, el servidor rep la petició i delega la tasca al controlador encarregat d'aquesta funció. Aquest accedirà al model, obtindrà el llibre amb l'ISBN indicat en l'URL com a paràmetre i retornarà al navegador una vista que en mostrarà els detalls.

La figura 4 mostra els tres tipus de components i la relació entre aquests des del punt de vista d'una acció, enumerant cada pas.

Figura 4. Arquitectura MVC



Les accions

Les accions també poden ser generades a partir d'un procés periòdic sense la intervenció de l'usuari.

1) L'usuari fa una acció sobre la interfície d'usuari. Per exemple, modificar un camp o sol·licitar informació que ha de ser visualitzada.

2) El controlador principal pot delegar en un d'específic.

3) El controlador accedeix al model per a efectuar les consultes o modificacions (en cas que sigui necessari).

4) El model consulta la base de dades en cas que vulgui obtenir valors.

5) La base de dades retorna els valors al model.

6) El model retorna el control al controlador anterior.

7) El controlador selecciona la vista encarregada de visualitzar la resposta i passa les dades que ha de visualitzar obtingudes pel model.

8) La vista s'envia de tornada a l'usuari com a resposta a la petició.

Els passos anteriors es reprodueixen en cadascuna de les peticions del navegador de l'usuari, de manera que el controlador adequat rep la petició, la processa (sol·licitant dades per mitjà del model si fos necessari), i quan la té finalitzada, crida la vista adequada perquè mostri la resposta a l'usuari final.

1.3.3. Estructura de carpetes i fitxers

Seguint la filosofia de "convenció enfront de configuració", un dels elements que està estandarditzat en el Rails és l'estructura de carpetes i fitxers d'un projecte. La creació d'un projecte Ruby on Rails crea per defecte una estructura de carpetes i fitxers.

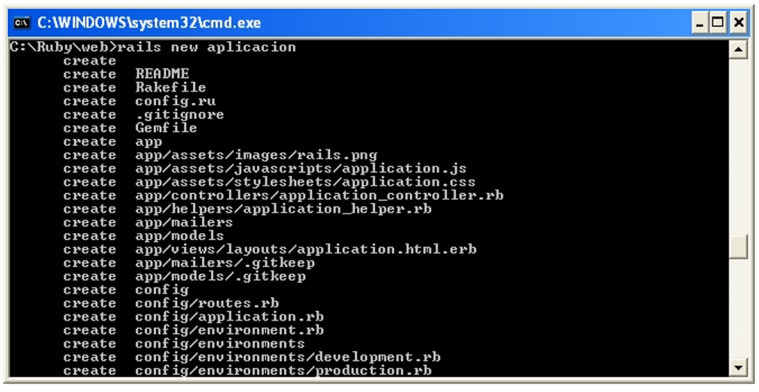
Per a què serveix cadascun? A continuació s'explica breument l'objectiu de cadascun dels elements d'aquesta estructura.

Per començar, és necessari crear un projecte perquè es generi l'estructura. "Per conveni", es crea un directori `web` a l'interior del directori d'instal·lació del Ruby. Creat aquest directori, s'obrirà la consola d'ordres del Windows, se situarà en el directori creat recentment, i s'executarà l'ordre següent:

```
rails new aplicacio
```

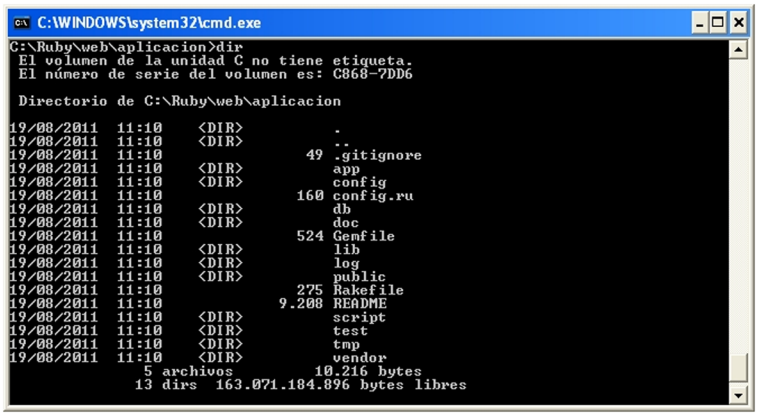
La sentència anterior crearà l'estructura de carpetes i fitxers del nou projecte creat (figura 5).

Figura 5. Creació d'un nou projecte Ruby on Rails



S'ha creat un nou directori `aplicacio`, i el seu contingut s'observa en la figura 6.

Figura 6. Estructura de directoris d'un projecte



En primer lloc apareixen un conjunt de fitxers en el directori arrel de l'aplicació. Es tracta de fitxers que el programador no haurà de modificar ni utilitzar. A continuació es fa una petita descripció dels fitxers principals que compleixen aquesta característica:

- `config.ru`: és el fitxer que conté la configuració per a arrencar l'aplicació en servidors web.
- `Gemfile`: aquest fitxer especifica les dependències que té l'aplicació amb perifèrics[10] externs.
- `Rakefile`: és el fitxer on es defineixen les tasques per a l'execució de tests, creació de documentació, generació de la base de dades, etc.
- `README`: és el fitxer clàssic de presentació de l'entorn de programació.

(10)En anglès, *plug-ins*.

En segon lloc apareixen un conjunt de directoris/subdirectoris, el contingut dels quals es descriu breument a continuació:

a) `app`: es tracta del directori més important, s'hi emmagatzema el codi font de l'aplicació. Aquest disposa alhora de la seva estructura de subdirectoris pròpia:

- **`views`**: on hi haurà els fitxers de vistes del projecte.
- **`controlers`**: on es desen els fitxers dels diferents controladors de l'aplicació.
- **`models`**: on s'emmagatzemen els fitxers de cadascun dels models de l'aplicació.

- **`mailers`**, **`assets`** i **`helpers`**: que emmagatzemen altres fitxers que formen part de l'aplicació.

b) **`config`**: en aquest directori el Rails emmagatzema els fitxers que defineixen la configuració per defecte (o per convenció). En aquest mòdul no serà necessari modificar cap configuració estàndard.

c) **`db`**: en aquest directori s'emmagatzemen els fitxers de creació de l'esquema de la base de dades i a més en cada modificació o migració de la base de dades es crea un fitxer que es desa en aquest directori.

d) **`doc`**: una característica interessant del Rails és que es tracta d'una eina que és capaç de generar la documentació de les aplicacions de manera automàtica i senzilla. En aquest directori es desa aquesta documentació.

e) **`lib`**: en la majoria d'aplicacions s'utilitzen biblioteques externes que fan tasques específiques i complexes que eviten que el programador les hagi d'implementar des de zero. Aquestes biblioteques es desen en aquest directori.

f) **`log`**: es tracta d'un dels directoris més útils quan s'està en la fase de desenvolupament, ja que emmagatzema fitxers de registre de sessió[11] amb informació molt detallada sobre l'execució de l'aplicació.

[11] Fitxers *log*, segons la seva expressió anglesa.

g) **`public`**: el servidor web utilitza aquest directori com la base de l'aplicació, ja que s'hi emmagatzemen les pàgines web estàtiques, fitxers CSS i JavaScript.

h) **`script`**: en aquest directori s'emmagatzemen els *scripts* definits pel programador, normalment són *scripts* que agrupen ordres que normalment s'utilitzen en tasques de manteniment.

i) **`test`**: una altra característica interessant del Rails és la capacitat de crear *tests* unitaris, funcionals i d'integració. Tots els fitxers que defineixen aquests *tests* s'emmagatzemen en aquest directori.

j) **`tmp`**: és el directori on s'emmagatzemen els fitxers temporals; hi apareixeran fitxers de continguts cau, de sessions d'usuari, de *sockets*, etc. En general aquest directori es buida automàticament, però és possible que alguna vegada sigui necessària l'eliminació manual de certs fitxers.

k) **`vendor`**: és el lloc on s'emmagatzemen els fitxers de control dels perifèrics que utilitza l'aplicació.

L'estructura anterior no s'ha de modificar i és comuna en totes les aplicacions Ruby on Rails. Gràcies a aquesta característica la localització d'un fitxer és molt senzilla navegant per l'estructura anterior.

El detall dels fitxers i subcarpetes s'anirà presentant al llarg del mòdul a mesura que sigui necessari, però aquest apartat ja en proporciona una primera aproximació.

1.4. La primera aplicació "Hola Món"

En tots els manuals de programació hi ha un primer apartat en el qual es crea una aplicació "Hola Món". L'objectiu de l'apartat és fer que el programador creï la seva primera aplicació amb unes simples línies de codi.

En el Ruby on Rails una aplicació simple com "Hola Món" crea la mateixa estructura que una aplicació amb més complexitat, però l'interessant és que la major part del treball haurà estat fet pel Rails, i no pel programador.

1.4.1. Creació de l'aplicació

El primer pas és la creació de l'aplicació. Per a això s'executa la sentència següent:

```
railsnew hola
```

En aquest moment s'ha creat l'estructura de directoris i de fitxers, però en aquesta primera aplicació s'ignoraran tots excepte alguns fitxers del directori `app`.

Però l'aplicació és web i, per tant, és necessari disposar d'un servidor web que sigui capaç d'executar-la. En el subapartat següent es prepara el servidor web.

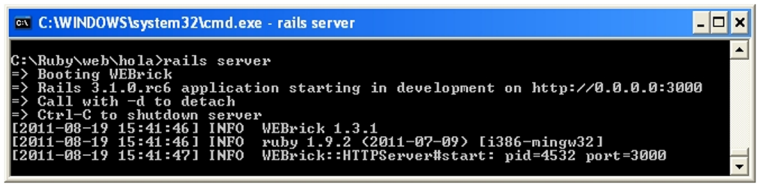
1.4.2. Arrencada del servidor web

La instal·lació feta del Ruby on Rails porta incorporat per defecte el servidor web WEBrick. L'arrencada del servidor es fa amb l'ordre `rails server`, executada en el directori creat recentment (`ruby/web/hola`):

```
rails server
```

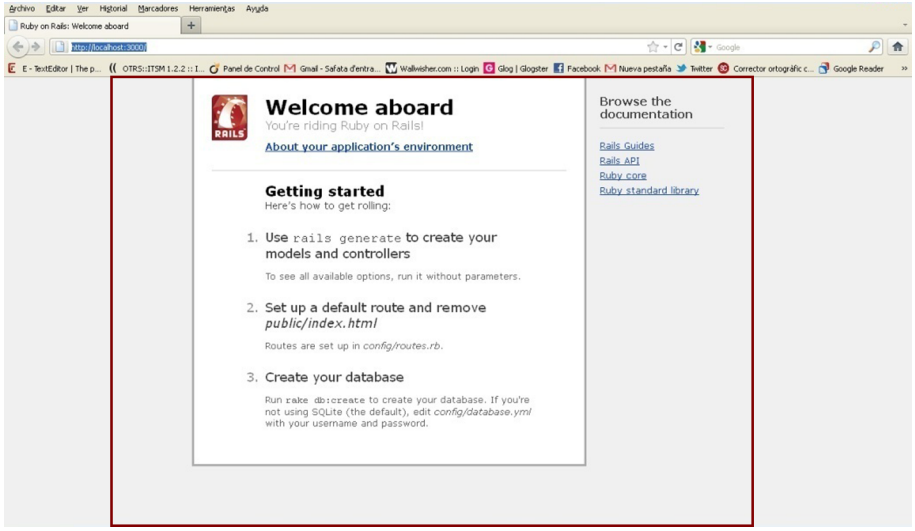
En executar l'ordre apareix el text següent en la consola:

Figura 7. Arrencada del servidor WEBrick



El servidor web ja està en marxa. És possible comprovar que funciona correctament obrint el navegador i indicant l'adreça web següent: `http://localhost:3000`. En la finestra del navegador ha d'aparèixer la pàgina que es mostra en la figura 8:

Figura 8. Pàgina d'inici d'un projecte Ruby on Rails buida



En aquests moments ja es disposa del servidor web a l'escolta en el port 3000 (opció per defecte del Ruby on Rails); de la mateixa manera, la pàgina inicial que apareix en el navegador és la pàgina inicial de l'aplicació `hola`.

Però l'objectiu d'aquest subapartat és mostrar el missatge "Hola Mundo!". Per a això s'ha de crear un controlador, que serà l'encarregat de processar la petició del navegador i retornar la vista que contindrà el missatge "Hola Mundo!".

1.4.3. Creació del controlador i de la vista

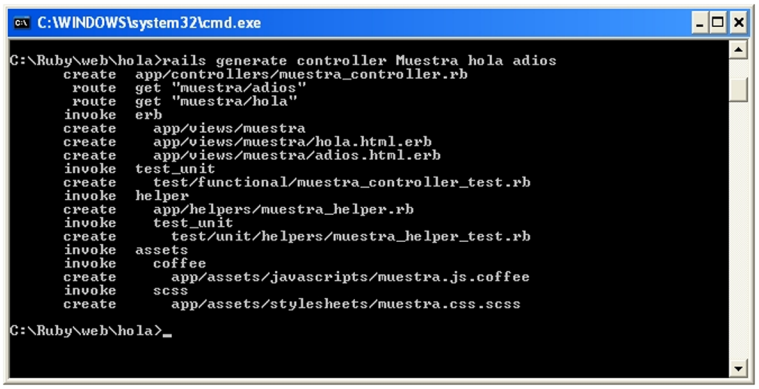
En aquest subapartat es tocaran aspectes que donen una lleugera idea del potencial del Ruby on Rails. Per començar, l'*script* `generate` crea el codi base del controlador i de cadascuna de les seves accions. Aquest *script* ha d'anar seguit del nom del controlador i de les accions que aquest necessitarà.

Com que la finestra de comandaments està ocupada executant el servidor web, és necessari obrir una finestra d'ordres nova per a poder executar la instrucció en el directori de l'aplicació (`ruby/web/hola/`):

```
rails generate controller muestra hola adios
```

En la finestra de comandaments apareixerà el següent (figura 9):

Figura 9. Creació del controlador Muestra i els seus mètodes hola/adios



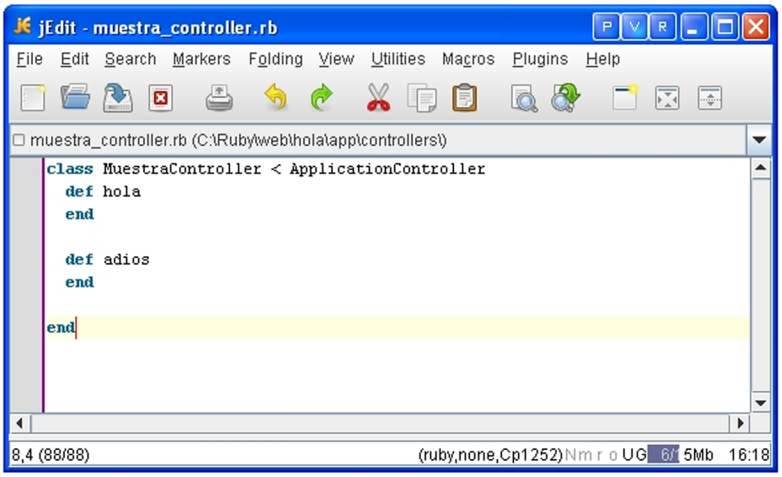
S'ha creat el controlador `Muestra`, s'ha creat una vista per al controlador i una per a cadascuna de les accions `hola` i `adios`. A més, es creen `tests`, `helpers` i `assets`, encara que aquests no s'estudiaran en aquest mòdul.

De tots els fitxers creats, a continuació (figura 10) es mostra el codi del fitxer que defineix el controlador.

El codi és molt simple. S'observa que la classe `MuestraController` hereta de la classe `ApplicationController` i té definides dues accions (totalment buides en aquests moments).

La navegació en aplicacions Ruby on Rails es fa a partir de l'estructura de l'URL. S'hi ha d'indicar l'identificador del controlador seguit del símbol "`/`" i finalitza amb el nom de l'acció.

Figura 10. Codi del controlador Muestra i les seves accions hola/adios



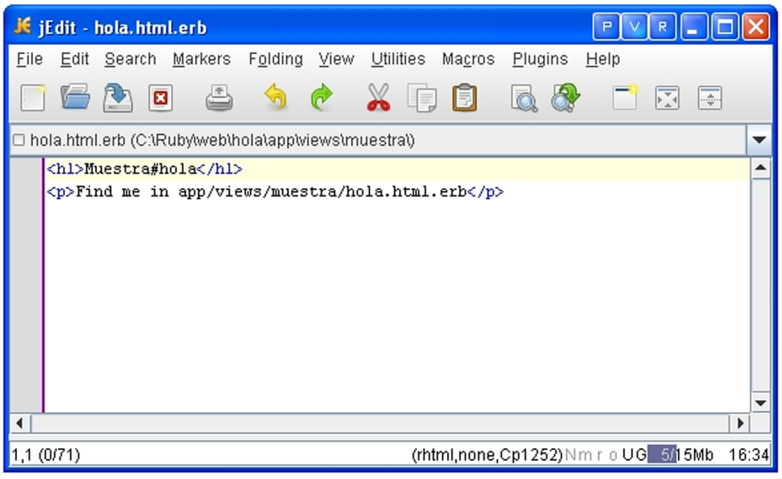
En escriure l'URL `http://localhost:3000/muestra/hola` s'obté la resposta següent en el navegador (figura 11):

Figura 11. Pàgina de la vista hola

Com ha ocorregut? És senzill: el navegador ha cridat l'acció `hola` del controlador `Muestra`. Aquesta acció no té cap codi definit, per la qual cosa no executa cap acció i retorna la vista definida per defecte (també creada a partir de l'*script* `rails generate controller`).

En la figura 12 es mostra el codi de la vista `hola.html.erb` que, tal com veiem a la pàgina que ens ha mostrat, la trobarem en el directori `app/views/muestra`.

Figura 12. Codi de la vista hola.html.erb

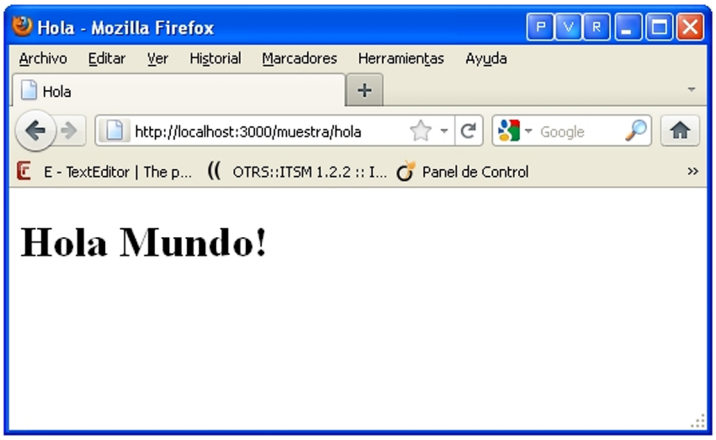


Com es pot observar en la figura 12, la vista conté les etiquetes HTML que es mostren en el navegador web. Per a aconseguir aquest objectiu, s'ha de substituir el codi anterior pel següent:

```
<h1> Hola Mundo! </h1>
```

Si s'actualitza la finestra del navegador, apareixerà el següent (figura 13):

Figura 13. Pàgina web de la vista hola.html.erb



Amb això s'ha obtingut el resultat esperat. Com cal esperar, es farà un pas més afegint codi Ruby com el que hem estudiat en el subapartat anterior a la vista.

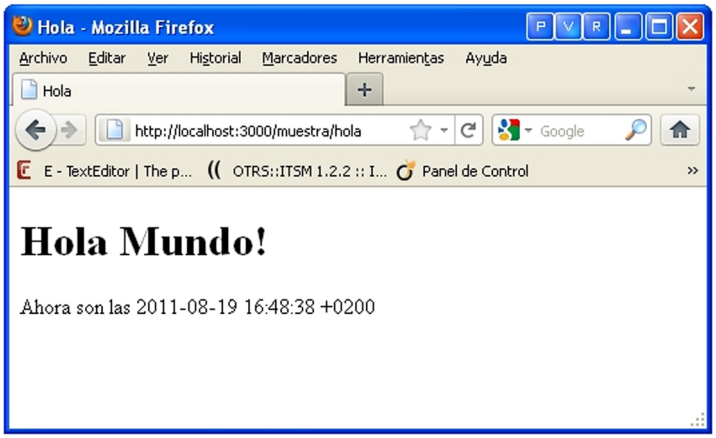
Per començar, s'afegirà a la vista anterior el codi següent:

```
<h1>Hola Mundo!</h1>
<p>
   Ahora son las %= <Time.now %>
</p>
```

Amb el codi anterior s'ha inserit codi Ruby en la vista que crida el mètode `now` de la classe `Time` (l'intèrpret detecta el codi, ja que està inserit entre els símbols "`<%`" i "`%>`").

Si s'actualitza el navegador, s'obté el resultat següent (figura 14):

Figura 14. Pàgina web actualitzada de la vista hola.html.erb

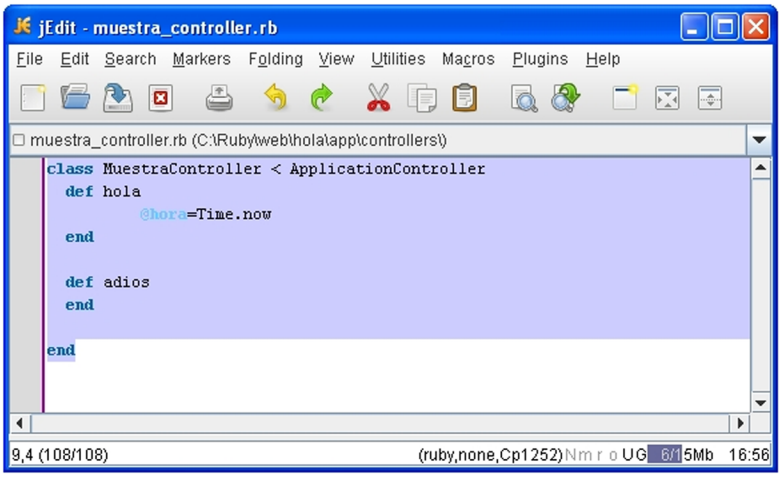


El codi anterior és correcte, però no segueix el patró MVC. Això és perquè tota la lògica de l'aplicació s'ha de programar en el controlador, deixant que la vista solament tingui la responsabilitat de mostrar informació. Per a això es modi-

ficarà el codi anterior introduint en el mètode `hola` una variable d'instància que conté el valor de l'hora en curs. Aquesta variable serà consultada per la vista i es mostrarà a la pàgina web.

Amb l'argument anterior, el codi del controlador quedarà de la manera següent (figura 15):

Figura 15. Codi del controlador Muestra



S'ha definit la variable d'instància *@hora*. A continuació és necessari modificar la vista perquè aquesta utilitzi el valor d'aquesta variable, substituint l'ús de la classe `Time` per la variable:

```
Ara són les %= <@hora %>
```

D'aquesta manera s'ha obtingut el mateix resultat, però s'ha aplicat el patró MVC de manera estricta.

2. Conceptes clau d'una aplicació Ruby on Rails

El patró MVC existeix en el Ruby on Rails gràcies a un conjunt de classes implementades en Ruby que faciliten el compliment del model. Per aquest motiu s'estudiarà sense entrar detalladament en cadascun dels components de Rails.

En la web de Ruby on Rails està disponible la documentació completa de l'API de Rails, on es pot consultar amb més detall cadascuna de les classes, els mètodes i les propietats que defineixen el conjunt del Rails.

2.1. `ActiveRecord`

Es tracta de la classe que implementa la part "Model" del patró MVC.

Les seves funcions principals són la implementació del mapatge entre els objectes de Ruby on Rails i les taules de les bases de dades relacionals (també conegut com a ORM); i en segon lloc proporciona els mètodes CRUD[12] de manera molt propera al llenguatge natural (sense utilitzar el llenguatge SQL).

(12)Sigla de l'expressió anglesa corresponent a *crear*, *llegir*, *modificar* i *eliminar*.

`ActiveRecord` evita que el programador de Ruby on Rails treballi directament amb la base de dades, amb les taules i les columnes. Evita que hagi d'utilitzar sentències complexes del llenguatge SQL, i això és perquè tot el treball l'implementa la classe mateixa de manera transparent.

En els subapartats següents es presenten els conceptes bàsics d'`ActiveRecord`, encara que a més de fer la funció d'enllaç entre la instància d'una classe i la fila de la taula on es troben les dades, també proporciona mètodes que permeten gestionar les relacions entre objectes, mètodes que permeten controlar el cicle de vida dels objectes i mètodes que gestionen transaccions (agrupacions de canvis en la base de dades que s'han de fer de manera conjunta, és a dir, tots els objectes s'actualitzen en una mateixa transacció o aquesta es cancel·la).

Reflexió

Alguns d'aquests punts sobrepassen l'abast d'aquest mòdul, per la qual cosa si se'n vol ampliar el coneixement es pot consultar la pàgina oficial de l'API, tal com s'ha plantejat en la introducció de l'apartat.

2.1.1. De la classe a la taula

El programador defineix un model amb els seus atributs o propietats i `ActiveRecord` transforma aquest model en una taula d'una base de dades relacional.

El Ruby on Rails estableix el conveni en el qual els models o classes es denoten en singular i sempre amb la primera lletra en majúscula, mentre que les taules associades es pluralitzen i canvien a minúscula la primera lletra.

Exemple

Si es defineix la classe `Menu`, es definirà la taula associada `menus`.

Cada instància de la classe (cada objecte) correspon a una fila de la taula i cada columna de la taula correspon a un atribut. A continuació es crearà la primera classe en un nou projecte que es dirà `Project` i que s'utilitzarà per a practicar els conceptes presentats al llarg de l'apartat.

La classe que volem crear serà `Equipo`, les propietats o atributs de la qual seran `nombre`, `estadio` i `historia`, de manera que els dos primers són del tipus cadena de text i l'última serà un camp amb més capacitat.

Per a crear la classe s'utilitzarà una bastida[13] , que no és més que un ajudant que a partir d'uns pocs paràmetres és capaç de crear les estructures i fitxers necessaris per a l'aplicació. Per a això crearem un nou projecte `laboratorio` sobre el qual farem totes les proves d'aquests subapartats:

(13)En anglès, *scaffold*.

```
rails new laboratorio
```

A continuació ens situem en la carpeta arrel del projecte creat recentment i executem la sentència següent:

```
rails generate scaffold Equipo nombre:string estadio:string historia:text
```

Si s'observa la sortida d'aquesta instrucció, aquesta invoca `activeRecord` i crea un conjunt de fitxers. El primer, creat en el directori `db/migrate`, té el contingut següent:

```
class CreateEquipos <ActiveRecord::Migration
   def self.up
      create_table :equipos do |t|
         t.string :nombre
         t.string :estadio
         t.text :historia

         t.timestamps
      end
   end

   def self.down
      drop_table :equipos
   end
```

```
end
```

El fitxer defineix la classe `CreateEquipos` com a subclasse de `Migration`, que és al seu torn una subclasse d'`ActiveRecord`. Aquest fitxer implementa el procés ORM, ja que es defineix la taula `equipos` amb les seves tres columnes que corresponen a la classe `Equipo`.

D'altra banda, s'ha creat el fitxer `equipo.rb` en la carpeta `app/models` que defineix la classe `Equipo`, encara que inicialment està buida de contingut:

```
class Equipo <ActiveRecord::Base

end
```

També s'ha creat el controlador `equipos_controller` i una vista `equipos`.

Reflexió

Els fitxers `equipos_controller` i `equipos` seran objecte d'estudi en els apartats següents.

El pas següent és l'execució de la sentència que efectua la migració en la consola de comandaments oberta:

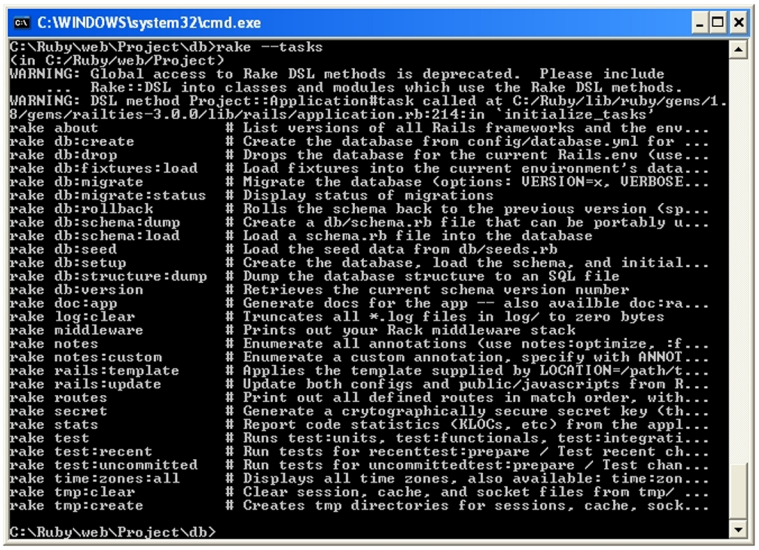
```
rake db:migrate
```

L'eina `Rake`

`Rake` és una utilitat de Ruby on Rails similar a la instrucció de Unix `make` i utilitza un fitxer `Rakefile` per a crear una llista de tasques. En el Rails, `Rake` s'utilitza per a implementar tasques d'administració d'una certa complexitat.

És possible obtenir una llista de les tasques `Rake` executant l'ordre `rake -tasks`, de manera que s'obté una llista amb la descripció de cadascuna de les tasques (però aquesta depèn del directori on s'executi la instrucció).

Figura 16. Sortida de la instrucció rake -tasks

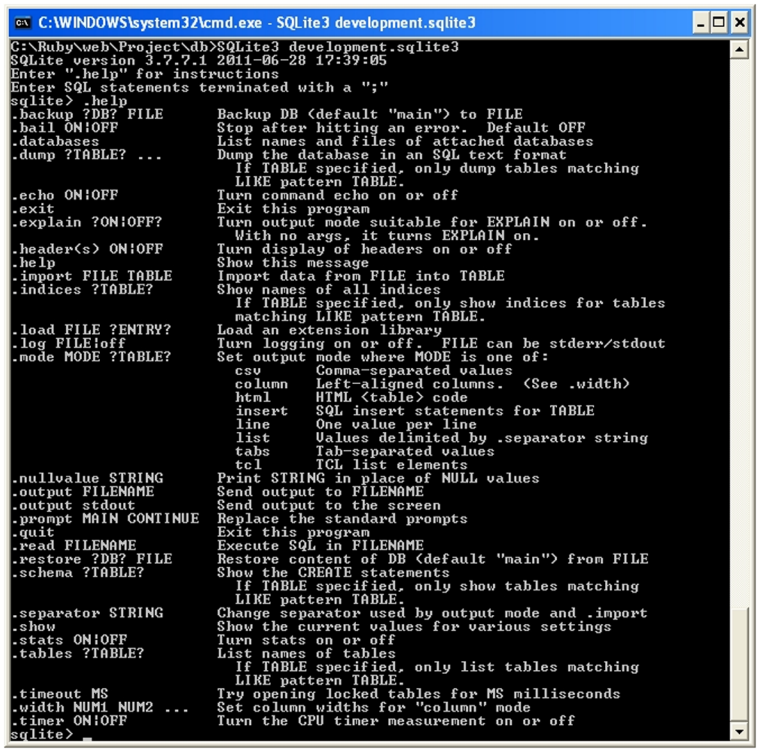


L'execució de la instrucció anterior ha creat una base de dades SQLite3 en el directori `db` del projecte, i el nom del fitxer és `development.sqlite3`. A més, ha s'ha creat una taula `Equipo` amb les columnes `nombre`, `estadio` i `historia`.

Es pot consultar la base de dades amb un parell d'ordres bàsiques d'SQLite3. Situant la consola en el directori `db`, en executar la instrucció següent, el gestor SQLite3 selecciona la base de dades creada recentment i dins del gestor es pot executar la instrucció `.help`, que mostra les diferents opcions disponibles (figura 17).

```
SQLite3 "development.sqlite3";
```

Figura 17. Opcions disponibles en la consola d'SQLite3



La instrucció `.tables` mostra el conjunt de taules que hi ha en la base de dades, de manera que en executar-la s'obté la taula `equipos`. D'altra banda, si s'executa la sentència `.schema equipos`, aquesta retorna la sentència SQL que es va executar per a crear la taula `equipos`.

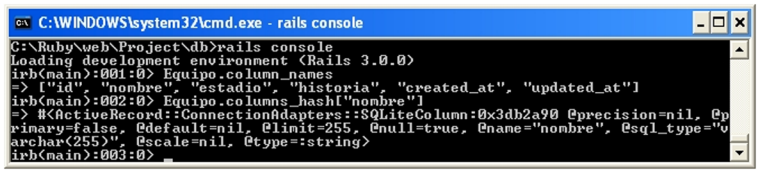
Fins aquest moment, amb dues simples sentències s'ha creat la classe `Equipo` amb les seves tres propietats (mitjançant l'ordre `rails generate scaffold`), a continuació s'ha creat la base de dades i la taula `equipos`, que emmagatzema les instàncies de la classe `Equipo` utilitzant la sentència `db:migrate`.

Una alternativa per a consultar els atributs creats és la consola de Rails, de manera que es consulten els atributs d'una classe i fins i tot les característiques d'aquests en la base de dades:

```
rails console
```

En el moment que s'obre la consola, l'ordre `Equipo.column_names` retorna un *array* amb els diferents camps de la taula. A més, l'ordre `Equipo.columns_hash["nombre"]` retorna la llista de les característiques que defineixen la columna `nombre` de la taula `equipos`.

Figura 18. Consulta de les classes amb la consola de Rails



Si s'observa la definició de la taula, aquesta disposa de tres columnes addicionals que han estat creades automàticament:

- `id`: es tracta d'una columna autonumèrica que és la clau principal de la taula.

- `created_at` i `updated_at`: són camps que emmagatzemen el moment de creació de la fila i el de l'última actualització.

2.1.2. Crear, modificar, llegir i esborrar

Tal com s'ha presentat en el subapartat anterior, una de les principals característiques que ofereix `ActiveRecord` és la possibilitat de manipular registres de la base de dades sense necessitat d'utilitzar sintaxi SQL. A continuació es presenten les diferents tècniques que fan possible implementar les accions de crear, modificar, llegir i esborrar registres d'una base de dades.

Crear

En primer lloc, la creació d'un registre en la taula és equivalent a l'acció de crear una instància de la classe; és a dir, en crear un objecte, es crea una fila de la base de dades. En la consola de Rails arrencada en el subapartat anterior s'introdueix el codi següent:

```
mi_equipo = Equipo.new
   mi_equipo.nombre = "Ondarense"
   mi_equipo.estadio = "Vicente Zaragoza"
   mi_equipo.historia = "Equipo de la población costera de Ondara, situada al norte de Alicante"
mi_equipo.save
```

Aquest codi provoca la creació d'un objecte `equipo`, que es pot consultar per mitjà de l'URL `localhost:3000/equipos` (recordem que s'ha d'arrencar el servidor web en el directori `laboratorio` per a poder executar la nova aplicació).

Figura 19. Creació d'instàncies d'una classe

En Ruby hi ha diferents alternatives sintàctiques per al codi anterior. En primer lloc, utilitzant un bloc de codi s'evita l'ús d'una variable:

```
Equipo.newdo |e|
   e.nombre = "Ondarense"
   e.estadio = "Vicente Zaragoza"
   e.historia = "Equipo de la población costera de Ondara, situada al norte de Alicante"
   e.save
end
```

Una alternativa vàlida és utilitzar un *array* associatiu, de manera que aquesta forma sintàctica és útil si s'emmagatzemen valors a partir d'un formulari HTML:

```
mi_equipo = Equipo.new(
   nombre => "Ondarense"
   estadio => "Vicente Zaragoza"
   historia => "Equipo de la población costera de Ondara, situada al norte de Alicante")
mi_equipo.save
```

Per finalitzar, Ruby disposa del mètode `create` per a crear nous objectes. La diferència amb `new` és que no necessita la crida al mètode `save` per a emmagatzemar els valors en la base de dades. Amb el mètode `create` els valors s'emmagatzemen de manera automàtica.

Llegir

La lectura de registres de la base de dades o d'una taula en concret es basa en la identificació dels registres que es volen obtenir. Per a això el Ruby on Rails disposa del mètode `find`. A continuació es revisen les diferents maneres en què es pot utilitzar aquest mètode.

Exemple

En l'exemple següent es busca el registre o l'objecte el valor del qual en la clau principal és el valor 1:

```
equipo_1 = Equipo.find(1)
```

Amb el codi anterior, la consola retorna l'assignació següent:

```
=> #<Equipo id: 1, nombre: "Ondarense", estadio: "Vicente Zaragoza",
historia: "Equipo de la población costera de Ondara, situada a...",
created_at: "2012-03-18 08:22:20", updated_at: "2012-03-18 08:22:20">
```

No és gens habitual buscar un registre a partir de la seva clau principal, ja que aquesta és creada pel sistema. Normalment les cerques es basen en els valors de certs atributs de l'objecte que l'usuari coneix.

Les cerques a partir de valors d'atributs s'implementen a partir del mètode `find_by_`*`atribut`*, en què *`atribut`* se substitueix pel nom d'aquest. S'obté el mateix resultat que en l'exemple anterior si s'utilitza la sintaxi següent:

```
equipo_Ondara = Equipo.find_by_nombre("Ondarense")
```

Aquesta sentència assigna a la variable `equipo_Ondara` el registre o instància de classe o objecte el nom del qual té el valor "Ondarense". A més, és possible afegir una nova clàusula mitjançant l'ordre `and` juntament amb la nova condició.

Exemple

La sentència següent emmagatzema en la variable `equipo_Ondara2` l'equip el nom del qual té el valor "Ondarense" i l'estadi del qual té el valor "Vicente Zaragoza":

```
equipo_Ondara2 = Equipo.find_by_nombre_and_estadio("Ondarense", "Vi-
cente Zaragoza")
```

Quan es busca un registre i aquest no existeix, el Ruby on Rails crea i retorna l'excepció `RecordNotFound`, encara que és possible modificar aquest comportament introduint el caràcter "`!`" entre la crida al mètode `find` i els parèntesis que contenen els paràmetres.

Exemple

La sentència següent provoca que s'assigni a la variable `equipo_5` el valor `Nil` si no hi ha cap registre la clau principal del qual és el valor 5:

```
equipo_5 =
Equipo.find!(5)
```

Per la seva banda, el mètode `find_last_by_` retorna l'últim dels registres o objectes si en la cerca se n'han trobat diversos.

Exemple

La sentència següent assigna l'últim dels equips el nom dels quals és "Ondarense" a la variable `equipo_u`:

```
equipo_u =
Equipo.find_last_by_nombre("Ondarens
```

Però si es volen obtenir tots els objectes que compleixen una condició s'utilitza la sentència `find_all_by_`.

Exemple

Si es crea una nova instància amb un nou estadi de l'equip "Ondarense", la sentència següent assignarà a `estadios` un *array* amb els objectes o instàncies el nom dels quals és "Ondarense".

```
estadios =
Equipo.find_all_by_nombre("Ondarense
```

Encara que el Ruby on Rails permet utilitzar sintaxi SQL en les cerques de registres, no és recomanable, ja que pot ser un focus d'errors en el codi. A més, amb l'ús de la sintaxi anterior s'assegura que el codi sigui funcional en tots els sistemes de bases de dades, independentment d'aquests.

Modificar

La modificació d'objectes es basa en la composició de les accions anteriors. En primer lloc es localitza l'objecte per modificar (utilitzant el mètode `find`), a continuació s'assigna el nou camp o atribut amb el nou valor i el procés finalitza amb el mètode `save`.

En l'exemple següent se selecciona l'objecte que en l'atribut `nombre` té el valor "Ondarense", es modifica el valor del camp `nombre` per "Valencia" i es desen els canvis:

```
equipoM = Equipo.find_by_nombre("Ondarense")
equipoM.nombre = "Valencia"
equipoM.save
```

També és possible fer modificacions utilitzant el mètode `update`, que actualitza un registre, i el mètode `update_all`, que actualitza tots els registres que compleixen una certa condició.

Exemple

Seguint amb l'exemple anterior, si el registre amb l'atribut `nombre` amb el valor "Ondarense" té el valor 1 en la clau principal, es podrà substituir per la sentència següent:

```
equipoM = Equipo.update(1, :nombre => "Barcelona")
```

En el cas que es vulguin substituir tots els objectes amb valor "Ondarense" en l'atribut `nombre` per "Barcelona" s'utilitzarà la sentència següent:

```
equipoM = Equipo.update_all("nombre = 'Barcelona'", "nombre like 'On-
darense'")
```

Eliminar

L'eliminació de registres o objectes es pot fer de dues maneres diferents:

- Amb els mètodes `delete` i `delete_all` es fa l'eliminació dels registres de la base de dades.

- Els mètodes `destroy` i `destroy_all`, executen el mètode `destroy` de l'objecte o dels objectes que es volen destruir.

L'avantatge de la segona sintaxi és que en la definició dels mètodes dels objectes es poden incorporar algunes accions o validacions que poden afegir una certa seguretat al procés d'eliminació (un dels més crítics en qualsevol aplicació).

Exemple

En l'exemple següent s'elimina l'objecte amb l'atribut `estadio` amb el valor "Tierra" i a continuació tots els objectes amb el nom "Barcelona":

```
equipoM = Equipo.find_by_estadio("Tierra")
equipoM.delete

Equipo.delete_all("nombre = 'Barcelona'")
```

Llavors s'han eliminat els dos registres que es tenien creats originalment amb el nom "Ondarense".

2.2. `ActionDispatch`

`ActionDispatch` és el primer d'un triplet d'elements que defineixen el que es coneix com a `ActionPack`. En els propers tres apartats s'estudien els tres components, l'objectiu dels quals es defineix breument a continuació:

- `ActionDispatch` és el responsable de fer arribar cada petició feta des del navegador de l'usuari al controlador adequat.

- `ActionController` rep les peticions, executa les accions i, quan finalitzen, retorna el control a la vista.

- `ActionView` rep el control des del controlador i aplica el format adequat a la resposta que és enviada de tornada a l'usuari.

Són responsables de la lògica del procés MVC, en defineixen els fluxos de treball. En els dos apartats següents s'estudiaran `ActionController` i `ActionView`, seguint d'aquesta manera la mateixa seqüència que es produeix en el flux de treball d'una aplicació Ruby on Rails.

2.2.1. Encaminament estàndard

En els subapartats anteriors s'ha creat una aplicació "Hola Món". A continuació s'ha aplicat la tècnica que ha fet possible executar cadascuna de les accions del controlador.

Vegeu també

Vegeu l'aplicació "Hola Món" en el subapartat 1.4, i les tècniques d'execució de les accions del controlador en el subapartat 1.4.3 d'aquest mòdul didàctic.

On es troben les instruccions que indiquen a `ActionDispatch` que executi el codi *X* quan es fa la crida *Y*? La resposta és senzilla: en el directori `config` del projecte hi ha el fitxer `routes.rb`, que conté les sentències següents, en què s'indica que el controlador `muestra` disposa de dues accions definides pel programador: `hola` i `adios`:

```
Hola::Application.routes.draw do
   resources :equipos
   get "muestra/hola"
   get "muestra/adios"
   # The priority is based upon order of creation:
   # first created -> highest priority.
   ####...
end
```

El codi anterior permet que l'usuari pugui executar cadascuna de les accions des del navegador a partir de la composició de l'URL indicada en el navegador (per a carregar les pàgines següents és necessari arrencar el servidor web en el directori `hola`):

```
http://localhost:3000/muestra/hola
http://localhost:3000/muestra/adios
```

El mecanisme anterior defineix les rutes de manera simple i comprensible, però aquestes poden ser modificades pel programador mateix.

2.2.2. Encaminament a mesura

L'encaminament es basa en la combinació dels mètodes HTTP (`GET`, `PUT`, `PUSH` i `DELETE`) amb els diferents noms que identifiquen els recursos de l'aplicació.

Utilitzant l'aplicació dels equips de futbol, s'analitzaran algunes de les possibilitats que ofereix el Ruby on Rails per a la personalització de l'encaminament. Si consultem el fitxer `routes.rb` del projecte `laboratorio` s'obté:

```
Laboratorio::Application.routes.draw do
   resources :equipos
   # The priority is based upon order of creation:
   ####...
end
```

El comportament de les rutes és el següent:

- Utilitzant `GET` i la ruta que apunta fins al controlador o recurs, s'obté una llista de tots els objectes d'aquest recurs (`http://localhost:3000/equipos`).

- Utilitzant `GET` i la ruta que apunta fins al controlador però indicant a continuació la clau principal d'un objecte, s'obté el contingut d'un dels equips (`http://localhost:3000/equipos/1`).

- Utilitzant `POST`, la ruta que apunta fins al controlador i les dades que defineixen un equip, es crea un equip nou amb els atributs indicats.

- Utilitzant `PUT`, la ruta que apunta fins al controlador i les dades que identifiquen i modifiquen l'objecte, aquest serà modificat amb els nous valors.

- Utilitzant `DELETE`, la ruta que apunta fins al controlador i la clau principal d'un objecte, aquest serà eliminat.

Conegudes les bases del protocol que permet la comunicació de les diferents accions, s'aprofundirà en les diferents opcions que apareixen en la pàgina que mostra tots els equips.

La diferència entre les rutes del projecte `hola` i `laboratorio` és que aquest últim disposa d'una única sentència:

```
resources :equipos
```

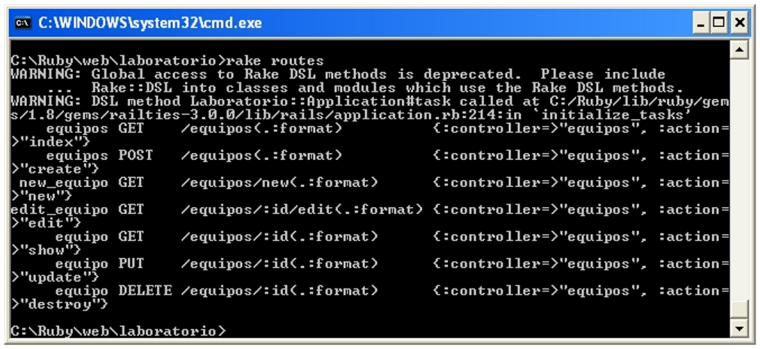
Aquesta sentència especifica que el controlador `equipos` disposa de set rutes estàndard, que es poden consultar amb l'ordre `rake routes` des del directori arrel de l'aplicació `laboratorio` (figura 20).

La resposta d'aquesta sentència és fàcil d'interpretar: mostra cadascuna de les rutes creades en l'aplicació. En cadascuna de les rutes s'identifica el controlador i l'acció que es du a terme. Tot això ha estat creat per la bastida, que va generar automàticament el codi en el controlador amb les set accions estàndard.

Vegeu també

Podeu veure la bastida que es va utilitzar en el subapartat 2.2.2 d'aquest mòdul didàctic.

Figura 20. Resultat de la instrucció rake routes



El nom de les accions estàndard identifica com és la seva funció, per la qual cosa són predictibles:

- `index`: mostra la llista amb els objectes existents.
- `create`: crea un nou objecte a partir de la informació en la crida `POST`.
- `new`: crea un objecte nou però no l'emmagatzema, el passa al client. Seria anàleg a crear un formulari buit perquè sigui emplenat.
- `show`: mostra el contingut d'un objecte identificat pel paràmetre `id`.
- `update`: actualitza el contingut de l'objecte identificat pel paràmetre `id`.
- `edit`: mostra el contingut de l'objecte identificat pel paràmetre `id` en un formulari llest per a ser editat per l'usuari.
- `destroy`: elimina l'objecte identificat pel paràmetre `id`.

Aquestes accions estàndard es poden modificar. Per exemple, si les dues últimes no fossin necessàries, es podria indicar en el fitxer `routes.rb` de la manera següent:

```
resources :equipos, :except => [:update, :destroy]
```

Conegut el mecanisme pel qual es fa l'encaminament de les accions d'un controlador concret, és imprescindible que aquestes accions es trobin definides en el controlador mateix.

En el fitxer `equipos_controller.rb` es troba el codi creat automàticament que defineix cadascuna de les set accions estàndard totalment funcionals:

```
class EquiposController < ApplicationController
   # GET /equipos
   # GET /equipos.xml
   def index
```

```
    @equipos = Equipo.all

    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @equipos }
    end
  end

  # GET /equipos/1
  # GET /equipos/1.xml
  def show
    @equipo = Equipo.find(params[:id])

    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @equipo }
    end
  end

  # GET /equipos/new
  # GET /equipos/new.xml
  def new
    @equipo = Equipo.new

    respond_to do |format|
      format.html # new.html.erb
      format.xml  { render :xml => @equipo }
    end
  end

  # GET /equipos/1/edit
  def edit
    @equipo = Equipo.find(params[:id])
  end

  # POST /equipos
  # POST /equipos.xml
  def create
    @equipo = Equipo.new(params[:equipo])

    respond_to do |format|
      if @equipo.save
        format.html { redirect_to(@equipo, :notice => 'Equipo was successfully
                      created.') }
        format.xml  { render :xml => @equipo, :status
                                  => :created, :location => @equipo }
      else
```

```
            format.html { render :action => "new" }
            format.xml  { render :xml => @equipo.errors, :status
                                    => :unprocessable_entity }
         end
      end
   end

   # PUT /equipos/1
   # PUT /equipos/1.xml
   def update
      @equipo = Equipo.find(params[:id])

      respond_to do |format|
         if @equipo.update_attributes(params[:equipo])
            format.html { redirect_to(@equipo, :notice => 'Equipo was successfully
                          updated.') }
            format.xml  { head :ok }
         else
            format.html { render :action => "edit" }
            format.xml  { render :xml => @equipo.errors, :status
                                    => :unprocessable_entity }
         end
      end
   end

   # DELETE /equipos/1
   # DELETE /equipos/1.xml
   def destroy
      @equipo = Equipo.find(params[:id])
      @equipo.destroy

      respond_to do |format|
         format.html { redirect_to(equipos_url) }
         format.xml  { head :ok }
      end
   end
end
```

En la majoria de les accions hi ha un bloc de codi `respond_to` en què, depenent de la variable indicada pel client, `format`, es donarà resposta en format HTML o XML, amb la qual cosa s'han definit dos tipus de resposta HTML i XML.

En la primera de les accions definides s'observa el següent:

```
def index
   @equipos = Equipo.all
```

```
    respond_to do |format|
       format.html # index.html.erb
       format.xml  { render :xml => @equipos }
    end
end
```

En la línia `@equipos = Equipo.all` s'assignen a la variable `equipos` tots els objectes, de manera que es converteix en un *array* amb cadascuna de les instàncies creades de la classe `Equipo`. A continuació, si el format de la resposta indicada en la variable `format` és HTML, es crida la vista `index.html.erb`, el contingut de la qual és el següent:

```
<h1>Listing equipos</h1>

<table>
   <tr>
      <th>Nombre</th>
      <th>Estadio</th>
      <th>Historia</th>
      <th></th>
      <th></th>
      <th></th>
   </tr>

   <% @equipos.eachdo |equipo| %>
   <tr>
      <td><%= equipo.nombre %></td>
      <td><%= equipo.estadio %></td>
      <td><%= equipo.historia %></td>
      <td><%= link_to 'Show', equipo %></td>
      <td><%= link_to 'Edit', edit_equipo_path(equipo) %></td>
      <td>
         <%= link_to 'Destroy', equipo, :confirm => 'Are you sure?',
                                        :method  => :delete %>
      </td>
   </tr>
   <% end %>
</table>

<br />

<%= link_to 'New Equipo', new_equipo_path %>
```

El codi crea una taula HTML i utilitza codi Ruby en la creació d'un bucle en el qual es va creant cada fila de la taula amb la informació de cada registre de la taula (o cada atribut dels objectes).

És possible afegir una nova acció utilitzant una extensió en la definició en el fitxer `routes.rb` amb les sentències següents, de manera que s'està indicant que hi haurà una nova acció `goles_a_favor`, en què el paràmetre `member` indica que l'acció s'aplica a cadascun dels objectes:

```
resources :equipos do
   get :goles_a_favor, :on => :member
end
```

Evidentment l'acció haurà d'estar definida en el controlador.

A més de l'opció anterior és possible definir una acció que s'apliqui a tots els equips en conjunt, i en aquest cas s'utilitzaria `collection` en la definició.

En els subapartats següents s'introdueix amb cert detall l'estructura dels controladors i de les vistes, i finalitza d'aquesta manera l'estudi dels tres components de l'`ActionPack`.

2.3. `ActionControler`

En aquest subapartat es presenta el comportament i la sintaxi que s'utilitza en els controladors. S'explica la relació d'aquests amb les vistes, com es processen plantilles, com s'envien fitxers des del servidor a l'usuari i com es pot encaminar a un URL des del controlador.

Reflexió

Per a aprofundir en la classe `ActionControler` heu de consultar l'API del Ruby on Rails.

2.3.1. Comportament del controlador

En primer lloc, quan el controlador rep una petició, prepara l'entorn d'execució posant a disposició tots els paràmetres de la crida i de l'entorn perquè puguin ser utilitzats en l'execució de l'acció.

Entre aquests paràmetres es troben els següents: el nom de l'acció i els paràmetres d'aquesta; les galetes, si n'hi ha; els detalls de la crida (mètode `http`, `ip` i port d'origen i característiques *ssl*); i les capçaleres HTTP.

En la majoria d'aplicacions solament s'utilitzen els paràmetres de la crida, però és important conèixer que hi ha la possibilitat de disposar de certa informació sobre la crida feta per l'usuari.

En segon lloc, quan un controlador rep una petició, es produeix el flux següent en l'execució:

- Es busca el mètode amb el mateix nom que el que rep el controlador; si aquest mètode existeix, se li passa el control; però si no existeix el mètode,

- es busca un mètode identificat com a `method_missing`; si es troba, s'executa, però en cas contrari,

- es busca una plantilla identificada amb el nom del controlador; si es troba, es retorna a l'usuari, però en cas contrari,

- el controlador retorna l'error `Unknown Action` a l'usuari.

En tercer lloc s'executarà el codi definit en l'acció definida i es generarà la resposta, que pot ser d'algun dels tipus següents:

a) En el 99% dels casos, la resposta és una crida a una plantilla o una vista que utilitza dades processades pel mètode o l'acció per a construir el resultat que s'envia a l'usuari.

Exemple

En l'exemple del final de l'apartat 1.4.3 d'aquest mòdul es fa una crida a la vista `hola.html.erb` que utilitza el valor de la variable calculada en el controlador.

b) En alguns casos, la notificació d'errors es fa amb la devolució d'una cadena de text simple al client.

c) Quan es programa amb el paradigma Ajax, algunes vegades una petició del client no disposa d'una resposta per part del navegador. Per tant, també és possible no retornar contingut al client; però com és imprescindible una resposta, es retornen un conjunt de capçaleres HTTP.

d) Hi ha escenaris en què s'ha de lliurar un fitxer al client. Seria l'exemple de fitxers PDF o JPG creats o emmagatzemats en el servidor web.

2.3.2. Processament de plantilles

Les plantilles són els fitxers que defineixen la resposta del servidor. El Ruby on Rails disposa de tres tipus de plantilles:

- erb, que està format per codi HTML i codi Ruby encastat.
- builder, que és un mecanisme amb el qual es creen documents XML.
- rjs, que s'encarrega de generar codi JavaScript.

Els noms de les plantilles són creats seguint l'estructura següent:

```
app/views/id_controlador/id_accion.tipo.xxx
```

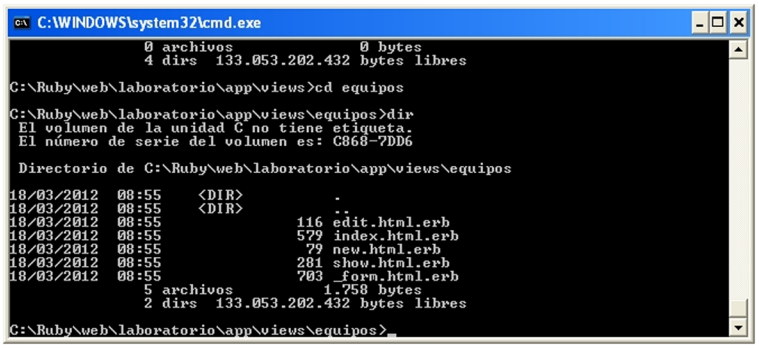
Aquesta estructura segueix realment el conveni següent estipulat pel Ruby on Rails:

- `id_controlador`: identifica el controlador.
- `id_accion`: identifica l'acció.

- `tipo`: identifica el tipus de fitxer final (HTML, JS o Atom)
- `xxx`: defineix el tipus de fitxer final (*erb*, *builder* o *rjs*).

En el cas del controlador `equipos` es tenen les plantilles següents:

Figura 21. Llista de les plantilles del controlador equipos



La crida a una vista o plantilla es fa utilitzant el mètode `render`, que disposa d'un conjunt de paràmetres opcionals. En general, el comportament d'aquest mètode és el següent:

a) `render()`: si no s'especifica cap paràmetre, el mètode `render` crida la vista del controlador assignada a l'acció en què s'executa la sentència `render`.

Exemple

En l'exemple següent, com no s'ha establert cap mètode sobre l'acció `index` que hi ha en la segona línia, s'interpreta que s'ha d'executar `render` sobre la vista amb el mateix nom que l'acció:

```
respond_to do |format|
   format.html # index.html.erb
   format.xml  { render :xml => @equipos }
```

És a dir, el Ruby on Rails considera que si no apareix cap mètode en la definició de l'acció, el mètode per defecte és `render` i l'executarà (encara que no es troba en l'acció). Per contra, en la tercera línia, on s'estipula l'acció si el format és XML, sí que es fa referència al mètode `render` perquè s'han d'introduir paràmetres, ja que la resposta no serà la vista estàndard.

b) `render(:text => string)`: en aquesta crida s'envia al navegador el text pla definit en la variable `string` sense fer cap tipus d'interpretació del tipus HTML.

Exemple

Si el codi de l'exemple anterior es modifica pel següent, quan s'executi l'acció `index` del controlador `Equipos` la resposta de l'usuari serà una pàgina web amb el text "`Respuesta texto simple`":

```
respond_to do |format|
   format.html { render :text => "Respuesta texto simple"}
   format.xml  { render :xml => @equipos }
```

c) `render(:inline => string, [ :type => "erb"|"builder"|"rjs" ], [ :locals => hash])`: en aquest tipus de crida s'indica que la cadena especificada en el paràmetre `inline` és el codi d'una plantilla o vista el tipus de la qual s'especifica mitjançant el paràmetre `type` i finalitza amb la indicació de la possibilitat d'utilitzar variables locals.

Exemple

L'exemple següent defineix un mètode per defecte d'un controlador. El comportament és que si el controlador és cridat amb un identificador d'una acció que no existeix, retorna una resposta a l'usuari amb el nom de l'acció i els seus paràmetres.

```
def metodoNo(name, *args)
render(:inline => %{<h2>Acción desconocida: #{name}</h2>Con los
parámetros siguientes: <br/><%= debug(params) %> })
end
```

d) `render(:action => nombre_accion)`: retorna a l'usuari la plantilla o vista per defecte de l'acció que és passada en el paràmetre.

e) `render(:file => path, [ :use_full_path => true|false ], [:locals => hash])`: retorna la plantilla que s'indica en la ruta especificada pel paràmetre `file`. Per defecte, es defineix una ruta absoluta.

f) `render(:template => name, [:locals => hash])`: retorna la plantilla identificada en el paràmetre `template` (que ha de contenir el nom del controlador i de l'acció).

Exemple

```
def index
render(:template => "equipos/nuev
end
```

g) `render(:partial => nombre, ...)`: retorna una plantilla parcial

Reflexió

Les plantilles parcials s'explicaran més endavant en aquest mateix mòdul.

h) `render(:nothing => true)`: retorna una pàgina HTML buida al navegador de l'usuari.

Com es pot observar, la sintaxi del mètode `render` és molt àmplia, però amb les opcions presentades es preveuen la majoria de les necessitats que sorgiran en una aplicació web de complexitat mitjana.

2.3.3. Enviament de fitxers

Hi ha dos mètodes que permeten l'enviament de fitxers des del servidor al client:

- `send_data`, que envia una cadena amb dades binàries.
- `send_file`, que envia un fitxer.

La sintaxi bàsica de cadascun d'aquests mètodes es presenta a continuació.

a) `send_data`

El mètode té la sintaxi següent, en què la variable *cadena* és el fitxer binari (per exemple, una imatge JPG o qualsevol tipus de fitxer binari):

```
send_data(cadena, opciones...)
```

Però, a més, hi ha una sèrie d'opcions que defineixen les característiques de la transmissió del fitxer:

- `:disposition` *inline/attachment*: quan l'opció seleccionada és `inline`, el navegador mostrarà el fitxer carregat, mentre que si és l'opció `attachment`, el navegador mostra la finestra de descàrrega del fitxer.

- `:filename` *nombre*: especifica el nom per defecte que el navegador donarà al fitxer quan aquest sigui emmagatzemat en l'equip client.

- `:type` *cadena*: defineix el tipus de fitxer, que per defecte és: `application/octet-stream`.

b) `send_file`

La sintaxi del mètode és la següent, en què la variable *ruta* indica la ruta del fitxer que es vol enviar:

```
send_file(ruta, opciones...)
```

I de la mateixa manera que el mètode anterior, `send_file` disposa de les següents opcions:

- `:buffer_size` *nombre*: s'hi indica la mida del fitxer enviat al navegador en cada escriptura si l'opció de *streaming* està habilitat.

- `:disposition` *inline/attachment*: quan l'opció és *inline*, el navegador mostrarà aquest fitxer; i si l'opció és *attachment*, indica que el navegador oferirà la descàrrega del fitxer.

- `:filename` *nombre*: especifica el nom per defecte que el navegador donarà al fitxer quan aquest sigui desat. Si no s'especifica, s'utilitzarà el mateix nom que té en la ruta original.

- `:stream` *true/false*: activa l'*streaming* en l'enviament de fitxers.

- `:type` *cadena*: defineix el tipus de fitxer, que per defecte és `application/octet-stream`.

Exemple

Si es modifica l'acció `index` del controlador `equipos` amb el codi següent, el navegador envia a l'usuari el fitxer `rails.png` que es troba en el directori `public/images` del projecte:

```
format.html {send_file("public/images/rails.png", :disposition =>
"attachment")}
```

2.3.4. Encaminament a un URL

En Ruby on Rails, l'encaminament s'utilitza quan l'acció sol·licitada pel client no pot ser processada i es necessita l'ajuda d'una altra acció. Per exemple, es pot donar el cas que per algun motiu en l'execució del codi d'una acció es necessiti executar una altra acció.

La sintaxi del mètode `redirect_to` és la següent:

- `redirect_to(:`*`action`*` => ..., `*`opciones`*`...)`: fa un encaminament a l'acció especificada en el paràmetre *`action`*.
- `redirect_to(`*`path`*`)`: fa l'adreçament a la ruta indicada com a paràmetre.
- `redirect_to(:back)`: fa un encaminament a l'URL indicat en la capçalera `HTTP_REFERER` de la crida.

Aquest mètode s'utilitza en la definició de les accions `create`, `update` i `destroy` del controlador `equipos`. Per exemple, en l'acció `create`:

```
def create
   @equipo = Equipo.new(params[:equipo])

   respond_to do |format|
      if @equipo.save
         format.html { redirect_to(@equipo, :notice => 'Equipo was successfully created.') }
         format.xml   { render :xml => @equipo, :status => :created, :location => @equipo }
      else
         format.html { render :action => "new" }
         format.xml   { render :xml => @equipo.errors, :status => :unprocessable_entity }
      end
   end
end
```

Després de crear l'equip en la primera línia de codi de l'acció:

```
@equipo = Equipo.new(params[:equipo]
```

es comprova que s'ha desat mitjançant el condicional:

```
if @equipo.save
```

i es fa l'encaminament:

```
format.html { redirect_to(@equipo, :notice => 'Equipo was
successfully created.') }
```

en què es crida l'acció `show`, que mostra l'equip creat recentment i a més el missatge: "Equip was successfully created.".

Es tracta d'una manera ràpida d'executar la sentència següent, però que a més d'implementar l'encaminament, afegeix la capçalera amb el missatge que indica que l'equip s'ha creat amb èxit:

```
format.html { redirect_to(:action => "shoe", :id =>
@equipo.id) }
```

2.4. `ActionView`

`ActionView` proporciona la funcionalitat necessària per a crear les plantilles que són transformades en pàgines HTML, fitxers XML o JavaScript i que s'envien al client com a resposta a l'acció sol·licitada. En aquest subapartat s'estudien algunes de les opcions disponibles en les plantilles, alguns ajudants de Rails amb els quals es creen formularis senzills, mecanismes per a pujar fitxers del client al servidor i l'ús de parcials que proporcionen la funcionalitat Ajax en el Ruby on Rails.

2.4.1. Plantilles eRB

Un dels convenis bàsics de Ruby on Rails és la localització dels fitxers en l'estructura de directoris. Les plantilles es troben en el directori: `app/views`, però a més, dins d'aquest directori es crea un directori per a cadascun dels controladors de l'aplicació. A l'interior d'aquests directoris cada plantilla és un fitxer erb amb l'especificació següent en el nom *`nombreaccion`*`.html.erb`.

En la figura 21 s'observa cadascuna de les plantilles de les accions del controlador `equipos` situades en `app/views/equipos`.

Les plantilles estan formades generalment per codi HTML i Ruby, encara que –com ja s'ha vist– hi ha la possibilitat de crear vistes amb contingut JavaScript o XML.

El codi Ruby es delimita utilitzant el delimitador següent `<%= ....%>`, de manera que es creen pàgines HTML amb sentències Ruby inserides en el seu interior i que són interpretades abans de ser enviades al client.

Una de les característiques importants de les plantilles és que el codi Ruby d'una vista té accés a l'entorn del controlador, és a dir:

- Pot accedir a totes les variables definides en el controlador. Aquest és el mecanisme principal de comunicació entre la vista i el controlador. En el controlador s'implementa la lògica de l'aplicació i el resultat s'emmagatzema en variables que són utilitzades per la vista per a compondre la pàgina de resultats.

- A més, l'objecte controlador és accessible des de la vista, la qual cosa permet executar des de la vista qualsevol dels mètodes públics definit en el controlador.

En l'exemple "Hola Món!" que hem presentat anteriorment, es va crear una vista amb codi Ruby inserit:

Vegeu també

Vegeu l'exemple "Hola Món!" en el subapartat 1.4.3.

```
<h1>Hola Mundo!</h1>
<p>
   Ahora son las %= <Time.now %>
</p>
```

En el codi Ruby inserit s'executa el mètode `now` de la classe `Time`, de manera que el resultat es converteix en una cadena i apareix a la pàgina HTML.

Es tracta d'un exemple senzill, però és possible inserir codi tan complex com sigui necessari. No obstant això, tal com es va comentar, és una bona pràctica imposar-se la lògica de negoci i utilitzar per a la implementació dels controladors i les vistes el mínim codi Ruby que permeti presentar els resultats.

En l'exemple anterior el símbol "=" és el responsable de la conversió a cadena i el retorn com a text del resultat del codi que té a continuació. Per això, si es vol executar codi Ruby que no retorna cap valor, simplement s'omet el símbol "=".

Si en la vista `hola.html.erb` de l'exemple "Hola Món" se substitueix el contingut pel següent, la primera secció de codi no retorna cap valor al client i és equivalent a fer l'assignació de la variable en el controlador, mentre que en la segona sí que retorna la data a l'usuari.

```
<% require 'date'
   @hora= Date.today
%>

<h1> Hola Mundo! </h1>
<p>
   Ahora son las %= <@hora %>
</p>
```

No obstant això, tal com s'ha comentat, és important que un codi com l'anterior es trobi en el controlador i no en la vista.

Una característica molt potent és la possibilitat d'inserir etiquetes HTML entre codi Ruby.

Exemple

El codi següent implementa un bucle de dos passos en el qual en el navegador s'escriu "Ei Ei".

```
<% 2.timesdo %>
Ei<br/>
<% end %>
```

Es pot crear un problema de seguretat a causa que el Ruby envia directament codi que és interpretat pel navegador, i com aquest interpreta codi JavaScript a més de les etiquetes HTML, es pot donar el cas que s'enviï al navegador codi JavaScript o etiquetes errònies que han estat introduïdes per un usuari intencionadament (per exemple, en un camp de text d'un formulari).

En el cas anterior, com que el contingut és enviat directament a la pàgina HTML, aquest seria interpretat pel navegador i podria crear certes accions que modifiquessin el comportament o els objectius inicials de l'aplicació.

Per a evitar aquest problema, s'utilitza el mètode `sanitize`, la traducció del qual al català, *desinfectar*, ja dóna unes pistes de quina és la seva funció: elimina etiquetes `<form>`, `<script>`, enllaços a codi JavaScript, etc.

Per tant, és recomanable desinfectar un resultat del codi Ruby si el contingut d'aquest ha estat generat inicialment per un client.

2.4.2. Formularis a partir d'ajudants

En totes les aplicacions web interactives, l'usuari o client en algun moment ha d'introduir informació en un formulari. El Ruby on Rails proporciona un conjunt de mètodes que faciliten la creació d'aquests formularis.

S'utilitzarà l'exemple següent per a mostrar el potencial dels ajudants de formularis.

Figura 22. Formulari d'exemple creat amb ajudants

El codi Ruby que construeix el formulari és el següent:

```
<h> Un formulario con ayudantes! </h>
<%= form_for :ejemplo do |form| %>

<p>
   <%= form.label :entrada %>
   <%= form.text_field :entrada, :placeholder => 'Introduce el texto aquí...' %>
</p>

<p>
   <%= form.label :direccion, :style => 'float: left' %>
   <%= form.text_area :direccion, :rows => 3, :cols => 40 %>
</p>

<p>
   <%= form.label :color %>:
   <%= form.radio_button :color, 'Rojo' %>
   <%= form.label :Rojo %>
   <%= form.radio_button :color, 'Naranja' %>
   <%= form.label :Naranja %>
   <%= form.radio_button :color, 'Verde' %>
   <%= form.label :Verde %>
</p>

<p>
   <%= form.label 'Acompañamiento' %>:
   <%= form.check_box :ketchup %>
   <%= form.label :ketchup %>
   <%= form.check_box :aceite %>
   <%= form.label :aceite %>
   <%= form.check_box :yogur %>
   <%= form.label :yogur %>
</p>

<p>
   <%= form.label :Prioridad %>:
   <%= form.select :Prioridad, (1..10) %>
</p>

<p>
   <%= form.label :Inicio %>:
   <%= form.date_select :Inicio %>
</p>

<p>
  <%= form.label :Alarma %>:
  <%= form.time_select :Alarma %>
</p>
```

```
<% end %>
```

En el codi anterior apareixen un conjunt de mètodes que es coneixen com a *ajudants* i la funció dels quals és simplificar la creació d'un formulari a partir de l'estandardització. La descripció bàsica de la funció s'explica a continuació:

- En la segona línia `form_for` crea el formulari i l'assigna a la variable `form`, que és utilitzada en la resta del codi.

- En l'exemple, les etiquetes es defineixen amb l'ajudant `label`, que s'acompanya de la cadena que apareixerà en l'etiqueta.

- Els camps de text es defineixen amb els ajudants `text_field` i amb `text_area` en cas de tractar-se d'un camp de text amb múltiples línies. A més, l'atribut `placeholder` defineix el text que apareixerà per defecte a l'interior del camp de text.

- Els camps tipus `radio` (botons d'opció) i `check` (caselles de selecció) es creen amb els ajudants `check_box` i `radio_button`.

- És interessant observar com s'ha creat la llista desplegable `Prioridad`, per la senzillesa amb la qual s'especifiquen els valors d'aquesta.

- Mitjançant l'ajudant `date_select` es creen tres llistes que permeten triar el dia, el mes i l'any. De manera anàloga, l'ajudant `time_select` crea dos desplegables que recopilen l'hora i els minuts per a assignar l'alarma.

Hi ha altres ajudants per a la construcció de formularis que no s'han utilitzat en l'exemple anterior, com `search_field`, `telephone_field`, `url_field`, `email_field`, `number_field` i `range_field`, i les seves característiques es poden experimentar de manera molt senzilla.

Possiblement, la característica principal que aporten els ajudants és que proporcionen una manera simple d'implementar programació *cross-platform*, i el fet que ens assegurem que es construeix un formulari que funcionarà en totes les plataformes i que aprofitarà els avantatges de cadascuna d'aquestes.

Com i quan s'utilitzen els formularis? Què ocorre amb les dades introduïdes en el formulari? A continuació es presenta un exemple de flux d'execució en el qual es produeix la modificació d'un objecte o instància de classe:

- El controlador rep una petició per a editar un objecte i amb la informació rebuda busca la instància de la classe que representa l'objecte.

- Localitzat l'objecte, el controlador crida una vista formada per un formulari els camps del qual contenen els valors de l'objecte que es vol modificar. Es tracta de la clàssica `edit`.

- Quan l'usuari modifica els camps i prem el botó d'enviar, els valors del formulari són enviats de nou al controlador a la recerca de l'acció que desarà els nous valors. El controlador rep els valors, els extreu i els assigna a l'*array* `params`.

- L'acció encarregada de desar l'objecte utilitza els paràmetres de `params` per a modificar i desarà els atributs nous.

S'ha de tenir en compte que les dades enviades en el mètode `POST` són transferides a l'*array* `params`, però si el nom del paràmetre té claudàtors, s'assumeix que aquest forma part d'una estructura complexa i es construeix un nou *array* per a emmagatzemar aquest paràmetre.

Per exemple, si el formulari té el camp `:id = 123`, l'*array* `params` tindrà el contingut següent: `{ :id => "123" }`, i en cas que es tingui el camp `[nombre] = Mestalla`, l'*array* `params` tindrà el contingut següent: `{ :campo => { :nombre => "Mestalla" }}`.

En l'exemple del controlador `equipos`, es disposa de l'acció `edit`, en què la vista `edit.html.erb` fa referència a un parcial en el qual s'observa la creació del formulari utilitzant ajudants, de manera que el formulari de l'exemple és únic i és utilitzat per les vistes `new` i `edit`, ja que en l'interior d'aquestes s'utilitza el mètode `render` per a inserir el formulari definit com una vista parcial:

Vegeu també

Els parcials s'estudien en el subapartat 2.4.3 d'aquest mòdul didàctic.

```
<div class="field">
   <%= f.label :nombre %><br />
   <%= f.text_field :nombre %>
</div>
<div class="field">
   <%= f.label :estadio %><br />
   <%= f.text_field :estadio %>
</div>
<div class="field">
   <%= f.label :historia %><br />
   <%= f.text_area :historia %>
</div>
```

El codi següent correspon a la vista `new.html.erb`:

```
<h1>New<equipo /h1>

<%= render 'form' %>
```

```
<%= link_to 'Back', equipos_path %>
```

El codi següent correspon a la vista `edit.html.erb`:

```
<h1>Editing equipo</h1>

<%= render 'form' %>

<%= link_to 'Show', @equipo %> |
<%= link_to 'Back', equipos_path %>
```

2.4.3. Ús de *layouts* i parcials

En qualsevol aplicació web de certa complexitat hi ha parts d'aquesta com menús o zones en què el codi es repeteix en diferents parts de l'aplicació. Però amb els conceptes presentats fins ara, cada vista haurà de contenir tot el codi que es presenta a l'usuari, per la qual cosa s'haurà de repetir tot el codi que defineix les capçaleres, els menús o la "cistella de la compra" i això complica el manteniment de l'aplicació (un petit canvi en el menú implicarà canviar totes les plantilles en què aquest s'hagi inserit). L'ús de *layouts* i parcials aporten un mecanisme per a resoldre aquest problema.

Exemples de codi que es repeteix

En la majoria d'aplicacions hi ha capçaleres o menús que es mantenen al llarg de l'aplicació. En el cas particular d'aplicacions de comerç electrònic, la "cistella de la compra" es repeteix en les diverses pàgines per les quals el client va navegant mentre fa les diferents transaccions.

Layouts

Els *layouts* són pàgines que contenen el disseny d'una pàgina HTML, és a dir, inclouen les seccions `html` i `css` que defineixen l'aspecte de la pàgina que serà enviada al client. Però en la secció `body` s'insereix la vista de l'acció que ha estat cridada pel controlador. El flux d'execució és el següent:

- Quan finalitza l'execució d'una acció, el controlador crida la vista associada.
- La vista es renderitza i es crida el *layout* associat, on la vista és inserida.
- Es retorna al client el *layout* en el cos del qual es troba la vista.

Aquest funcionament no és nou. Encara que no hi hem fet referència amb anterioritat, ja s'ha utilitzat, per exemple, quan s'ha creat la classe `Equipo` mitjançant la bastida.

Vegeu també

Vegeu la bastida que es va utilitzar en el subapartat 2.2.2 d'aquest mòdul didàctic.

En el directori `app/views/layouts` hi ha un fitxer `application.html.erb` que defineix el *layout* per defecte, el contingut del qual és el següent:

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

```
      <title>Laboratorio</title>
      <%= stylesheet_link_tag :all %>
      <%= javascript_include_tag :defaults %>
      <%= csrf_meta_tag %>
   </head>
   <body>

      <%= yield %>

   </body>
</html>
```

L'objectiu d'aquest *layout* és definir la pàgina HTML en el cos de la qual es mostra el resultat de cadascuna de les plantilles associades a les accions de la classe `Equipo` (`index`, `edit`, `show` i `new`). El mètode `yield` és l'encarregat de fer la càrrega de la plantilla de l'acció en el cos del *layout*.

Cada *layout* és específic d'un controlador, per la qual cosa un controlador busca en el directori `app/view/layouts` un *layout* amb el seu mateix nom. Però si el controlador no el troba, buscarà un *layout* identificat amb `application`, que és el *layout* que s'utilitza per defecte en tota l'aplicació.

El comportament descrit en el paràgraf anterior es pot modificar. Per a això es pot especificar en la definició del controlador el nom del *layout* que es vol utilitzar:

```
layout "standard"
```

Fins i tot és possible especificar un *layout* per a cadascuna de les accions del controlador. Per a fer aquesta especificació s'utilitza l'atribut `only`:

```
layout "standard" :only => [:show, :edit]
```

Com el *layout* defineix l'aparença de la resposta del controlador, és possible definir l'aparença depenent de certes condicions. En l'exemple següent, el *layout* aplicat depèn de si l'equip pertany a la primera o a la segona divisió:

```
classEquipoController <ApplicationController
layout :determina_layout
# ...
private
   def determina_layout
      ifEquipo.es_SegundaDivision?
         "equipo_segunda"
      else
         "equipo_primera"
      end
```

```
    end
end
```

Fins i tot cada acció pot triar el *layout* indicant-lo en la sentència `render`. En el primer exemple no es crida cap *layout* i en el segon un d'específic:

```
def salir
   render(:layout => false)
end

def resumen
   render(:layout => "layouts/resumen" )
end
```

Una característica fonamental dels *layouts* és que aquests tenen accés, igual que les plantilles, a les variables del controlador, i això permet implementar una personalització en certes parts de la pàgina com les capçaleres, els menús, etc.

Hi ha una alternativa més estructurada basada en la incorporació de cert contingut a la vista (que no es mostra, però que el *layout* sí que utilitza posteriorment). A continuació es mostra un exemple senzill d'aquesta tècnica:

```
<h1>Vista normal</h1>
<% content_for(:menu) do %>
   <ul>
      <li>Este texto está preparado y guardado para después</li>
      <li>Contiene partes<%= "dinámicas" %></li>
   </ul>
<% end %>
<p>
```

A continuació es planteja un *layout* que fa ús de la vista anterior:

```
<!DOCTYPE .... >
<html>
   <body>
      <div class="menu">
         <p>
         Menú con contenido dinámico:
         </p>
         <div class="menu-dinamico">
            <%= yield :menu %>
         </div>
      </div>
   </body>
</html>
```

La clau és la definició del segment de codi en la vista amb `content_for` i en la crida a aquest codi mitjançant la sentència `yield: menu` en *el layout*.

Parcials

Amb els parcials és possible la inserció de petits fragments de codi en vistes, la qual cosa permet funcionalitats com les comentades a l'inici d'aquest apartat (cistella de la compra, menú compartit, etc.).

Els **parcials** són vistes que són cridades des d'altres vistes, que reben com a paràmetre un objecte i que quan finalitzen retornen el control a la vista que l'ha cridat.

Els parcials s'identifiquen fàcilment perquè el seu nom comença amb el guió baix (_). Per exemple, en la classe `Equipo` es va crear un parcial `_form.html.erb` que es troba juntament amb la resta de plantilles en `app/views/equipos`, i el codi del qual és:

```
<%= form_for(@equipo) do |f| %>
   <% if @equipo.errors.any? %>
   <div id="error_explanation">
      <h2><%= pluralize(@equipo.errors.count, "error") %>
         prohibited this equipo from being saved:
      </h2>
      <ul>
         <% @equipo.errors.full_messages.eachdo |msg| %>
         <li><%= msg %></li>
         <% end %>
      </ul>
   </div>
   <% end %>

   <div class="field">
      <%= f.label :nombre %><br />
      <%= f.text_field :nombre %>
   </div>
   <div class="field">
      <%= f.label :estadio %><br />
      <%= f.text_field :estadio %>
   </div>
   <div class="field">
      <%= f.label :historia %><br />
      <%= f.text_area :historia %>
   </div>
   <div class="actions">
      <%= f.submit %>
```

```
    </div>
<% end %>
```

En el codi anterior es defineix un formulari per a la gestió dels atributs de la classe `Equipo`, utilitzant la sintaxi que ja hem estudiat. Aquest parcial és cridat des de la vista `edit` i `new` per mitjà de la sentència següent:

```
<%= render 'form' %>
```

En el cas que la crida es produeixi des d'un altre controlador (diferent del que conté el parcial), és necessari més detall en la sentència `render`, ja que s'ha d'especificar com és l'objecte en el qual hi ha el parcial:

```
<%= render(:partial => "form",:object=> @un_equipo) %>
```

De vegades, certes pàgines HTML estan formades per llistes d'elements (llista de la compra, llista d'equips de primera divisió, etc.), en què un parcial és encarregat d'aplicar format a cada element individual de la llista.

Per a implementar una vista a cadascun dels elements d'una col·lecció, s'utilitza el paràmetre `collection` juntament amb el paràmetre `:partial`.

Exemple

Per a mostrar una llista d'articles utilitzant el parcial `_articulo.html.erb` (on es defineix com es mostrarà cada article) s'utilitza la sintaxi següent:

```
<%= render(:partial=>"articulo",: collection=>@lista_articulos) %>
```

Es pot aplicar una certa separació entre els diferents articles, i es pot especificar amb el paràmetre `spacer_template` en la crida:

```
<%= render(:partial => "articulo", :collection => @lista_articulos), :spacer_template => "separador" %>
```

Amb la sentència anterior s'aplica en primer lloc el parcial `articulo` i a continuació s'aplica el parcial `separador`. Aquesta combinació es fa en cadascun dels articles de manera iterativa.

En les crides als parcials s'assumeix que aquests es troben en el mateix directori on hi ha les vistes del controlador, però de vegades no ha de ser així necessàriament. És el cas de parcials que són utilitzats per diferents controladors; en aquests casos es pot especificar la ruta on es troba el parcial tenint en compte que la ruta base és `app/views`.

D'altra banda, els parcials també poden utilitzar *layouts* que en defineixen el disseny. Per a implementar-ho, es defineixen utilitzant el paràmetre *layout* en la crida.

Per finalitzar l'apartat, un dels usos més interessants dels parcials és que els controladors poden utilitzar parcials directament amb l'objectiu d'actualitzar parts de codi de les pàgines (sense actualitzar tota la pàgina), i proporcionar d'aquesta manera la tècnica necessària per a implementar Ajax en Ruby on Rails.

2.5. Migracions

En els subapartats anteriors s'ha presentat l'estructura de cadascun dels components de les aplicacions Ruby on Rails. A més, s'ha vist com es poden modificar, de manera que a partir de l'estructura inicial creada amb la bastida es va personalitzant l'aplicació.

Però a mesura que es progressa en el desenvolupament de l'aplicació, l'esquema de la base de dades necessitarà evolucionar i adaptar-se a les particularitats que van sorgint. L'eina que proporciona el Ruby on Rails per a aquestes tasques és la migració.

Les **migracions** són fitxers Ruby que es troben en el directori `db/migrate`, el nom del qual és especial, ja que està format per la data i hora de creació del fitxer. Aquesta identificació defineix la versió del fitxer i la seqüència temporal en la qual s'apliquen les migracions. La identificació de l'interval temporal està seguida del nom de la classe sobre la qual s'aplicaran les modificacions.

Com cal esperar, la sentència `rails generate` va crear un fitxer `migration` que s'encarregava de crear la base de dades i la taula que emmagatzemava els valors de la classe `Equipo`. En el mateix apartat es cridava la sentència `rake :migrate`, que executava el fitxer anterior.

Vegeu també

Vegeu la sentència `rails generate` en el subapartat 2.1.1 d'aquest mòdul didàctic.

El fitxer de migració anterior té el nom següent:

```
20111001184529_create_equipos.rb
```

i el seu contingut és:

```
class CreateEquipos <ActiveRecord::Migration
   def self.up
      create_table :equipos do |t|
         t.string :nombre
         t.string :estadio
         t.text :historia

         t.timestamps
      end
```

```
    end

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

El subapartat següent presenta les característiques bàsiques que permeten la modificació de l'esquema de la base de dades d'una manera simple, i que a més permet adaptar-la a les necessitats que sorgeixen durant el desenvolupament de l'aplicació.

2.5.1. Creació i execució simple

Totes les bases de dades de les aplicacions Ruby on Rails disposen d'una taula en la qual s'especifica la versió de l'esquema aplicat. Aquest nombre de versió coincideix amb el valor que identifica la data i hora dels fitxers de migració.

El mecanisme anterior és la clau del flux d'execució d'una migració, ja que al llarg de la creació d'una aplicació es poden crear diferents fitxers de migració: on es creen taules, es modifiquen, es creen columnes, es modifiquen, s'eliminen, etc. Cadascun d'aquests fitxers de migració té un identificador únic i s'executen de manera seqüencial seguint l'ordre temporal definit en la data que forma part del nom del fitxer.

Amb la lògica anterior, en executar la sentència `rake db :migrate` es consulta la versió de l'esquema que té la base de dades i a continuació s'executen tots els fitxers de migració amb identificador posterior a la versió en curs. És important destacar que es fa de manera seqüencial i ordenada per la data i hora indicada en el fitxer de migració.

Quan la migració finalitza, el nou nombre que identifica la nova versió de la base de dades s'emmagatzema en la taula *schema_migrations*.

Però si no es volen executar totes les migracions, o si es vol tornar a una versió anterior de l'esquema, es pot especificar la versió de la base de dades que s'aconseguirà:

```
rake db:migrate VERSION=20100301000009
```

La tornada a una versió anterior es fa a partir de l'execució del codi de l'acció `down` definida en cadascun dels fitxers de migració.

Exemple

En l'exemple següent es defineix una migració que afegeix el camp `telefono` a la taula `Equipos`:

```
classTelefonoToEquipo<ActiveRecord::Migration
```

```
    def self.up
      add_column :equipos, :telefono, :string
    end

    def self.down
      remove_column :equipos, :telefono
    end
  end
```

El Ruby on Rails és independent del sistema gestor de base de dades, per la qual cosa disposa d'uns tipus de dades pròpies i fa la traducció als tipus específics de cada base de dades de manera automàtica. Els tipus existents són: `binary`, `boolean`, `date`, `datetime`, `decimal`, `float`, `integer`, `string`, `text`, `time` i `timestamp`.

Cadascun dels tipus de dades anteriors disposa de les tres opcions següents:

- *`null`* `=> true/false`, de manera que amb el valor assignat *`true`* la columna creada no pot emmagatzemar valors nuls.

- *`limit => size`*, que defineix la mida del camp i és utilitzat majoritàriament per a definir la mida de les cadenes de text.

- *`default => valor`*, que especifica el valor per defecte de la columna en cas que no se n'especifiqui cap.

Els tipus numèrics disposen de dues opcions addicionals: `precision` i `scale`, que defineixen en primer lloc el nombre de dígits que s'emmagatzemaran, i en segon lloc, on se situarà la coma decimal en aquests dígits.

Les accions disponibles són les següents: crear, reanomenar i eliminar tant columnes com taules, i també és possible crear i eliminar índexs.

S'utilitza el mètode `rename_column` per a implementar els canvis de nom en les columnes.

Exemple

El codi següent defineix una migració que canvia el nom de la columna `pueblo` per `poblacion`. Com s'observa, hi ha dos mètodes: `up`, que aplica el canvi, i `down`, que desfà el canvi:

```
class RenamePuebloColumn <ActiveRecord::Migration
  def self.up
    rename_column :equipos, :pueblo, :poblacion
  end

  def self.down
    rename_column :equipos, :poblacion, :pueblo
  end
end
```

El mètode `change_column` s'utilitza per a canviar el tipus o certs atributs d'una columna. Aquest mètode s'utilitza de manera similar al mètode `add_column`.

Exemple

En l'exemple següent, la columna `nombre`, que és del tipus `integer`, és canviada al tipus `string`:

```
def self.up
   change_column :equipos, :nombr
end

def self.down
   change_column :equipos, :nombr
end
```

Els canvis de tipus de dades són complexos. Això és així perquè de vegades no totes les dades poden ser emmagatzemades en un altre tipus diferent (per exemple, una cadena com "Hola Món" no es pot emmagatzemar com un tipus `integer`).

Els mètodes `create_table` i `drop_table` són utilitzats per a la creació i eliminació de taules.

Exemple

En l'exemple següent s'utilitzen aquests dos mètodes:

```
class CreateEquiposCopas<ActiveRecord::Migration
   def self.up
      create_table :equipo_copas do |t|
         t.integer :equipo_id, :null => false
         t.text :descripcion

         t.timestamps
      end
   end

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

En aquest exemple es crea una nova taula `equipo_copas`, amb dos camps `equipo_id` i `descripcion`, però no s'ha definit la clau principal. Realment no és necessari, ja que el Ruby on Rails la defineix per defecte com el camp `id` com a autonumèric i a més el mètode `timestamps` crea els camps `created_at` i `updated_at`.

El mètode `create_table` disposa d'un conjunt de paràmetres opcionals:

- `force: true`, en cas d'haver-hi una taula amb el mateix nom, és eliminada.
- `temporary: true`, crea la taula en format temporal, és a dir, s'eliminarà quan l'aplicació finalitzi.

D'altra banda, el mètode `rename_table` permet canviar el nom de la taula.

Exemple

A continuació es mostra un exemple d'ús del mètode `rename_table`:

```
class RenameEquipoCopas <ActiveRe
   def self.up
      rename_table :equipo_copas,
   end

   def self.down
      rename_table :equipo_trofeo
   end
end
```

El canvi de nom en les taules pot provocar problemes, ja que pot trencar la relació ORM, és a dir, la relació entre les classes i les taules que s'han creat anteriorment.

Per finalitzar el subapartat, els mètodes `add_index` i `drop_index` proporcionen la funcionalitat que permet afegir índexs en una taula. Es creen índexs quan hi ha cerques de registres per un cert camp el rendiment del qual és adequat.

Exemple

En l'exemple següent s'afegeix un índex en el camp `nombre` de la taula `equipos`:

```
class AddCustomerNombreIndexToEqu
   def self.up
      add_index :equipos, :nombre
   end

   def self.down
      remove_index :equipos, :nom
   end
end
```

3. Creació d'una aplicació: Restaurante UOC

En aquest apartat es desenvoluparà una aplicació web en què es podran registrar les reserves d'un restaurant. L'aplicació serà accessible per a clients i empleats del restaurant, en què es podran donar d'alta, de baixa i es podran modificar reserves.

Els empleats del restaurant disposaran de vistes per a visualitzar les reserves previstes per als dies propers i planificar així el menjador amb anterioritat.

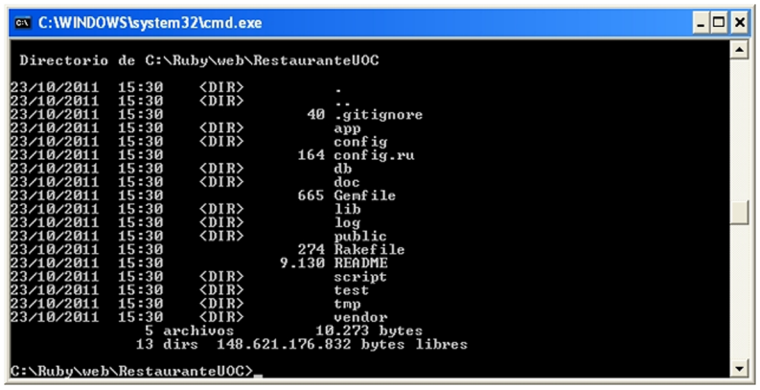
3.1. Creació de l'aplicació `RestauranteUOC`

En primer lloc es crearà un projecte anomenat *RestauranteUOC*. Per a fer-ho, ens situarem en la carpeta `C:\Ruby\web` de la consola de Windows i executarem la instrucció següent:

```
rails new RestauranteUOC
```

Perquè les següents sentències s'executin, és necessari situar la línia d'ordres en el directori `RestauranteUOC` (figura 23).

Figura 23. Estructura dels directoris del projecte

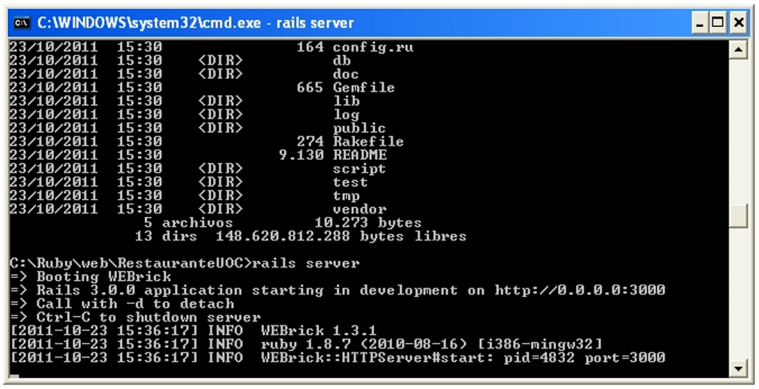


Arrencar el servidor web

En crear el projecte, es crea de manera automàtica un servidor que allotjarà l'aplicació.

Per a provar l'aplicació creada recentment, serà necessari iniciar un servidor que allotgi l'aplicació i permeti executar-la. Per a això s'executarà `rails server` des de la línia de comandaments en l'interior del directori creat recentment `RestauranteUOC` (figura 24).

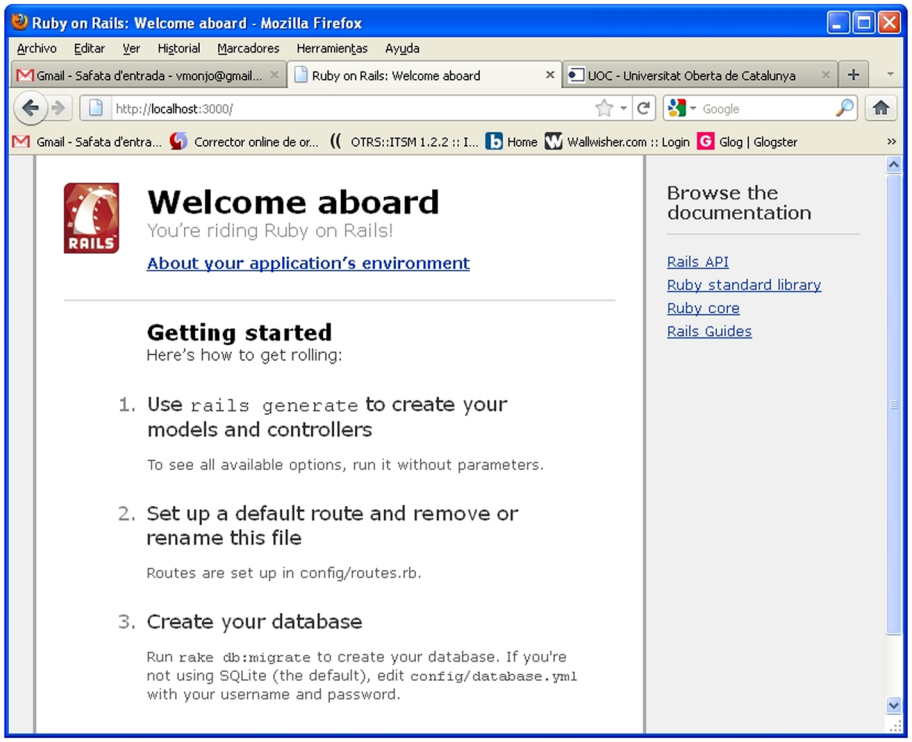
Figura 24. Arrencada del servidor web WEBrick



En executar l'aplicació en el sistema local, aquesta estarà accessible indicant l'URL `http://localhost:3000` en qualsevol navegador web.

La figura 25 mostra la pantalla de benvinguda del nou projecte.

Figura 25. Pantalla inicial del projecte creat recentment



El pas següent és la creació de la base de dades, i per a això s'utilitza la instrucció següent en el directori `RestauranteUOC`:

```
rake db:create
```

Amb aquesta instrucció ja s'ha creat l'estructura necessària per a construir l'aplicació, però encara no té cap contingut específic; aquest s'anirà ampliant en els subapartats següents.

3.2. Fase 1. Introducció de reserves

El requisit principal de l'aplicació és la gestió de reserves, que ha de disposar d'altes, baixes i modificacions. Una reserva disposarà dels camps següents:

Camp	Tipus	Descripció
Nom	*String*	Nom del client que fa la reserva.
Cognoms	*String*	Cognoms.
Telèfon	*String*	Telèfon de contacte.
Data	Data-Hora	Data i hora que vol disposar de la taula disponible.
Comensals	Numèric	Nombre de comensals.
Comentaris	*String* llarg	Comentaris addicionals.

Per a disposar de gestió de reserves en l'aplicació, s'hauran de crear els elements següents:

a) El **model de la reserva**: una classe que representi una reserva i permeti també fer diferents operacions com ara la cerca d'una reserva a partir d'uns criteris, l'actualització i eliminació de reserves, etc.

b) Les **vistes**: l'aplicació ha de tenir les vistes necessàries per a visualitzar la llista de reserves, el formulari d'alta d'una reserva, etc.

c) Els **controladors**: que s'encarregaran de gestionar les peticions fetes.

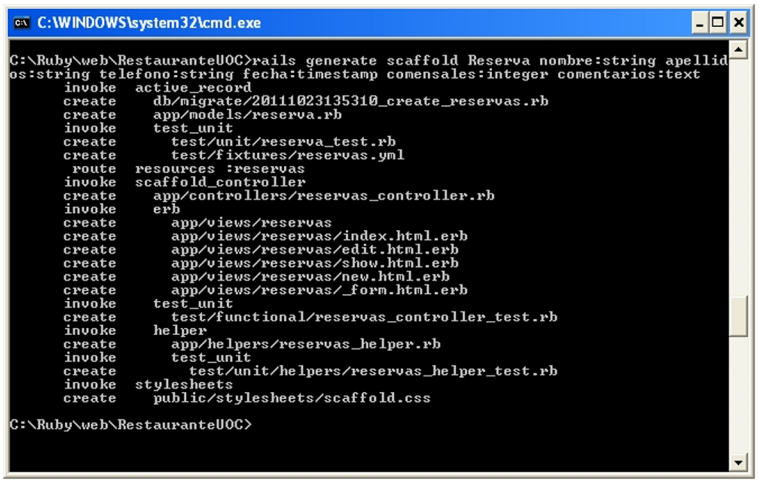
3.2.1. Creació de l'MVC

S'executarà el codi següent per a la creació mitjançant una bastida del model `Reserva`:

```
rails generate scaffold Reserva nombre: string apellidos:
string telefono: string fecha: timestamp comensales: integer
comentarios: text
```

S'obtindrà el resultat que s'observa a la figura 26. En aquests moments es disposa de la infraestructura necessària per a donar d'alta, baixa, modificar i eliminar reserves.

Figura 26. Creació del model mitjançant la bastida



a) El model `Reserva`

La bastida ha creat el model `Reserva` declarat en el fitxer `app/models/reserva.rb`. El contingut del fitxer és el següent:

```
class Reserva < ActiveRecord::Base
end
```

Aquestes dues línies defineixen la classe `Reserva`, que hereta d'`ActiveRecord`. Aquesta herència proveeix la funcionalitat necessària per a l'accés i la manipulació de reserves en la base de dades.

ActiveRecord

`ActiveRecord` proporciona el mapatge entre els objectes de la classe `Reserva` i la taula `reservas` de la base de dades.

Per convenció, es prendran els atributs de la classe a partir de les columnes de la taula `reservas`.

Per a crear i destruir aquesta taula, també s'ha creat la classe `CreateReservas` en el fitxer `db/migrate/create_reservas.rb`:

CreateReservas

`CreateReservas` conté el nom del model indicat en el generador `scaffold` i se li ha afegit el plural de manera automàtica per convenció.

```
class CreateReservas <ActiveRecord::Migration
   def self.up
      create_table :reservas do |t|
         t.string :nombre
         t.string :apellidos
         t.string :telefono
         t.timestamp :fecha
         t.integer :comensales
         t.text :comentarios
         t.timestamps
      end
   end
```

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

`CreateReservas` hereta d'`ActiveRecord::Migration` i defineix els mètodes tal com s'ha estudiat:

Vegeu també

Vegeu la definició de mètodes en l'apartat 2 d'aquest mòdul didàctic.

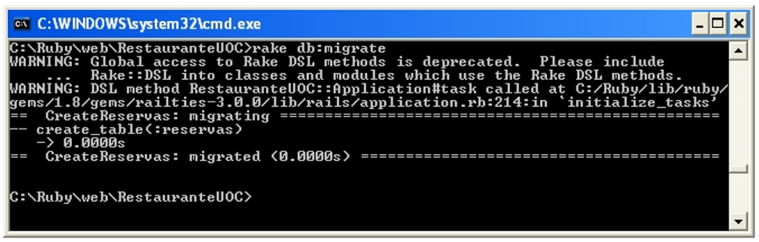
- `up`: que s'executa per a crear la taula que representa una reserva en la base de dades configurada en el projecte. Les columnes són les mateixes que es van especificar en els paràmetres de la bastida.

- `down`: elimina la taula `reservas` de la base de dades.

El pas següent és la creació de la taula `reservas`, que es fa utilitzant la instrucció següent:

```
rake db:migrate
```

Figura 27. Creació de la taula amb l'eina rake



Després de l'execució de la instrucció anterior es pot comprovar que s'ha creat la taula `reservas` amb les columnes següents:

Columna	Tipus
id	INT
nombre	VARCHAR(255)
apellidos	VARCHAR(255)
telefono	VARCHAR(255)
fecha	DATETIME
comensales	INT
comentarios	TEXT
created_at	DATETIME
updated_at	DATETIME

Per a això s'utilitzarà la consola de `rails`:

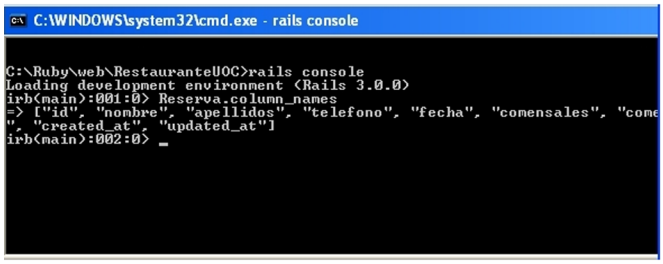
Vegeu també

Vegeu la consola de `rails` en el subapartat 2.1.1 d'aquest mòdul didàctic.

```
rails console
```

I dins de la consola es consulten les columnes de la taula `Reserva` amb `Reserva.column_names` (figura 28).

Figura 28. Creació de la taula amb l'eina rake



Es pot observar que s'han afegit automàticament columnes addicionals a les indicades en la generació: `id`, `created_at` i `updated_at`. En particular, `id` és un identificador únic, autoincremental i clau principal de la taula. Per la seva banda, `created_at` i `updated_at` contenen la data i hora de creació i de l'última actualització, respectivament.

Els atributs d'una reserva podrien estar declarats en totes dues classes: en el model `Reserva` i en la classe `ReservaMigration`; però això entraria en contradicció amb el principi "Don't repeat yourself". Per aquest motiu, inicialment solament es troben en la classe `ReservaMigration`.

Execució de l'aplicació

A continuació s'executarà l'aplicació per a comprovar els resultats de la generació del nou model, controlador i vistes. S'ha d'accedir a l'URL següent:

```
http://localhost:3000/reservas
```

La figura 29 mostra una llista de reserves que inicialment estarà buida, però l'enllaç `Newreserva` fa una crida a l'acció `new` del controlador `reservas`, de manera que aquest crida la vista `new`, que és enviada a l'usuari.

Aquesta vista té un parcial en el seu interior, que és el formulari, de manera que es carrega una pàgina que conté un formulari amb els camps `Nombre`, `Apellidos`, `Telefono`, `Fecha`, `Comensales` i `Comentarios`. En emplenar els camps i fer clic en el botó `Create`, es crida l'acció `create` del controlador, que desa els valors i redirigeix el resultat mitjançant la vista `show`.

Figura 29. Pàgina de llista de reserves

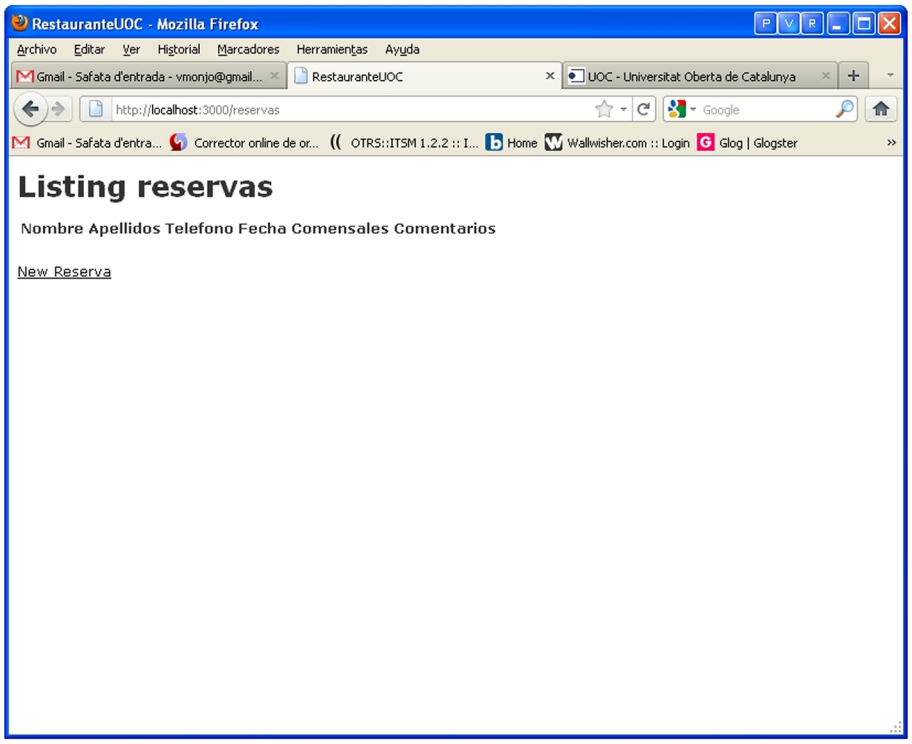
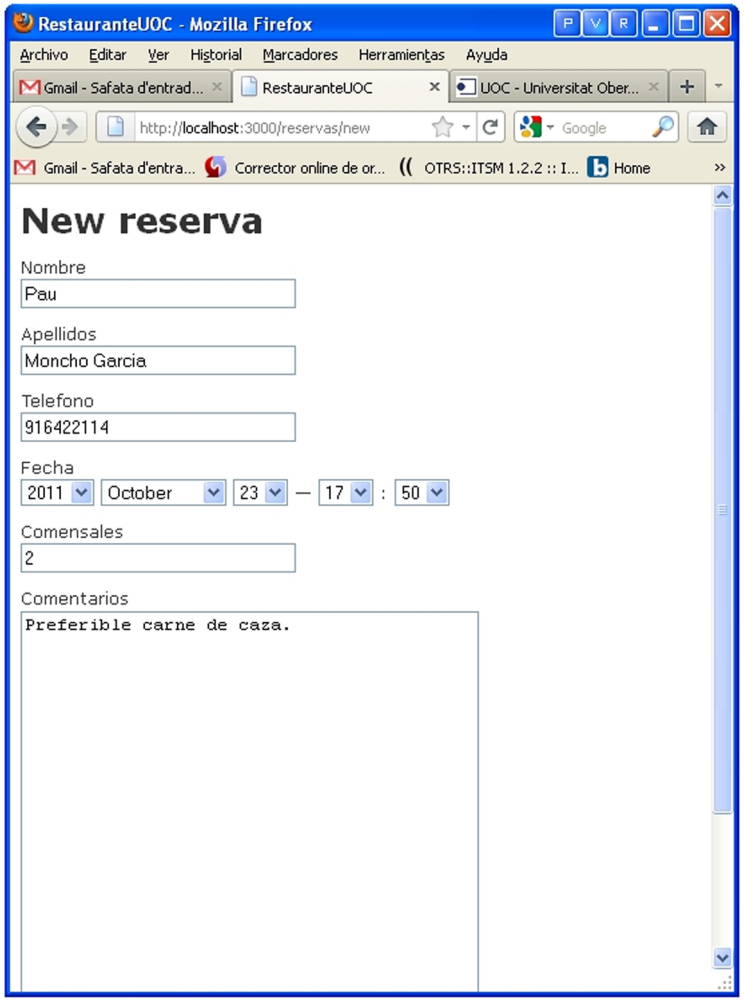
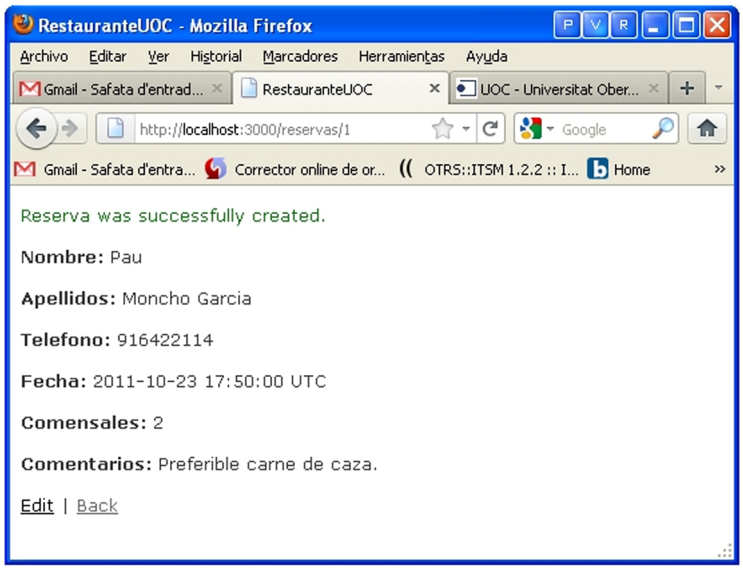


Figura 30. Formulari de creació d'una reserva



En crear la reserva, es mostra una fitxa (figura 31) amb la reserva en l'URL `http://localhost:port/reservas/1`. L'últim nombre de l'URL indica l'`id` de la reserva assignat en crear el registre. Al peu de la pàgina apareixen dos enllaços: el primer, `Edit`, que torna al formulari anterior per a modificar la reserva, mentre que el segon, `Back`, torna a la llista de reserves inicial.

Figura 31. Vista d'una reserva



Si es torna a la llista inicial, es pot veure el nou registre de reserva (figura 32). En la mateixa fila apareixen tres enllaços: `Show`, `Edit` i `Destroy`, que serveixen per a mostrar la reserva, editar-la i eliminar-la, respectivament:

Figura 32. Llista inicial amb la reserva creada recentment

3.2.2. Adaptació de la pàgina inicial

Si es vol accedir a la llista de reserves amb l'URL inicial de l'aplicació, com per exemple, `http://localhost:3000`, s'haurà de modificar l'arxiu `routes.rb` situat en el directori `config` afegint la línia següent:

```
root :to => "Reservas#index"
```

Aquesta línia encamina qualsevol petició dirigida a l'arrel cap a l'acció `index` del controlador `Reservas`. També s'ha d'eliminar la pàgina inicial anterior situada en el directori `public` amb el nom `index.html`. Perquè els canvis siguin efectius, s'ha de reiniciar el servidor web.

A més de seleccionar la llista de reserves, com a pàgina inicial, se li aplicarà un cert format utilitzant un full d'estil CSS. L'assignació del full d'estil es fa en el *layout* `application`, ja que d'aquesta manera s'aplicarà a totes les vistes de l'aplicació. Si es consulta el codi d'aquest, s'obté:

```
<!DOCTYPE html>
<html>
   <head>
      <title>RestauranteUOC</title>
      <%= stylesheet_link_tag :all %>
      <%= javascript_include_tag :defaults %>
      <%= csrf_meta_tag %>
   </head>
   <body>

      <%= yield %>

   </body>
</html>
```

La línia `<%= stylesheet_link_tag :all %>` indica que es carregaran tots els fulls d'estil que es trobin en el directori `public/stylesheets`. En aquest directori solament hi ha el fitxer `scaffold.css`, que és el full d'estil que crea la bastida per defecte.

Com la instrucció indica que carregarà tots els fulls d'estil, l'aplicació d'un nou estil és molt simple, ja que simplement emmagatzemant el nou full d'estil en el directori anterior, aquest s'aplicarà.

A més es modificarà el fitxer `index.html.erb` per a assignar-li l'estil, i quedarà de la manera següent:

```
<div id="reserva_list">
   <h1>Listing reservas</h1>
```

```
    <table>
      <% @reservas.eachdo |reserva| %>
      <tr class="<%= cycle('list_line_odd', 'list_line_even') %>">
        <td><%= reserva.nombre %></td>
        <td><%= reserva.apellidos %></td>
        <td><%= reserva.telefono %></td>
        <td><%= reserva.fecha %></td>
        <td><%= reserva.comensales %></td>
        <td><%= reserva.comentarios %></td>
        <td class="list_actions">
        <%= link_to 'Show', reserva %><br/>
        <%= link_to 'Edit', edit_reserva_path(reserva) %><br/>
        <%= link_to 'Destroy', reserva,
        :confirm => 'Are you sure?',
        :method => :delete %>
        </td>
      </tr>
      <% end %>
    </table>
</div>
<br />
<%= link_to 'New reserva', new_reserva_path %>
```

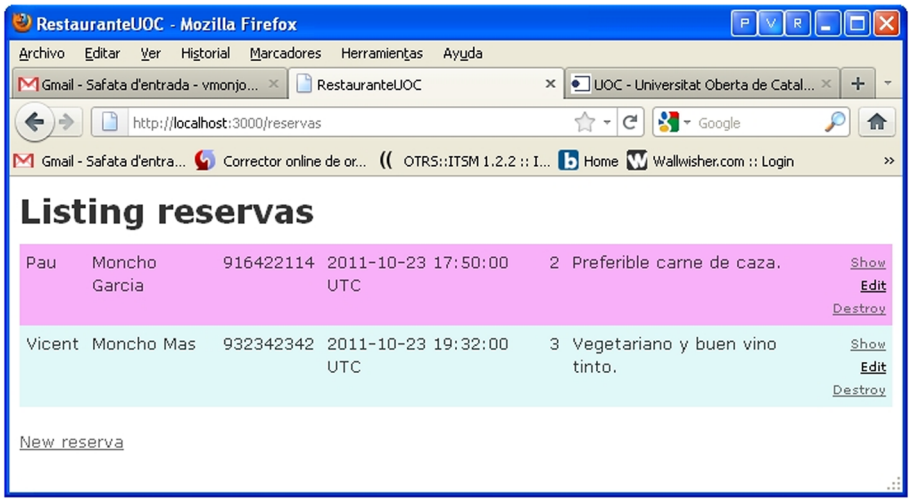
La sentència `cycle` és la responsable d'aplicar els estils de manera alternativa. Per acabar, en el directori `public/stylesheets` s'inserirà un fitxer CSS amb el codi següent:

```
#reserva_list table {
   border-collapse: collapse;
}

#reserva_list table tr td {
   padding: 5px;
   vertical-align: top;
}

#reserva_list .list_image {
   width:   60px;
   height:  70px;
}

#reserva_list .list_description {
   width:   60%;
}

#reserva_list .list_description dl {
   margin: 0;
```

```
}

#reserva_list .list_description dt {
   color:       #244;
   font-weight: bold;
   font-size:   larger;
}

#reserva_list .list_description dd {
   margin: 0;
}

#reserva_list .list_actions {
   font-size:      x-small;
   text-align:     right;
   padding-left:   1em;
}

#reserva_list .list_line_even {
   background:     #e0f8f8;
}

#reserva_list .list_line_odd {
   background:     #f8b0f8;
}
```

El resultat de l'aplicació de l'estil es pot apreciar en la figura 33.

Figura 33. Llista amb el format aplicat



3.2.3. Creació del formulari amb validacions

El lloc indicat en el patró MVC per a validar les entrades de valors és la definició del model, de manera que quan són desats en la base de dades, ja arriben amb valors validats.

Enllaç d'interès

Es pot consultar el detall de les validacions disponibles en "Active Record Validations and Callbacks", RailGuides [en línia].

En primer lloc, es consulta com és el codi del model `Reserva`:

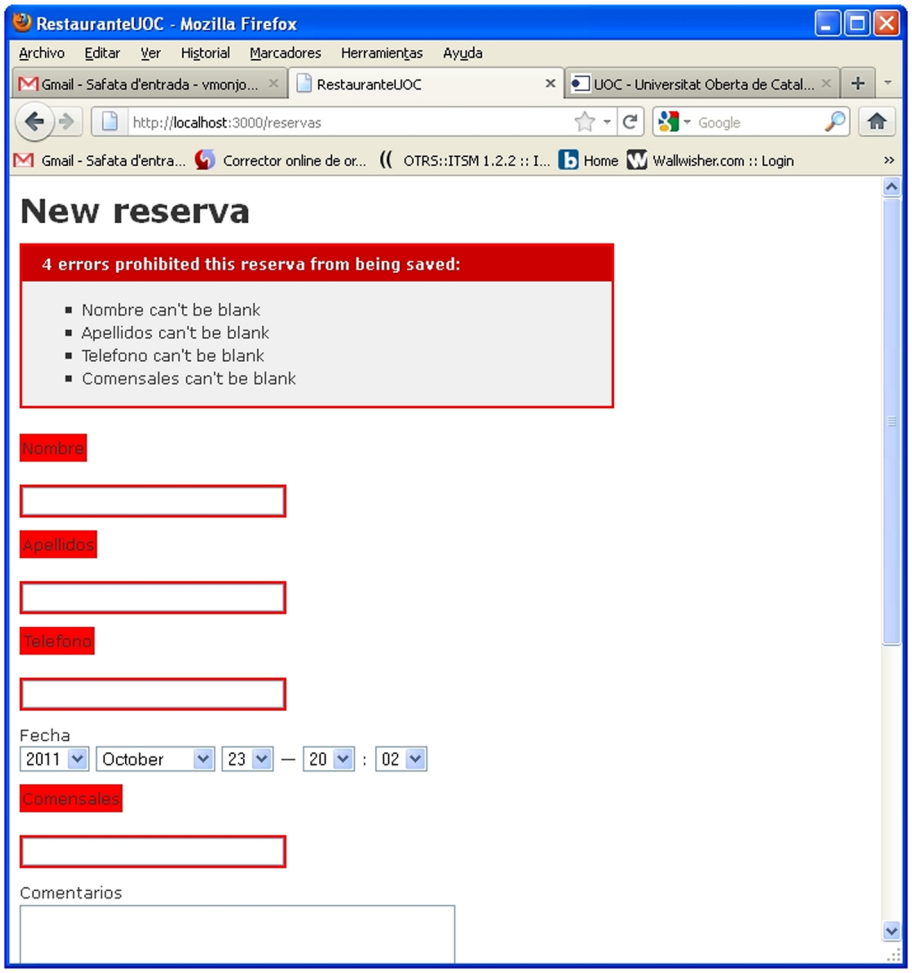
```
class Reserva < ActiveRecord::Base
end
```

El mètode `validates` s'utilitza per a fer validacions del text introduït. A continuació es provaran certes variants d'aquest mètode. El codi següent s'ha d'introduir en l'interior de la definició de la classe `Reserva`:

```
validates :nombre, :apellidos, :telefono, :fecha, :comensales, :presence => true
```

El codi anterior comprova que els camps no estan buits, i en cas que algun d'aquests estigui buit mostra l'error següent (figura 34):

Figura 34. Pàgina amb els avisos de les validacions

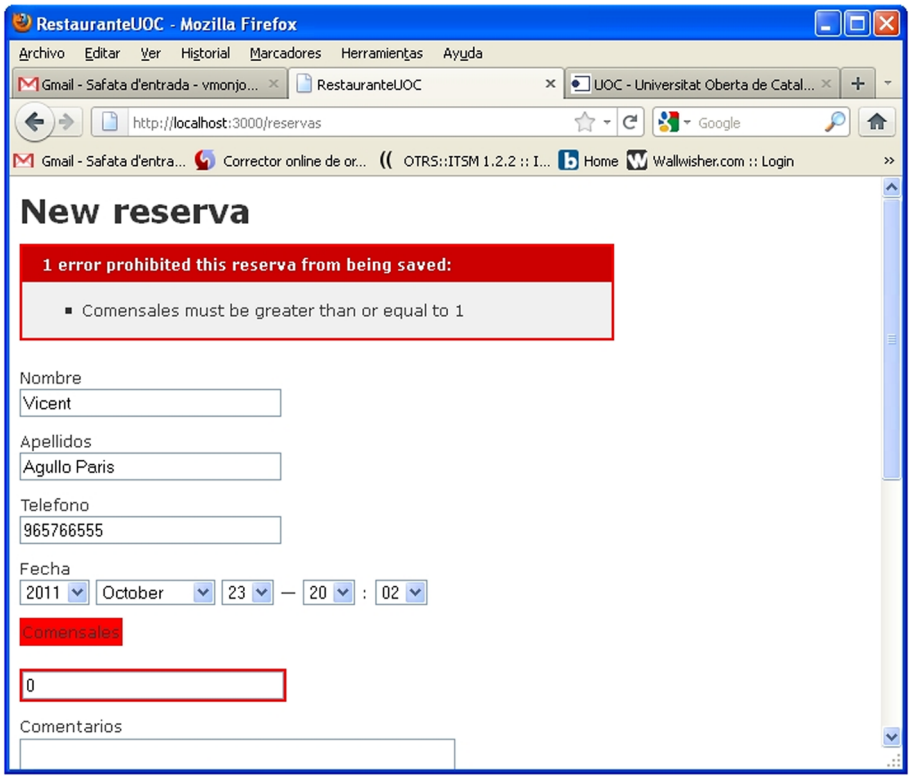


A continuació, el codi següent valida que el nombre de comensals és més gran o igual que un:

```
validates :comensales, :numericality => {:greater_than_or_equal_to => 1}
```

Amb la línia anterior, si no s'introdueix un nombre més gran o igual que 1, apareix el codi d'error següent (figura 35):

Figura 35. Pàgina amb la validació del nombre de comensals



Amb les dues validacions anteriors, el codi del model `reserva` queda de la manera següent:

```
class Reserva < ActiveRecord::Base
   validates :nombre, :apellidos, :telefono, :fecha, :comensales, :presence => true
   validates :comensales, :numericality => {:greater_than_or_equal_to => 1}
end
```

3.3. Fase 2. Gestió de reserves

3.3.1. Creació de l'MVC

El pas següent és crear un controlador i una vista que permeti gestionar les reserves dels propers dies. L'objectiu es basa a poder consultar les reserves dels propers dies per mitjà de l'URL següent:

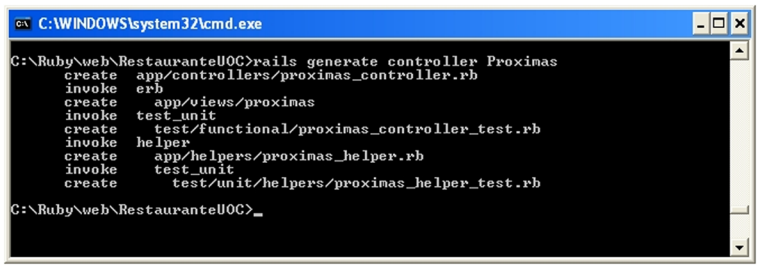
`http://localhost:3000/proximas?dias=5`

El paràmetre `dias` serà opcional i permetrà especificar els dies d'antelació en els quals es volen visualitzar les reserves. Si no s'especifica aquest paràmetre, es visualitzaran les reserves de les properes 24 hores per defecte.

Per a crear el controlador s'executa la sentència següent:

```
rails generate controller Proximas
```

Figura 36. Creació del controlador Proximas



El generador ha creat un fitxer anomenat `proximas_controller.rb` i en el seu interior la classe `ProximasController`, que encara no disposa de cap acció.

```
class ProximasController < ApplicationController
end
```

Es definiran dues accions `index` i `obtenerReservas` en el controlador anterior, que tindran el codi següent:

```
def obtenerReservas
   @dias = params[:dies] || 1
   inicio = DateTime.now
   fin = @dias.to_i.days.from_now
   @reservas = Reserva.where('fecha >= ? and fecha <= ?',
                             inicio.to_s(:db), fin.to_s(:db))
end

def index
   obtenerReservas
end
```

L'acció `obtenerReservas` fa les tasques següents:

- Obté de l'URL el paràmetre `dias` en la variable `@dias`. En el cas que el paràmetre no existeixi, s'establirà un dia per defecte.

- Crea una segona variable anomenada `inicio`, amb la data i hora en curs.

- Crea una tercera variable anomenada `fin`, en què s'estableix la data resultant de sumar a la data actual els dies obtinguts en el paràmetre.

- Finalment, es declara una quarta variable anomenada `@reservas` que contindrà les reserves amb data compresa entre la data d'inici i data de fi.

El conjunt de reserves s'obté a partir de la classe `Reserva`, es fa el filtratge per data mitjançant el mètode `where` i el resultat és un conjunt de reserves amb data en el període indicat.

L'acció `index` crida l'acció `obtenerReservas`. `Index` és el mètode per defecte del controlador, per la qual cosa no és necessari indicar l'acció en l'URL, simplement `http://localhost:3000/proximas`.

En aquests moments es crearà la vista associada amb l'acció `index` que permetrà visualitzar la llista de reserves contingudes en la variable `@reservas`. Per a això, s'utilitzaran parcials, ja que d'aquesta manera es podrà reutilitzar el codi en diferents vistes.

Així, es crearà `parcial` per a mostrar la llista de reserves, per la qual cosa es crearà un fitxer `_mostrarReservas.html.erb`, que es desarà en el directori `/app/views/proximas`.

El contingut del fitxer serà el següent:

```
<style type="text/css">
   div.Item {
       width:600px;
       border:1px solid gray;
       background-color:#FFF8F0;
       margin:10px ;
       text-align:left;
   }
</style>

<% @reservas.eachdo |reserva|%>
   <div class='Item'>
      <table>
         <tr>
            <td><b>Nombre</b></td>
            <td><%= reserva.nom %></td>
         </tr>
         <tr>
            <td><b>Apellidos</b></td>
            <td><%= reserva.apellidos %></td>
```

```
          </tr>
          <tr>
             <td><b>Teléfono</b></td>
             <td><%= reserva.telefono %></td>
          </tr>

          <tr>
             <td><b>Fecha y hora</b></td>
             <td><%= reserva.fecha.to_s(:short) %></td>
          </tr>

          <tr>
             <td><b>Comensales</b></td>
             <td><%= reserva.comensales %></td>
          </tr>
          <tr>
             <td><b>Comentarios</b></td>
             <td><%= reserva.comentarios %></td>
          </tr>
       </table>
    </div>
 <%end %>
```

L'etiqueta `<style>` defineix un estil per als elements `<div>` de la pàgina, i posteriorment es recorren totes les reserves mitjançant codi Ruby encastat. Cadascuna de les reserves està continguda en un `<div>` en el qual s'aplica l'estil definit. Dins de la capa es defineix una taula que mostra tota la informació d'una reserva accedint als diferents membres d'aquestes.

Però encara no es disposa de la vista, ja que el codi anterior és el parcial que crearà el contingut d'aquesta. La vista associada a l'acció `index` del controlador `proximas` es crearà en el directori `/app/views/proximas`, amb el nom `index.html.erb` i el contingut següent:

```
<%= render :partial => 'mostrarReservas' %>
```

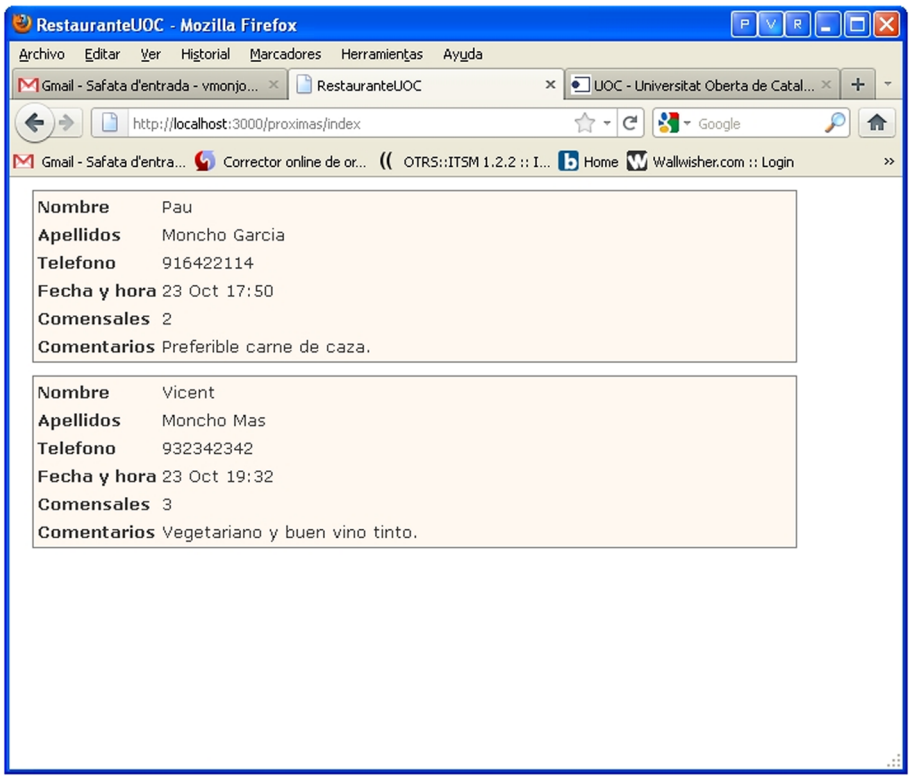
Quan s'executi el mètode `index` del controlador `proximas`, es renderitza el fitxer `index.html.erb`, que al seu torn crida el parcial `_mostrarReservas` i mostra totes les reserves contingudes en la variable `@reservas` proporcionada pel controlador.

Per a encaminar el controlador `proximas` amb la seva acció `index` s'ha d'afegir la línia següent en el fitxer `routes.rb`:

```
get "proximas/index"
```

Per a provar la nova acció, és necessari reiniciar el servidor i accedir a l'URL `/proximas/index`. És necessari crear una reserva per a abans de les properes 24 hores.

Figura 37. Llista de les properes reserves



En cas de voler consultar les reserves dels dos propers dies es farà la crida de la manera següent:

```
http://localhost:3000/proximas/index?dias=2
```

A continuació es modificarà el *layout* per a mostrar una capçalera, un peu de pàgina, i es definirà una secció dins de la qual mostrarà la vista en curs.

Perquè la capçalera i el peu de pàgina apareguin en totes les vistes de l'aplicació es modificarà el *layout* `aplication.html.erb`, que es desarà en el directori de *layouts* amb el contingut següent:

```
<!DOCTYPE html>
<html>
   <head>
      <title>Reservas: <%= controller.action_name %></title>
      <%= stylesheet_link_tag :all %>
      <%= javascript_include_tag :defaults %>
      <%= csrf_meta_tag %>
   </head>
   <body>
```

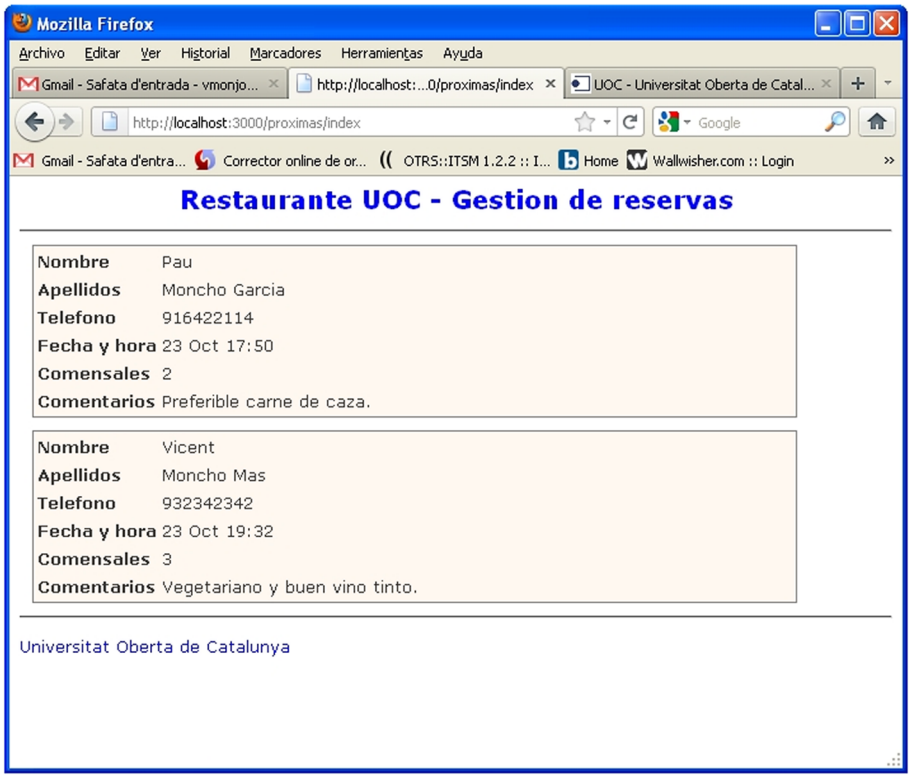
```
        <h2 style="color: blue" align="center">
           Restaurante UOC - Gestión de reservas
        </h2>
        <hr width="100%" align="center"/>
           <%= yield %>
        <hr width="100%" align="center"/>
        <p style="size: 8; color: blue">Universitat Oberta de Catalunya</p>

    </body>
</html>
```

Amb el nou *layout* es mostra el nom del restaurant i un peu de pàgina. En el cos es troba l'etiqueta `<%= yield %>`, que serà substituïda per la vista de cada acció. El resultat d'accedir a la pàgina `/proximas/index` és el següent:

Atenció!

Pot ser necessari reiniciar el servidor per a visualitzar els nous canvis en la plantilla.

Figura 38. Acció index del controlador reservas amb la plantilla modificada



3.4. Avís de reserves amb Ajax

Mentre es gestionen reserves, seria interessant conèixer si una reserva està propera perquè es pugui preparar el servei. Per a obtenir aquesta informació, es crearà un *layout* per al controlador `Reservas` afegint un procés que comprova periòdicament mitjançant Ajax si hi ha alguna reserva propera. En el cas que n'hi hagi alguna, es mostrarà un enllaç a la pàgina que les visualitza.

Reflexió

En afegir una funció Ajax en el *layout* de `Reservas` s'obtindran notificacions sobre noves reserves en qualsevol vista del controlador en la qual estiguem situats.

Prototype

Prototype és una biblioteca escrita en JavaScript per a facilitar el desenvolupament de pàgines web dinàmiques amb Ajax. Ofereix funcions que abstreuen les diferències entre les implementacions del DOM dels navegadors.

La plantilla `app/views/layouts/reservas.html.erb` contindrà el següent codi:

```
<!DOCTYPE html>
<html>
   <head>
      <script type="text/javascript" src="/javascripts/prototype.js"></script>
      <%= javascript_tag do %>
         new Ajax.PeriodicalUpdater('notificacionReserva',
                                    '/proximas/existeReserva',
         {
         method: 'get',
         frequency: 2,
         decay: 1
         });

      <% end %>
      <title>Reservas: <%= controller.action_name %></title>
      <%= stylesheet_link_tag :all %>
      <%= javascript_include_tag :defaults %>
      <%= csrf_meta_tag %>
   </head>

   <body>
      <h2 style="color: blue" align="center">
         Restaurante UOC - Gestión de reservas
      </h2>
      <div id="notificacionReserva"></div>
      <hr width="100%" align="center"/>
      <%= yield %>
      <hr width="100%" align="center"/>
      <p style="size: 8; color: blue">Universitat Oberta de Catalunya</p>
   </body>
</html>
```

Les diferències entre aquest *layout* i el que es va definir en el subapartat anterior són les següents:

1) L'ordre següent importa la biblioteca JavaScript Prototype per a l'ús d'Ajax:

```
<script type = "text/javascript" src = "/javas-
cripts/prototype.js"></script>
```

2) La sentència següent defineix una funció Ajax que fa peticions asíncrones de manera periòdica:

```
new Ajax.PeriodicalUpdater
```

Dins de la funció es defineixen les característiques següents:

- `notificacionReserva`: especifica la capa sobre la qual es carregarà el resultat de la crida.

- `/proximas/existeReserva`: indica el controlador i l'acció que es cridarà.

- `method`, `frequency` i `decay`: indiquen el mètode HTTP, la freqüència en segons de la crida periòdica i la velocitat a la qual la freqüència creix quan la resposta rebuda és exactament la mateixa que l'anterior.

3) La línia següent defineix la capa que contindrà la resposta de les peticions periòdiques:

```
<div id = "notificacionReserva"></div>
```

La modificació anterior fa peticions asíncrones cada dos segons a l'acció `existeReserva` del controlador `Proximas`; el contingut d'aquesta acció és el següent:

```
def existeReserva
   dias = params[:dias] || 1
   inicio = DateTime.now
   fin = dias.to_i.days.from_now
   @number = Reserva.where('fecha >= ? and fecha <= ?', inicio.to_s(:db),
   fin.to_s(:db)).count || 0
   render :layout => false
end
```

Amb l'acció anterior s'obté el nombre de reserves entre la data actual i els propers dies passats en el paràmetre. Aquest nombre s'emmagatzema en la variable `@number` i serà utilitzat per la vista `existeReserva.html.erb`.

Perquè l'acció `existeReserva` sigui accessible s'ha d'incloure en el fitxer `routes.rb` la línia següent:

```
get "proximas/existeReserva"
```

L'última instrucció indica que no s'ha de renderitzar cap *layout* per a aquesta acció. La crida feta és asíncrona.

La vista associada al mètode `existeReserva` mostrarà la informació en funció del nombre obtingut, de manera que si hi ha reserves mostrarà un enllaç a `/proximas` i en cas que no n'hi hagi cap retornarà un text informatiu.

S'ha de crear la vista `existeReserva.html.erb` dins del directori `app/views/proximas` amb el contingut següent:

```
<% if @number == 1 %>
   <%= link_to ('Hay 1 reserva en las próximas 24 horas',
      {:action => 'index', :dias => '1'},:popup =>
      ['new_Window','height=600, vwidth=650,scrollbars=yes']) %>
<% elsif @number > 1 %>
   <%= link_to ("Hay #{@number} reservas en las próximas 24 horas",
      {:action => 'index', :dias => '1'},:popup =>
      ['new_Window','height=600, width=650,scrollbars=yes']) %>
<% else %>
   No hay reservas en las proximas 24 horas.
<% end %>
```

El codi de la vista avalua el nombre de reserves obtingudes en el controlador a partir de la variable `@number`, de manera que depenent del valor fa les accions següents:

- Si el nombre de reserves properes és 1, es retorna un enllaç que obre una finestra emergent cap a `/proximas/index`, que mostrarà la llista de la propera reserva.

- Si el nombre de reserves és més gran que 1, es retornarà un enllaç semblant a l'anterior però amb el text en plural, atès que hi ha més d'una reserva.

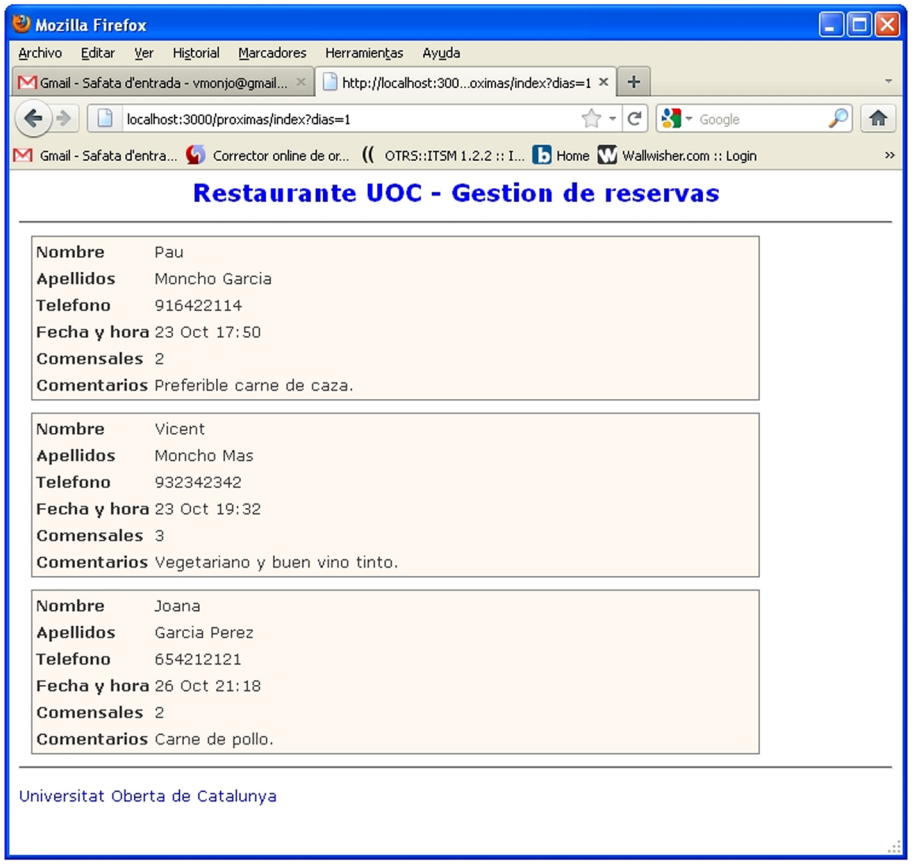
- Si no es compleix cap de les dues anteriors, es mostra un text que indica que no hi ha reserves en les properes 24 hores.

Per a provar el codi anterior, s'ha d'obrir l'URL `http://localhost:3000/reservas/`. Si hi ha una reserva en les properes 24 hores, la vista principal de gestió de reserves mostrarà un enllaç com el de la figura 39.

Figura 39. Actualització asíncrona de la pàgina amb notificació que hi ha una propera reserva

En fer clic en l'enllaç, s'obrirà una finestra emergent que mostrarà la reserva (figura 40).

Figura 40. Llista de properes reserves en una finestra emergent



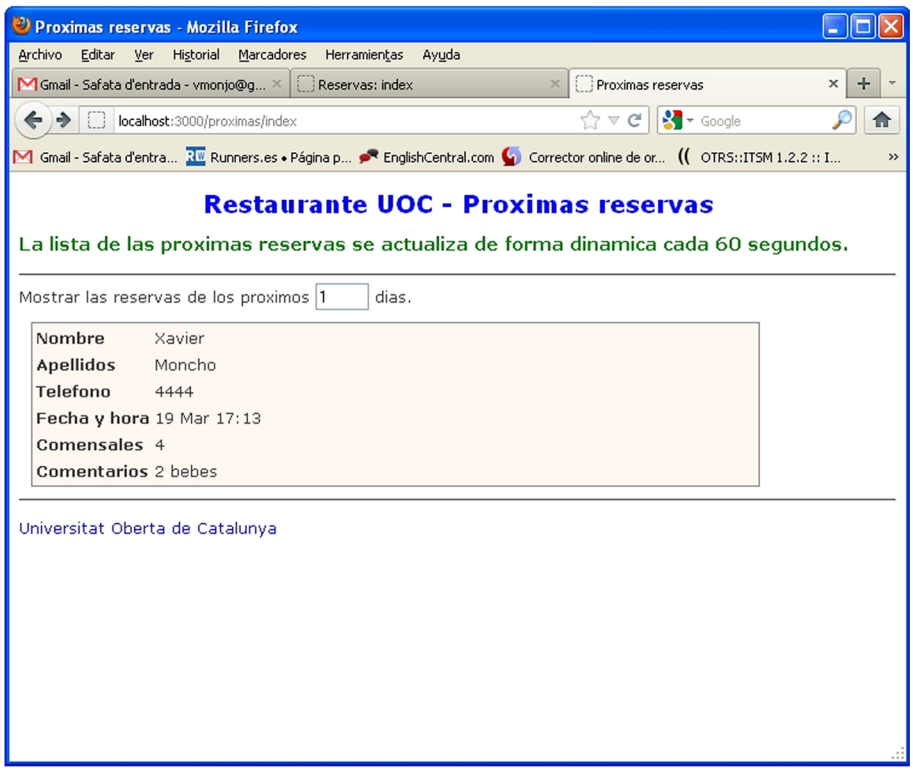
Layout per a les properes reserves

El pas següent és modificar el *layout* del controlador `Proximas` perquè la finestra emergent com la de la figura 40 mostri els elements següents:

- Informació sobre el restaurant amb una capçalera i peu de pàgina, similar al *layout* del controlador `Reservas`.
- Informació sobre la llista de properes reserves de manera automàtica, periòdica i asíncrona mitjançant Ajax.
- Una caixa de text per a poder indicar el rang de dies de properes reserves.

La finestra emergent es mostrarà de la manera següent (figura 41):

Figura 41. Finestra emergent amb el layout modificat



Aquesta vista s'actualitzarà a mesura que passi el temps amb les reserves que s'acostin a les properes 24 hores o en els dies introduïts en el camp de text.

El primer pas es basa en la creació d'una nova acció `buscarNuevas` dins del controlador `Proximas` que s'encarregarà d'obtenir la llista de les properes reserves:

```
    def buscarNuevas
      obtenerReservas
      render :layout => false
```

```
    end
```

De la mateixa manera que en l'acció anterior, és necessari encaminar-la perquè sigui accessible, per la qual cosa s'afegirà el codi següent al fitxer `routes.rb`:

```
get "proximas/buscarNuevas"
```

I es reiniciarà el servidor web perquè els canvis en la ruta tinguin efecte.

Igual que el mètode `index`, `buscarNuevas` crida `obtenerReservas` per a retornar a la vista les variables `@dias` i `@reservas`, que contenen el nombre de dies de la petició i el conjunt de reserves obtingudes de la base de dades.

La instrucció `render :layout => false` evita que la vista retornada per la crida asíncrona mostri el *layout* definit per al controlador `Proximas`, ja que es tracta d'una crida asíncrona i la vista obtinguda s'inserirà en una capa. Es crearà una vista a dins associada a l'acció `buscarNuevas` en el directori `/app/views/proximas` anomenada `buscarNuevas.html.erb`. El contingut del fitxer serà exactament igual que `index.html.erb`:

```
<%= render :partial => 'mostrarReservas' %>
```

Per acabar es crearà el *layout* `proximas.html.erb` en el directori `/app/views/layouts`, que tindrà el codi següent:

```
<!DOCTYPE html>
<html>
   <head>
      <script type="text/javascript" src="/javascripts/prototype.js"></script>

      <title>Proximas reservas</title>
      <%= stylesheet_link_tag :all %>
      <%= javascript_include_tag :defaults %>
      <%= csrf_meta_tag %>
   </head>

   <body>
      <h2 style="color: blue" align="center"> Restaurante UOC - Proximas reservas </h2>
      <h3 style="color: green" align="left">
         La lista de las proximas reservas se actualiza de manera dinámica
         cada 60 segundos.
      </h3>
      <hr width="100%" align="center"/>
      Mostrar las reservas de las proximas

      <%= text_field_tag "nreDias" ,"#{@dias}", :maxlength => 3, :size => 3 %> dias.
```

```
        <%= javascript_tag do %>
           new Ajax.PeriodicalUpdater('notificacionReserva', 'buscarNuevas',
        {
        parameters: 'dias=' + eval($('nreDias').value),
        method: 'get',
           frequency: 60,
           decay: 1
        });
        <% end %>

        <div id="notificacionReserva">
           <%= yield %>
        </div>
        <hr width="100%" align="center"/>
        <p style="size: 8; color: blue">Universitat Oberta de Catalunya</p>
    </body>
  </html>
```

Les etiquetes més destacables del codi anterior són:

- `<%= text_field_tag"nreDias", ...%>`: mostra una caixa de text que contindrà el nombre de dies obtingut per la variable `@dias`. A aquest control se li estableix l'identificador `nreDias`. La llista de properes reserves s'actualitzarà agafant els dies indicats en aquest control.

- `new Ajax.PeriodicalUpdater`: funció Ajax que fa consultes periòdiques. Cada minut actualitza la llista de properes reserves continguda dins de la capa `notificacionReserva`. L'acció feta és `buscarNuevas` del controlador `Proximas`.

Amb aquestes modificacions, es mantindrà la llista actualitzada amb les properes reserves a partir del valor de la caixa de text. Si es volguessin obtenir les properes reserves a una setmana, n'hi hauria prou de canviar el valor de la caixa de text a 7.