Introducción a Ruby on Rails

Vicent Moncho Mas

PID_00194071

Universitat Oberta de Catalunya

Índice

Introducción

En este módulo, se presentará el lenguaje de programación Ruby on Rails. En realidad se está frente a un lenguaje de programación Ruby que es utilizado como lenguaje de servidor en un entorno de programación (*framework*) Rails, que es la plataforma sobre la que se desarrollan las aplicaciones web. Ruby on Rails basa su popularidad y un gran número de adeptos en su agilidad y sencillez. Por un lado, Ruby es un lenguaje muy cómodo y expresivo que intenta acercarse al lenguaje humano, y por otro lado, Rails está basado en una filosofía práctica que intenta evitar la repetición de código y las múltiples configuraciones necesarias en estos entornos complejos de la programación web.

Objetivos

Los objetivos de este módulo didáctico son los siguientes:

1. Conocer qué es Ruby on Rails.

2. Entender la arquitectura básica de una aplicación Ruby on Rails.

3. Conocer los componentes del patrón de diseño modelo-vista-controlador (MVC).

4. Aprender a realizar aplicaciones sencillas con Ruby on Rails.

1. Qué es Ruby on Rails

Ruby on Rails, (RoR[1]), es un entorno de programación[2] cuyo objetivo es facilitar el desarrollo y el mantenimiento de aplicaciones web. Para ello se combinan un conjunto de características que lo hacen posible:

(1)RoR es la abreviatura de Ruby on Rails.

(2)En inglés, *framework*.

- Todas las aplicaciones se desarrollan utilizando la arquitectura MVC. Esto provoca que cada pieza de código tenga su lugar definido, por lo que es sencillo saber dónde se encuentra y dónde tenemos que introducirlo.

- Rails facilita la creación de tests unitarios, funcionales y de integración sobre la aplicación que se va creando. Esta característica supone un ahorro muy importante de tiempo para el programador profesional y un aumento en la calidad del software desarrollado.

- El uso de Ruby como lenguaje de programación moderno y orientado a objetos es explotado con el objetivo de simplificar el código final. La filosofía DRY[3] ('no te repitas') es implementada a partir del uso de clases de Ruby que definen los objetos necesarios para el patrón de diseño modelo-vista-controlador (MVC[4]).

- El uso de convenio en lugar de configuración; para ello se utilizan los valores definidos por defecto, lo que simplifica la necesidad de definir distintas configuraciones.

(3)DRY es la sigla de la expresión inglesa *Don't repeat yourself*.

(4)MVC es la abreviatura de *patrón de diseño modelo-vista-controlador*.

Son varios los factores que hacen que RoR sea una buena elección para el programador de aplicaciones web, pero hay que destacar que la curva de aprendizaje es menor que en otras tecnologías, si se dispone de experiencia en el mundo de la programación web.

1.1. Instalación de RoR

Ruby on Rails es un entorno de programación formado por varios componentes y cada uno de estos debe ser instalado en el puesto de trabajo. En los siguientes subapartados se van a explicar los pasos para proceder con la instalación en el entorno Windows.

El entorno Ruby on Rails necesita el siguiente software:

- El intérprete del lenguaje de programación Ruby. Se recomienda la versión 1.8.7 o 1.9.2, ya que estas dos son soportadas por Rails 3.0, que es la versión sobre la que hemos desarrollado este módulo.

- El sistema de paquetes de Ruby, más conocido como RubyGems. Se debe utilizar la versión 1.3.6, que es la que usamos en el módulo.

- El entorno de desarrollo Rails. En el módulo se utiliza la versión 3.0.

- Se instalaran bibliotecas[5] de software específicas para ciertas tareas.

- Se necesita un sistema gestor de base de datos. En este módulo vamos a utilizar SQL-lite, ya que es nativo y totalmente adaptado a Ruby on Rails.

- Un editor de texto para la manipulación del código. En este sentido se recomienda NotePad++, aunque existen muchas alternativas válidas y siempre depende del gusto del programador.

(5)En inglés, *libraries*.

1.1.1. Instalación en Microsoft Windows

1) Instalación de Ruby

El primer paso es la instalación de Ruby. Para ello se descargará del espacio de ficheros del aula el siguiente paquete de instalación de Ruby para Windows:

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

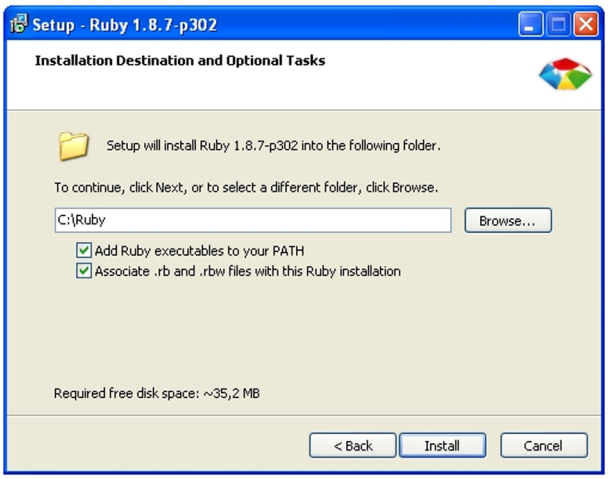
Reflexión

Los ejemplos del módulo se han desarrollado con la versión 1.8.7, por lo que es la versión recomendada. En caso de utilizar una versión distinta, es posible que estos no funcionen.

Al ejecutar el fichero, se debe aceptar la licencia y seleccionar las casillas de selección[6]; se cambiará el directorio a `C:\Ruby` y se pulsará *Install*, tal como se observa en la figura 1.

(6)En inglés, *check-boxes*.

Figura 1. Ventana de instalación de Ruby



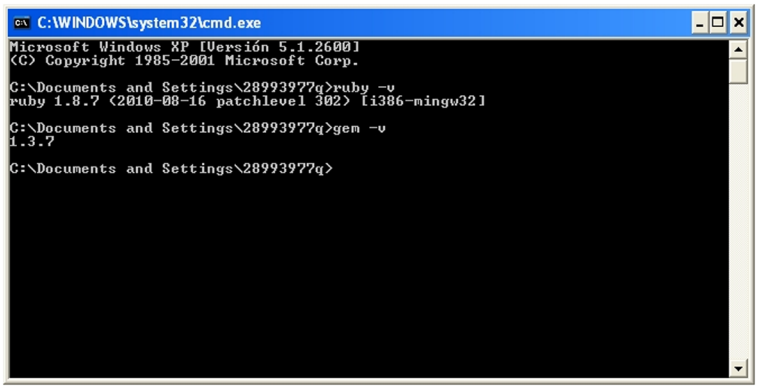
Para comprobar la instalación se ejecutan las siguientes sentencias en una ventana de comandos de Windows:

```
ruby -v
```

```
gem - v
```

El resultado será el que se observa en la figura 2.

Figura 2. Ventana de confirmación de la versión de Ruby y de RubyGems



Si no se utiliza el fichero instalable del aula, es imprescindible que la versión de RubyGems sea superior a 1.3.5. La actualización a la última versión se realiza con los siguientes comandos:

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

2) Instalación de SQLite

La instalación de SQLite requiere de los siguientes ficheros (disponibles en el aula):

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

El primer fichero contiene el aplicativo que permite el acceso y la modificación de bases de datos SQLite y el segundo las bibliotecas DLL del sistema (`sqlite3.def`, `sqlite3.dll` y `sqlite3.exe`).

El proceso de instalación se basa en la descompresión de los ficheros en la carpeta `C:\Ruby\bin`. Se debe tener especial cuidado en que estos ficheros se encuentren en el propio directorio y en un subdirectorio. Una vez realizado el punto anterior, se instalan los enlaces entre Ruby on Rails a SQLite3 mediante el comando siguiente:

```
gem install sqlite3-ruby
```

La ejecución del comando anterior requiere de una conexión a Internet activa. En caso de existir un servidor intermediario[7], se tiene que modificar la sintaxis para introducir las características de este último.

[7] En inglés, *proxy*.

3) Instalación de Rails

La preparación del entorno finaliza con la instalación de Rails. En el módulo se utiliza la versión 3.0.0. Esta se instala con el siguiente comando:

```
gem install rails –v 3.0.0
```

Se comprueba la versión de Rails instalada con el comando siguiente:

```
rails -v
```

Que retornará el valor 3.0.0.

1.1.2. Instalación en otros OS

Aunque Ruby on Rails es independiente de los sistemas operativos, la instalación del entorno en entornos OS-X o Linux tiene gran dependencia de la versión de los sistemas operativos, por lo que los procedimientos de instalación dependen de la versión sobre la que se instala y por tanto, es muy complejo definir un único procedimiento.

Por ello no se van a explicar los procedimientos de instalación para estos sistemas operativos, y en caso de que lo necesitéis, se darán las pautas o recomendaciones especificas a cada versión de sistema operativo, tanto basado en Linux como en OS-X.

1.1.3. El editor de código

El desarrollo de aplicaciones Ruby on Rails no necesita de complejos entornos de desarrollo integrados (IDE[8]) tal como sucede con otros lenguajes de programación. La mayoría de programadores actuales de Ruby on Rails utilizan un simple editor de texto que cumple una serie de requisitos básicos como los siguientes:

(8)IDE es la sigla de la expresión inglesa *integrated development environment*.

- El editor debería soportar código Ruby y HTML para facilitar el trabajo con ficheros erb.
- Es recomendable que este pueda crear el esqueleto de las estructuras estándar de Ruby.
- Es una ayuda el hecho de que la navegación por los ficheros sea sencilla (esto se entenderá más adelante cuando se explique la estructura de Rails).
- Debido a que en Ruby on Rails los nombres tienden a ser largos, es interesante que el editor sugiera nombres a partir de la introducción de varios caracteres.

Pero cada programador tiene sus propias preferencias, costumbres o manías, por lo que la elección final es responsabilidad de este. En el caso de que no se tenga un editor habitual, una recomendación avalada en la red es la de TextMate en el entorno OS-X, cuyo equivalente en Windows es E-TextEditor, aunque un simple editor como Notepad++ es suficiente para seguir este módulo.

En el momento de creación de este módulo existen complejos entornos IDE con soporte para Ruby on Rails, como Aptana, Komodo, NetBeans, pero no se recomiendan para el seguimiento de este módulo ya que, además de aprender Ruby on Rails, se debe aprender a utilizar estos entornos, lo que añade una dificultad innecesaria a los objetivos iniciales.

1.2. El lenguaje de programación Ruby

Ruby es un lenguaje de programación interpretado y orientado a objetos inspirado en otros lenguajes como Python, Perl y Smalltalk. El resultado es un lenguaje de programación conciso, legible y potente.

Ruby

Ruby fue creado por Yukihiro "Matz" Matsumoto, programador japonés. Inició su desarrollo en 1993 y lo presentó en el año 1995.

En el entorno de Matsumoto llamaban a este lenguaje *rubí*, dadas las similitudes con el lenguaje Perl, que significa 'perla'.

En este apartado se plantearán las características básicas del lenguaje que permiten seguir sin problema los algoritmos que se explicarán en los siguientes apartados del módulo. Por lo tanto, no se va a realizar un estudio detallado del lenguaje de programación Ruby; este daría para un módulo completo, o, posiblemente, para una asignatura completa.

1.2.1. Principales características

Las siguientes características definen a Ruby como un lenguaje orientado a objetos, moderno y sencillo:

a) Está orientado a objetos: todos sus tipos de datos son objetos, incluso los simples tales como los numéricos, los booleanos, los valores nulos, etc. Por tanto, en Ruby no existe un tipo de datos nativo como en lenguajes como JavaScript.

Sin tipos de datos nativos

Ruby es un lenguaje de código abierto y gratis. Puede ser usado, copiado, modificado y distribuido libremente.

b) Es un lenguaje interpretado: dispone de intérpretes para diferentes arquitecturas y sistemas operativos y estos permiten la ejecución de programas escritos en Ruby sin previa compilación. Esta característica tiene grandes ventajas, como la independencia de plataforma y la reflexión. Se utilizará el intérprete de Ruby para probar los conceptos que se presentarán en el siguiente apartado.

Independencia de la plataforma

Una aplicación Ruby es fácilmente portable a otros sistemas operativos como GNU/Linux, UNIX, Mac OS X y toda la familia de sistemas operativos Windows.

c) Es un lenguaje reflexivo: se puede generar y ejecutar código creado de forma dinámica. Por ejemplo, es posible ejecutar código Ruby contenido dentro de una cadena de caracteres en tiempo de ejecución. Esto implica que podemos modificar el código que se va a ejecutar durante el proceso de ejecución.

d) Dispone de una biblioteca estándar: un conjunto de clases y funciones escritas en Ruby que proporcionan soporte para realizar determinadas operaciones: tipos abstractos de datos, conectores[9], gestión de fechas, etc.

(9)En inglés, *sockets*.

e) Dispone de un *garbage collector*: al igual que en lenguajes como Java o JavaScript, la destrucción y liberación de memoria se gestiona de forma automática sin intervención del programador.

1.2.2. Tipos de datos

Tal como se ha comentado, una de las características de Ruby on Rails es la convención frente a la configuración. Esto afecta también a cómo se nombran las variables, los objetos, los métodos, etc. En particular, las siguientes reglas son "de obligado cumplimiento":

- Los nombres de las variables, métodos y los parámetros de estos últimos deben comenzar con una letra minúscula: `m_valor`, `s100`.

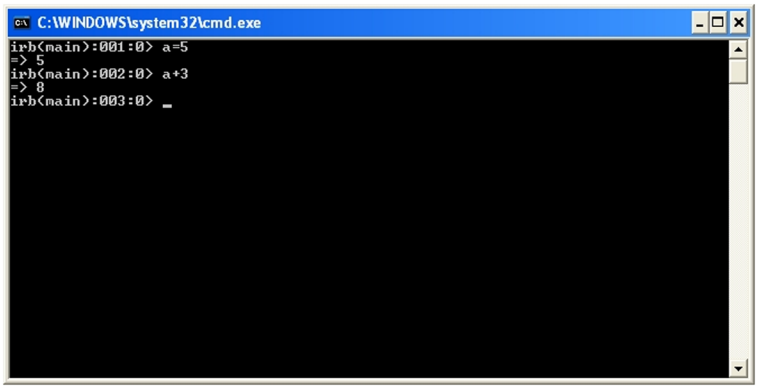
- Las instancias de las clases deben comenzar con el símbolo `@`: `@registro`, `@menu`.

- Los nombres compuestos se separan utilizando el símbolo "_": `valida_valor()`, `guarda_menu`.

- Los nombres de las clases, módulos y constantes comienzan con una letra mayúscula y en este caso la separación de nombres compuestos se implementa con una letra mayúscula: `MenuPrincipal`, `Comensal`.

La ejecución de las sentencias y estructuras se realiza abriendo en una consola de Windows y ejecutando el comando `irb`. Este comando ejecuta el intérprete de Ruby y permite al usuario ejecutar código (figura 3).

Figura 3. Ejemplo de ejecución de comandos en el intérprete IRB



La asignación de variables y los distintos operadores (`+`, `-`, `*`, `/`, `**`) se comportan de la misma manera que en el resto de lenguajes de programación. Se puede utilizar el intérprete para practicar con estos operadores.

El hecho de que cada valor es un objeto hace que cuando se asigne un valor a una variable, realmente se está creando una instancia de una clase (un objeto) y por tanto, se dispone de métodos y propiedades que se pueden utilizar. Por ejemplo, una variable con un valor entero (por tanto: un objeto) dispone de métodos como `to_s` (que convierte el valor a *string*), `to_f` (que convierte el valor a coma flotante) y `next` (que proporciona el siguiente entero).

Reflexión

El detalle de cada una de las clases que definen tipos de datos no es objetivo de esta asignatura, pero se puede consultar en línea cuando se necesite en Index of Classes & Methods in Ruby 1.9.3.

Cadenas

Las cadenas de texto se definen delimitando el texto mediante el uso de comillas, simples o dobles.

```
cadena1 = "esta es una cadena de caracteres\n"
cadena2 = 'y esta es otra'
```

La diferencia entre las dos definiciones anteriores es que las cadenas definidas mediante comillas dobles pueden incluir caracteres especiales tales como el `\t` (tabulador), `\n` (retorno de carro), y números en diferentes representaciones (octales, \061, hexadecimal, etc.).

Una característica interesante es que es posible introducir en una cadena el valor almacenado en una variable utilizando la notación `#{}`. Por ejemplo, en el siguiente código, se asigna a la variable cadena el texto "La edad es 25":

```
edad = 25
cadena = "La edad es #{edad}"
```

Al igual que en los tipos numéricos, la clase `string` dispone de un conjunto amplio de métodos, entre los que destacan las siguientes funciones relacionadas con letras minúsculas y mayúsculas.

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

Se accede a las letras de una cadena a partir del índice que define su posición, introduciendo este entre corchetes `[]` (en Ruby el primer carácter tiene el índice cero).

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

```
y[0] = "z"                #--> "zomate"
```

Los métodos `delete`, `insert` y `reverse` insertan, eliminan e invierten la posición de las letras de una cadena.

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

Para finalizar con las cadenas, se selecciona una sección de una cadena especificando el rango de esta.

```
n = "0123456789"
n[5...n.length]           # subcadena desde el quinto elemento hasta el final"56789"
```

Arrays

La estructura *array* en Ruby es muy similar a la existente en otros lenguajes de programación. Consiste en un conjunto de datos discretos que son accesibles por el índice que define su posición.

Estructura *array*

```
a = ["Menú", "de", "9", "euros"]
a.size                    #--> 4 el número de elementos en la lista
```

`a[i]` nos devuelve el (i – 1)-ésimo elemento del *array*.

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

El método `split` convierte una cadena en un *array* y `concat` añade al final del *array* el *array* indicado entre paréntesis.

Método *split*

```
b= "y 50 céntimos".split   #convierte la cadena en un array

a.concat(b)               #--> ["Menú", "de", "9", "euros" "y", "50", "céntimos"]
```

Las siguientes funciones son básicas en el manejo de *arrays*:

```
a.first                   #--> "Menú" devuelve el primer elemento de array
a.last                    #--> "céntimos" devuelve el último elemento del array
a.empty?                  #--> false pregunta si el array está vacio
```

Un *array* puede contener a su vez otros *arrays* (y convertirse en un *array* multidimensional):

***Array* multidimensional**

```
c = [1, 2, 3, [4, 5, [6, 7, 8], 9], 10]
```

El método `flatten` convierte el *array* multidimensional en unidimensional:

Método *flatten*

```
c.flatten         #--> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Para insertar elementos se utiliza el método `insert(índice, valor)`:

Método `insert`

```
c.insert(3, 99)   #--> [1, 2, 3, 99, [4, 5, [6, 7, 8], 9], 10]
```

El método `reverse` permite invertir el orden de los elementos del *array*:

Método `reverse`

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

Es posible ordenar los elementos alfabéticamente utilizando el método *sort*:

Método `sort`

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

Hash

Se trata de un *array* en el que los valores están indexados por cadenas en lugar de por números enteros.

Ejemplo

A continuación se crea un *hash* vacío al que se le van añadiendo elementos:

```
ingredientes = Hash.new           # declara un hash nuevo
ingredientes["arroz"] = "Bomba"   # el índice es "arroz" y el valor es "Bomba"
ingredientes["aceite"] = "Oliva virgen"
```

Normalmente se usa un *hash* para almacenar pares de valores relacionados entre sí, tales como las palabras de un diccionario y sus definiciones, las veces que aparece cada palabra en un texto o las propiedades de un objeto y sus valores respectivos.

Un *hash* se puede inicializar con todos los valores desde el principio:

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

A continuación se muestran distintos métodos de la clase *hash*:

```
lenguaje['Ruby']                      # devuelve el valor de la clave
lenguaje.has_key?('Java')             # devuelve false ya que no existe la clave
```

```
lenguaje.has_value?('McCarthy')       # devuelve false ya que no existe valor
lenguaje.sort                         # ordena por orden de la clave
lenguaje.values_at('Python','Ruby')   # devuelve los valores ['Guido', 'Matsumoto']
lenguaje.values                       # devuelve los valores ['Larry', 'Guido', 'Matsumoto']
lenguaje.keys                         # devuelve las claves ['Perl', 'Python', 'Ruby']
lenguaje.length                       # devuelve el número de elementos
```

Para reemplazar los valores existentes, se realizan simplemente asignaciones con el índice y el valor:

```
lenguaje['Ruby'] = 'Matz'             # reemplaza 'Matsumoto' por 'Matz'
lenguaje['COBOL'] = 'CODASYL'         # agrega un nuevo elemento al hash
```

Para eliminar pares que ya no se necesitan, se dispone de la función `delete`:

```
lenguaje.delete('COBOL')              # elimina el par 'COBOL'=>'CODASYL'
```

Fechas y horas

Ruby dispone de un módulo llamado `Time` que se utiliza para la manipulación de fechas y de horas. La fecha y hora actual se obtiene con el método `now`:

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

El resultado es una cadena formada por letras y números, a los que es posible acceder individualmente mediante los siguientes métodos:

```
t.day             # 10
t.month           # 9
t.year            # 2011
t.hour            # 23
t.min             # 11
t.sec             # 6
t.wday            # 1 primer día de la semana
t.yday            # 253 día del año
t.strftime("%B")  # "September" nombre del mes completo
t.strftime("%b")  # "Sep" idem abreviado
t.strftime("%A")  # "Monday" día de la semana
t.strftime("%a")  # "Mon" idem, abreviado
t.strftime("%p")  # "PM"  AM o PM
```

Una aplicación práctica de los métodos anteriores es el cálculo de la diferencia entre dos momentos de tiempo.

Aplicación práctica

La creación de un objeto del tipo fecha, se realiza con el método `mktime()`:

```
vacaciones = Time.mktime( 2011, "dec", 31, 1, 15, 20 )
```

Por otra parte, la clase `Date` se utiliza para especificar fechas que no requieren el uso del tiempo.

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

Este subapartado finaliza con el acceso a los elementos de un *array* o un *hash* de forma secuencial. La función `each` se aplica a un *array*, a un *hash* o a un *string* de varias líneas con la siguiente sintaxis:

```
variable.each {bloque}
```

La sentencia provoca la aplicación de las instrucciones que se encuentran dentro del bloque a cada uno de los elementos de la variable. Es decir, se produce una enumeración dentro de un ciclo repetitivo donde en cada iteración se evalúa uno de los elementos del *array, hash o string*.

Ejemplo con un *array*

El ejemplo siguiente utiliza un *array* y muestra cada elemento del *array* con una sentencia muy simple:

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

Ejemplo con un *hash*

En el siguiente ejemplo se utiliza un *hash*:

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

En lugar de las llaves `{}` del `each`, se pueden usar las palabras clave `do` y `end`. Esta sintaxis permite escribir el bloque de código en varias líneas, y es recomendable cuando el contenido del bloque tiene una lógica más elaborada, que es difícil expresar en una línea de código.

```
lineas = "En un lugar de la Mancha, de cuyo nombre...Fin\n".split
num = 0
lineas.each do |linea|              # usa do al principio del bloque
   num += 1                         # el contenido del bloque va de por medio
   print "Linea #{num}:  #{linea}"   # se extiende sobre varias lineas
end
```

Por tanto, se trata de un mecanismo que permite realizar iteraciones sobre estructuras complejas, aunque no sustituye los mecanismos clásicos de iteración, que se estudian en el subapartado siguiente.

1.2.3. Estructuras de control e iterativas

Estructuras de control

Las estructuras condicionales son similares en la mayoría de los lenguajes de programación. En Ruby, la estructura de control `if/elsif/else/end` tiene la siguiente sintaxis:

```
if expr1 [then|:]
   bloque1
[elsif expr2 [then|:]
   bloque2]
[else
   bloque3]
end
```

Se ejecutará el bloque de código en el que la `expr1` o `expr2` adquiere el valor *cierto*.

La sentencia `then` se puede omitir, y en lugar de esta se pueden usar dos puntos, `":"`, aunque estos también son opcionales.

Ejemplo

A continuación se presenta un ejemplo clásico del uso de la estructura, en la que se comparan dos números:

```
a = 5
b = 3
if a > b
   puts "a es mayor que b"
elsif a == b
   puts "a es igual a b"
else
   puts "b es mayor que a"
end
```

Ruby dispone de una sintaxis compacta que se utiliza para implementar estructuras condicionales más simples:

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

Que asigna el valor 1 a x si $a > b$, y 0 en caso contrario.

Cuando se quiere comparar una sola variable con una cantidad de valores se utiliza la estructura `case`. La sintaxis de esta estructura es la siguiente:

```
case variable
```

```
when valor1
   bloque1
[when valor2
   bloque2]
[else
   bloque3]
end
```

Cuando la variable tiene un valor igual a alguno de los examinados, entonces se ejecuta el bloque asociado a ese valor y se descartan el resto. En caso de que la variable no coincida con ninguno de los valores examinados, entonces se ejecuta la sección `else` y el `bloque3` (este último caso es conveniente para atrapar datos que están fuera de los rangos esperados). En cualquier caso, solo se ejecuta uno de los bloques, y los otros son descartados.

En la expresión anterior, los bloques `when` y `else` son opcionales; los bloques `when` pueden ocurrir por lo menos una vez y tantos como se necesiten. Si los valores comparados son numéricos, la expresión `when` puede tener valores puntuales (discretos), rangos o expresiones regulares.

Ejemplo

A continuación se plantea un ejemplo numérico de la estructura:

```
i = 8
case i
when 1, 2..5
   print "i esta entre 1..5"
when 6..10
   print "i esta entre 6..10"
else
   print "Error, dato fuera de rango"
end
```

Estructuras repetitivas

En el subapartado anterior se presentó un caso especial de repetición utilizando la función `each`, con la que se itera sobre elementos de un *array*.

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

Se puede conseguir el mismo resultado con la estructura `for/in`.

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

Cuando se va a repetir un ciclo un número definido de veces, el rango se indica con los límites de la iteración separados por dos puntos.

```
for i in 5..10
```

```
   puts i
end
```

En el caso de que no se quiera iterar hasta el último valor, sino que la iteración se detenga en el valor penúltimo (del 5 al 9 en el ejemplo anterior) se puede indicar añadiendo un tercer punto. De esta forma cuando se utilizan tres puntos se excluye el último valor del rango y esta sintaxis es muy útil cuando se itera sobre un *array* (cuyo último índice es $n - 1$).

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

En el caso de que el número de repeticiones sea conocido, se puede utilizar la sentencia `times`:

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

Repetiría *n* veces la sentencia `puts 'hola'`.

Una alternativa es el uso del método `upto` en un número entero:

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

De la misma forma se puede utilizar el método `downto`:

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

Para implementar iteraciones con pasos distintos de uno, se utiliza el método `step`, donde el primer parámetro es el número final y el segundo parámetro indica el incremento que se produce en cada iteración.

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

Es interesante el incremento que produce `step` cuando se utilizan números reales, ya que los incrementos son fraccionarios:

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

```
    puts r
end
```

La estructura `while` evalúa una expresión cuyo resultado es un valor booleano y repite la iteración tantas veces como la expresión sea cierta.

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

En la estructura `while` el uso de la palabra `do` es opcional.

De la misma manera la estructura `until` es similar a `while`, excepto que la expresión evaluada tiene una lógica negativa, es decir, el bloque se repite mientras la condición sea falsa.

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

La estructura `loop` crea un bucle inicialmente infinito, pero para salir de este se utiliza la instrucción `break` junto con una condición que la ejecuta. El ejemplo anterior se podría programar de la siguiente manera:

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

Una ventaja de esta estructura es el hecho de que la interacción puede finalizar en cualquier momento sin necesidad de ejecutar todo el código incluido en el cuerpo.

1.2.4. Clases y módulos

Clases

La definición de clases se realiza de manera similar que en el resto de lenguajes de programación orientada a objetos. En Ruby se deben tener en cuenta algunos aspectos sobre las variables y los métodos de la clase, que solamente se usarán dentro de la clase, mientras que otras serán visibles desde el exterior de la clase:

- A las variables internas las llamaremos *variables de la instancia* y para distinguirlas del resto de variables; van precedidas del signo `@`.

- A las variables visibles desde el exterior, se las conoce como *atributos*. Son propiedades del objeto y se pueden leer y modificar. Los atributos son sustantivos y sus valores son adjetivos o números. Por ejemplo: `perro.color`, `planeta.gravedad`, `motor.cilindros`, `paciente.edad`.

Las funciones que se definen en las clases se conocen como **métodos** de la clase y se utilizan verbos para describir el tipo de acciones que realizan. Por ejemplo: `perro.ladrar`, `motor.arrancar`, `paciente.crear`.

Ejemplo

A continuación se define una clase que define un perro que tiene nombre y que ladra:

```
class Perro                    # nombre de la clase

def initialize(nombre)         # método para inicializar
   @nombre = nombre            # @nombre variable interna
   @ladrido = "guau"           # @ladrido variable interna
end

def to_s                       # representación textual
   "#{@nombre}: #{@ladrido}" # devuelve valores internos
end

def nombre                     # exporta nombre como propiedad pública
   @nombre                     # devuelve el valor de @nombre
end

def nombre=(nombreNuevo)      # método que permite modificar el nombre
   @nombre = nombreNuevo       # asigna el valor a la variable interna
end

def <=> (perro)                # operador de comparación para ordenar
   @nombre <=> perro.nombre    # compara por nombre
end

def ladrar                     # define método ladrar
   @ladrido                    # devuelve el ladrido
end

end
```

En la definición de la clase anterior se tienen en cuenta los aspectos siguientes:

- El nombre de la clase siempre empieza en mayúscula.

- Las variables que se declaran dentro de la clase tienen alcance local por defecto y van precedidas por el signo `@`.

- El método `initialize()` se conoce como **constructor**, ya que es el primer método que se ejecuta cuando se crea una instancia de la clase. El constructor es opcional, pero se suele utilizar para efectuar cualquier tipo de inicialización dentro de la clase. Este método se ejecuta automáticamente (nunca se invoca explícitamente) al crear una instancia de la clase con el método `new`.

- Es conveniente también definir el método `to_s` donde se indica qué se debe devolver cuando se solicita la representación textual de la clase.

- El método de comparación <=> se define para hacer que la clase sea comparable, es decir, que se le puedan aplicar operaciones tales como `sort()`. Consiste en declarar qué variable interna se utilizará para implementar comparaciones. En el ejemplo anterior, las comparaciones se ejecutan por nombre.

- El método `nombre()` devuelve el nombre del objeto.

- El método `ladrar()` hace que el perro ladre (solo devuelve el valor de la variable `@ladrido`).

Pero una vez está definida la clase, se va a crear una instancia y esto se implementa indicando el nombre de la clase seguido de la palabra `new` y los posibles parámetros que se hayan definido en el método `initialize`.

Ejemplo

```
f = Perro.new("fifi")
m = Perro.new("milu")
puts f.ladrar
puts m.ladrar
puts f.to_s
puts m.to_s
m <=> f
```

En la clase `Perro` no se puede acceder al ladrido directamente ya que es una variable interna. Ahora bien, se puede hacer visible un atributo interno a partir de la definición de un método que devuelva el valor de este.

Ejemplo

```
class Perro

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

def nombre      # propiedad pública legible
   @nombre      # retorna el nombre
end
...
end
```

A continuación se utiliza el método público definido para acceder al valor del nombre de la clase:

```
f.nombre
```

Pero para que la propiedad pueda ser modificada es necesario definir un método que modifique su valor:

```
class Perro
...
def nombre=(nombreNuevo)
   @nombre = nombreNuevo
end
...
end
```

A continuación se utiliza el método anterior para modificar el nombre de `fifi`.

```
f = Perro.new("fifi")
f.nombre = "fifirucho"
```

Ruby permite implementar esta característica de forma más sencilla, aunque no sea muy ortodoxa en el mundo de la programación orientada a objetos. Se pueden declarar atributos legibles utilizando la palabra clave `attr_reader`.

```
class Perro
   attr_reader :nombre, :fecha_de_nacimiento, :color
...
end
```

Pero se debe modificar el constructor para que este acepte los valores iniciales:

```
class Perro
...
def initialize(nombre, nacimiento, color)
   @nombre = nombre
   @fecha_de_nacimiento = nacimiento
   @color = color
end
...
```

```
end
```

Con el código anterior se puede acceder a las propiedades de la siguiente manera:

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

Es posible definir un atributo como modificable añadiendo `attr_writter` en la declaración de este. Con la sintaxis anterior el atributo será actualizable directamente, tal como se puede observar en los siguientes ejemplos:

```
class Perro
   attr_writer :sobrenombre
...
end
```

Por lo que sería modificable utilizando la siguiente sintaxis:

```
f = Perro.new("fifi", "20000621","gris")
f.sobrenombre = "fifirucho"
```

Pero si se necesita tanto poder acceder como modificar los atributos, se utiliza `attr` en la declaración.

```
attr :nombre, true
```

El valor booleano especifica que además de accesible sea modificable.

Las variables precedidas por dos puntos se conocen como **símbolos** y son valores constantes cuyo valor es una cadena con el mismo nombre y que no pueden ser modificados o actualizados.

Módulos

Los módulos proporcionan un mecanismo para agrupar código que puede ser reutilizado.

La creación de un módulo es muy simple ya que, como se puede observar a continuación, se agrupan funciones y clases. Utilizan la sintaxis siguiente:

```
module MiModulo
   PHI = 1.61803398874989    # el cociente aureo

def mifuncion
   puts "Un saludo desde mifuncion"
```

```
end

class Miclase

def mimetodo
   puts "Un saludo desde Miclase.mimetodo"
end

end

end
```

Para utilizar el módulo se necesita una sentencia `require` que indique el nombre del módulo que se quiere utilizar.

```
require 'MiModulo'
cociente = MiModulo::PHI
c = MiModulo::Miclase.new
c.mimetodo
```

En una clase se pueden incluir funciones definidas en un módulo (utilizando la sintaxis `include`), de forma que se amplía la funcionalidad de estas:

```
require 'MiModulo'

class MiClase1
   include MiModulo
end

class MiClase2
end

m1 = MiClase1.new
m1.mifuncion
m2 = MiClase2.new
m2.extend(MiModulo)   # extiende la instacia
m2.mifuncion          #"Un saludo desde mifuncion"
```

La técnica anterior se conoce como **polimorfismo de interfaces** y permite ampliar clases y asignarles un comportamiento especial compartido.

Un módulo predefinido que se utiliza normalmente de esta manera es `Enumerable`. Se definen los métodos `initialize` y `each` y al implementar el polimorfismo, métodos como `collect`, `detect`, `map`, `each_with_index` son heredados y por tanto se pueden utilizar en la instancia de la clase.

Ejemplo de polimorfismo de interfaces

En el siguiente ejemplo se observa esta potente técnica.

```
class MultiArray
include Enumerable

def initialize(*arrays)   # acepta una coleccion de arrays
   @arrays = arrays
end

def each                  #por cada uno, itera su contenido
   @arrays.each { |a| a.each { |x| yield x } }
end

end
```

A continuación utilizamos la clase 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 "El elemento #{i} es #{x}" }
```

1.3. El entorno de programación Rails

En este subapartado se explican los conceptos básicos que definen Rails y que provocan que este entorno de programación sea considerado por los programadores como un entorno simple, potente y de fácil mantenimiento.

1.3.1. La filosofía "convención frente a configuración"

Para simplificar el proceso de desarrollo es fundamental simplificar al máximo las decisiones que el programador tiene que tomar. Rails toma estas decisiones y las define por defecto. Esta técnica se conoce como **valores por convención**.

La filosofía anterior es la base del entorno de programación. Con la configuración por defecto se puede crear una aplicación totalmente funcional en un tiempo récord (tal como se verá en el subapartado siguiente).

Las opciones definidas por defecto de Rails implican opciones de:

- Configuración compleja de acceso a bases de datos.
- Configuración de la comunicación entre la aplicación del servidor y la del usuario.
- Organización de los directorios y de su contenido.

Toda la estructura por defecto cumple el patrón modelo-vista-controlador.

1.3.2. El patrón modelo-vista-controlador

El patrón modelo-vista-controlador se basa en la separación de la interfaz de usuario, los datos y la lógica de control, en tres componentes completamente diferenciados.

El modelo especifica además el mecanismo que define la relación entre estos:

a) Modelo: gestiona los datos de la aplicación; se encarga de la carga, persistencia e integridad, y realiza la conexión con la base de datos, los ficheros binarios, XML, etc. Tal como se verá más adelante, en el modelo no solo se gestiona el acceso a los datos sino también las reglas de validación de estos, y separa estas de los otros dos componentes.

b) Vista: es el componente responsable de generar las interfaces de usuario. Parte de los datos mostrados en las interfaces se basan generalmente en los datos obtenidos a partir del modelo. En una aplicación web, las vistas representan el conjunto de páginas HTML (y las generadas de forma dinámica), código JavaScript y estilos CSS asociados.

c) Controlador: recibe las peticiones provenientes de la interfaz de usuario y coordina las respuestas. Cada petición se delega a un controlador que se encarga de ejecutar las acciones requeridas hasta generar una respuesta.

La utilidad del controlador

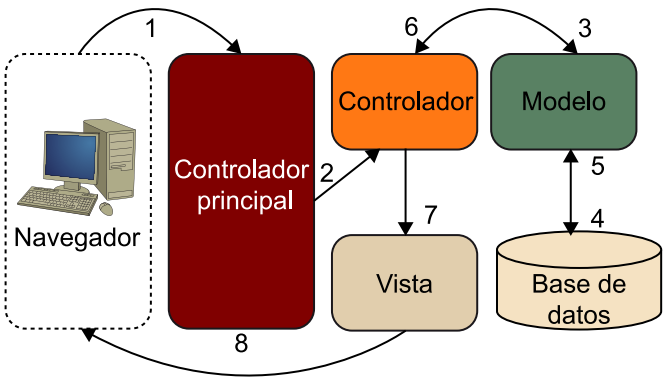
Se dispone de una aplicación web para la venta de libros. Si el usuario hace clic sobre el título de un libro para visualizar sus características, el servidor recibe la petición y delega la tarea al controlador encargado de esta función. Este accederá al modelo, obtendrá el libro con el ISBN indicado en la URL como parámetro y devolverá al navegador una vista que mostrará los detalles.

Las acciones

Las acciones también pueden ser generadas a partir de un proceso periódico sin la intervención del usuario.

La figura 4 muestra los tres tipos de componentes y la relación entre ellos desde el punto de vista de una acción, enumerando cada paso.

Figura 4. Arquitectura MVC



1) El usuario realiza una acción sobre la interfaz de usuario. Por ejemplo, modificar un campo o solicitar información que debe ser visualizada.

2) El controlador principal puede delegar en uno específico.

3) El controlador accede al modelo para efectuar las consultas o modificaciones (en caso de que sea necesario).

4) El modelo consulta la base de datos en caso de que desee obtener valores.

5) La base de datos devuelve los valores al modelo.

6) El modelo devuelve el control al controlador anterior.

7) El controlador selecciona la vista encargada de visualizar la respuesta y pasa los datos que debe visualizar obtenidos por el modelo.

8) La vista se envía de vuelta al usuario como respuesta a la petición.

Los pasos anteriores se reproducen en cada una de las peticiones del navegador del usuario, de forma que el controlador adecuado recibe la petición, la procesa (solicitando datos a través del modelo si fuera necesario), y cuando la tiene finalizada, llama a la vista adecuada para que muestre la respuesta al usuario final.

1.3.3. Estructura de carpetas y ficheros

Siguiendo la filosofía de "convención frente a configuración", uno de los elementos que está estandarizado en Rails es la estructura de carpetas y ficheros de un proyecto. La creación de un proyecto Ruby on Rails crea por defecto una estructura de carpetas y ficheros.

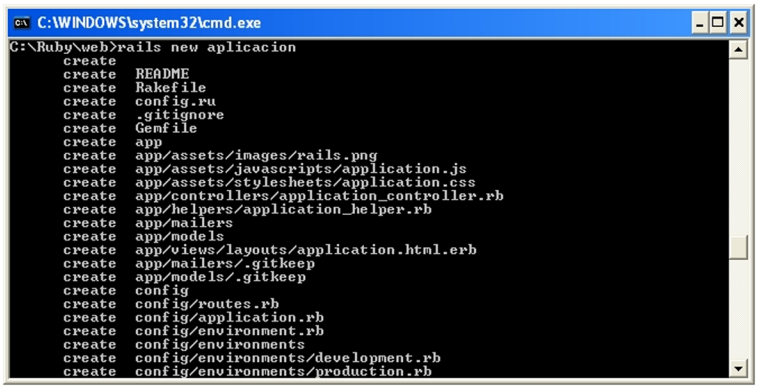
¿Para qué sirve cada uno? A continuación se explica brevemente el objetivo de cada uno de los elementos de esta estructura.

Para empezar es necesario crear un proyecto para que se genere la estructura. "Por convenio", se crea un directorio `web` en el interior del directorio de instalación de Ruby. Creado este directorio, se abrirá la consola de comandos de Windows, se situará en el directorio recién creado, y se ejecutará el comando siguiente:

```
rails new aplicacion
```

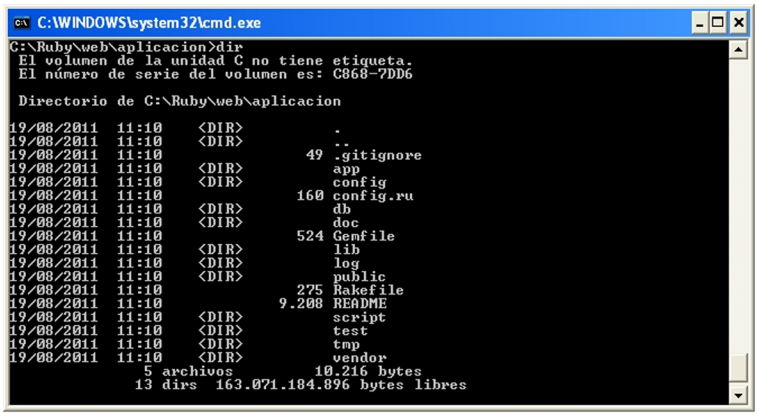
El comando anterior creará la estructura de carpetas y ficheros del nuevo proyecto creado (figura 5).

Figura 5. Creación de un nuevo proyecto Ruby on Rails



Se ha creado un nuevo directorio `aplicacion`, donde su contenido se observa en la figura 6.

Figura 6. Estructura de directorios de un proyecto



En primer lugar aparecen un conjunto de ficheros en el directorio raíz de la aplicación. Se trata de ficheros que el programador no va a tener que modificar ni utilizar. A continuación se realiza una pequeña descripción de los ficheros principales que cumplen esta característica:

- `config.ru`: es el fichero que contiene la configuración para arrancar la aplicación en servidores web.
- `Gemfile`: este fichero especifica las dependencias que tiene la aplicación con periféricos[10] externos.
- `Rakefile`: es el fichero donde se definen las tareas para la ejecución de test, creación de documentación, generación de la base de datos, etc.
- `README`: es el fichero clásico de presentación del entorno de programación.

[10] En inglés, *plug-ins*.

En segundo lugar aparecen un conjunto de directorios/subdirectorios, cuyo contenido se describe brevemente a continuación:

a) **`app`**: se trata del directorio más importante, en este se almacena el código fuente de la aplicación. Este dispone a su vez de su propia estructura de subdirectorios:

- **`views`**: donde residirán los ficheros de vistas del proyecto.
- **`controlers`**: donde se guardan los ficheros de los distintos controladores de la aplicación.
- **`models`**: donde se almacenan los ficheros de cada uno de los modelos de la aplicación.
- **`mailers`**, **`assets`** y **`helpers`**: que almacenan otros ficheros que forman parte de la aplicación.

b) **`config`**: en este directorio Rails almacena los ficheros que definen la configuración por defecto (o por convención). En este módulo no será necesario modificar ninguna configuración estándar.

c) **`db`**: en este directorio se almacenan los ficheros de creación del esquema de la base de datos y además en cada modificación o migración de la base de datos se crea un fichero que se guarda en este directorio.

d) **`doc`**: una característica interesante de Rails es que se trata de una herramienta que es capaz de generar la documentación de las aplicaciones de forma automática y sencilla. En este directorio se guarda esta documentación.

e) **`lib`**: en la mayoría de aplicaciones se utilizan bibliotecas externas que realizan tareas específicas y complejas que evitan que el programador tenga que implementarlas desde cero. Estas bibliotecas se guardan en este directorio.

f) **`log`**: se trata de uno de los directorios más útiles cuando se está en la fase de desarrollo ya que almacena ficheros de registro de sesión[11] con información muy detallada sobre la ejecución de la aplicación.

[(11)] Ficheros *log*, según su expresión inglesa.

g) **`public`**: el servidor web utiliza este directorio como la base de la aplicación, ya que en este se almacenan las páginas web estáticas, ficheros CSS y Javascript.

h) **`script`**: en este directorio se almacenan los *scripts* definidos por el programador; normalmente son *scripts* que agrupan comandos que normalmente se utilizan en tareas de mantenimiento.

i) **`test`**: otra característica interesante de Rails es la capacidad de crear *tests* unitarios, funcionales y de integración. Todos los ficheros que definen estos *tests* se almacenan en este directorio.

j) **`tmp`**: es el directorio donde se almacenan los ficheros temporales; en este aparecerán ficheros de contenidos *cache*, de sesiones de usuario, de *sockets*, etc. En general, este directorio se vacía automáticamente, pero es posible que en alguna ocasión sea necesaria la eliminación manual de ciertos ficheros.

k) **`vendor`**: es el lugar donde se almacenan los ficheros de control de los periféricos que utiliza la aplicación.

La estructura anterior no se debe modificar y es común en todas las aplicaciones Ruby on Rails. Gracias a esta característica la localización de un fichero es muy sencilla navegando por la estructura anterior.

El detalle de los ficheros y subcarpetas se irá presentando en el módulo a medida que sea necesario, pero este apartado ya proporciona una primera aproximación.

1.4. La primera aplicación "Hola Mundo"

En todos los manuales de programación, hay un primer apartado en el que se crea una aplicación "Hola Mundo". El objetivo del apartado es hacer que el programador cree su primera aplicación con unas simples líneas de código.

En Ruby on Rails una aplicación simple como "Hola Mundo" crea la misma estructura que una aplicación con mayor complejidad, pero lo interesante es que la mayor parte del trabajo habrá sido realizado por Rails, y no por el programador.

1.4.1. Creación de la aplicación

El primer paso es la creación de la aplicación. Para ello se ejecuta el comando siguiente:

```
railsnew hola
```

En este momento se ha creado la estructura de directorios y de ficheros, pero en esta primera aplicación se van a ignorar todos excepto algunos ficheros del directorio `app`.

Pero la aplicación es web y por tanto es necesario disponer de un servidor web que sea capaz de ejecutarla. En el subapartado siguiente se prepara el servidor web.

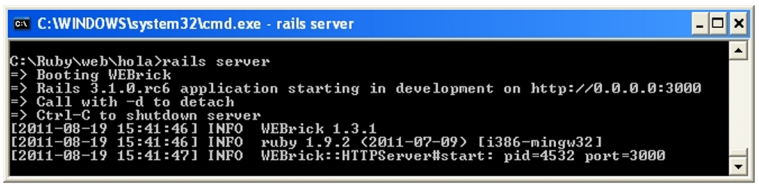
1.4.2. Arranque del servidor web

La instalación realizada de Ruby on Rails lleva incorporado por defecto el servidor web WEBrick. El arranque del servidor se realiza con el comando `rails server`, ejecutado en el directorio recién creado (`ruby/web/hola`):

```
rails server
```

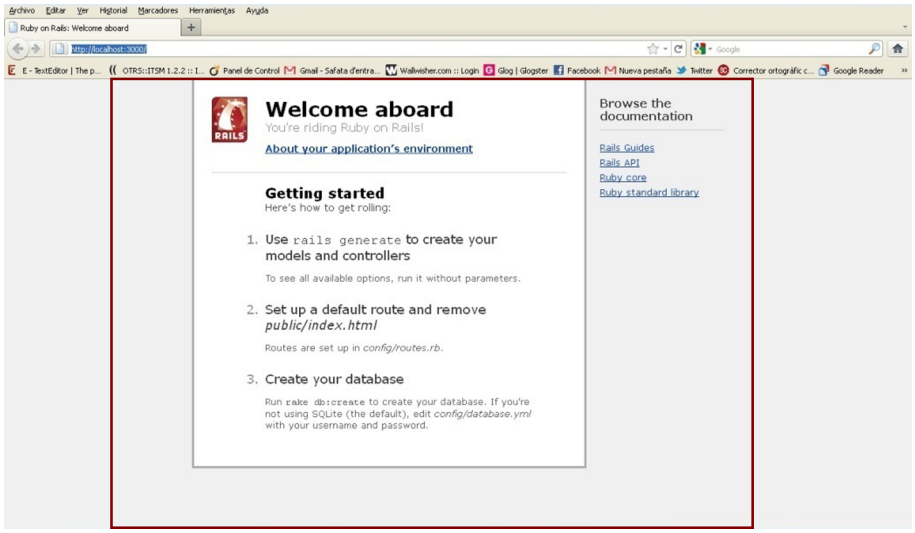
Al ejecutar el comando aparece el siguiente texto en la consola:

Figura 7. Arranque del servidor WEBrick



El servidor web ya está en marcha. Es posible comprobar que funciona correctamente abriendo el navegador e indicando la siguiente dirección web: `http://localhost:3000`. En la ventana del navegador tiene que aparecer la página que se muestra en la figura 8.

Figura 8. Página de inicio de un proyecto Ruby on Rails vacía



En estos momentos ya se dispone del servidor web a la escucha en el puerto 3000 (opción por defecto de Ruby on Rails). Asimismo, la página inicial que aparece en el navegador es la página inicial de la aplicación `hola`.

Pero el objetivo de este subapartado es mostrar el mensaje "Hola Mundo!". Para ello se tiene que crear un controlador, que será el encargado de procesar la petición del navegador y devolver la vista que contendrá el mensaje "Hola Mundo!".

1.4.3. Creación del controlador y de la vista

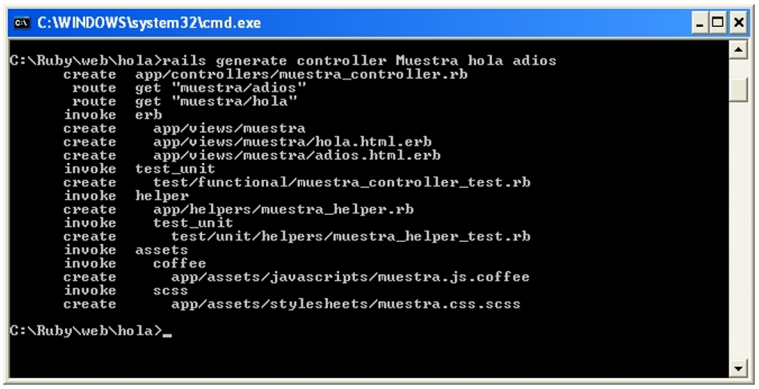
En este subapartado se tratarán aspectos que dan una ligera idea del potencial de Ruby on Rails. Para empezar, el *script* `generate` crea el código base del controlador y de cada una de sus acciones. Este *script* tiene que ir seguido del nombre del controlador y de las acciones que este va a necesitar.

Como la ventana de comandos está ocupada ejecutando el servidor web, es necesario abrir una ventana de comandos nueva para poder ejecutar el comando en el directorio de la aplicación (`ruby/web/hola/`):

```
rails generate controller Muestra hola adios
```

En la ventana de comandos aparecerá lo siguiente (figura 9):

Figura 9. Creación del controlador `Muestra` y sus métodos `hola`/`adios`



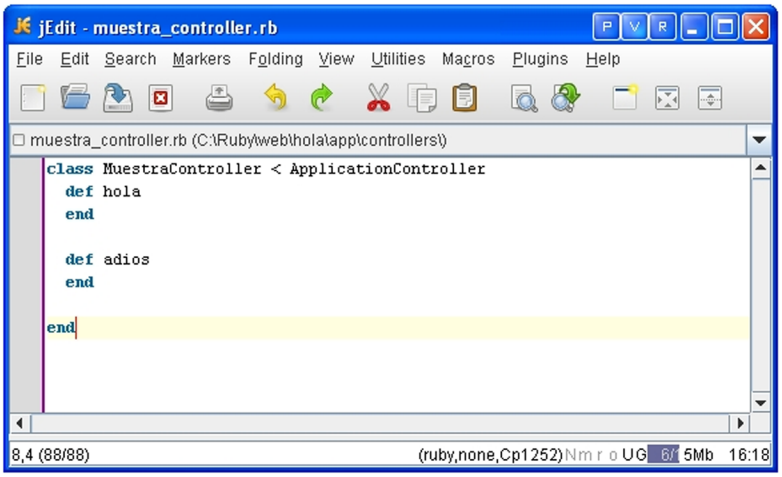
Se ha creado el controlador `Muestra`, se ha creado una vista para el controlador y una para cada una de las acciones `hola` y `adios`. Además se crean `tests`, `helpers` y `assets`, aunque estos no se van a estudiar en este módulo.

De todos los ficheros creados, a continuación (figura 10) se muestra el código del fichero que define el controlador.

El código es muy simple. Se observa que la clase `MuestraController` hereda de la clase `ApplicationController` y tiene definidas dos acciones (totalmente vacías en estos momentos).

La navegación en aplicaciones Ruby on Rails se realiza a partir de la estructura de la URL. En esta se tiene que indicar el identificador del controlador seguido del símbolo "`/`" y finalizará con el nombre de la acción.

Figura 10. Código del controlador `Muestra` y sus acciones `hola`/ `adios`



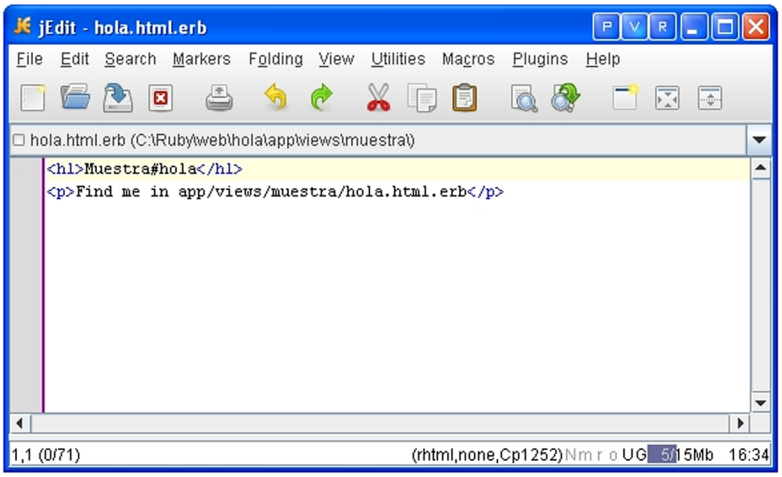
Al escribir la URL `http://localhost:3000/muestra/hola` se obtiene la siguiente respuesta en el navegador (figura 11):

Figura 11. Página de la vista `hola`

¿Cómo ha ocurrido? Es sencillo: el navegador ha llamado a la acción `hola` del controlador `Muestra`. Esta acción no tiene ningún código definido, por lo que no ejecuta ninguna acción y devuelve la vista definida por defecto (también creada a partir del *script* `rails generate controller`).

En la figura 12 se muestra el código de la vista `hola.html.erb` que, tal como vemos en la página que se nos ha mostrado, la encontraremos en el directorio `app/views/muestra`.

Figura 12. Código de la vista `hola.html.erb`

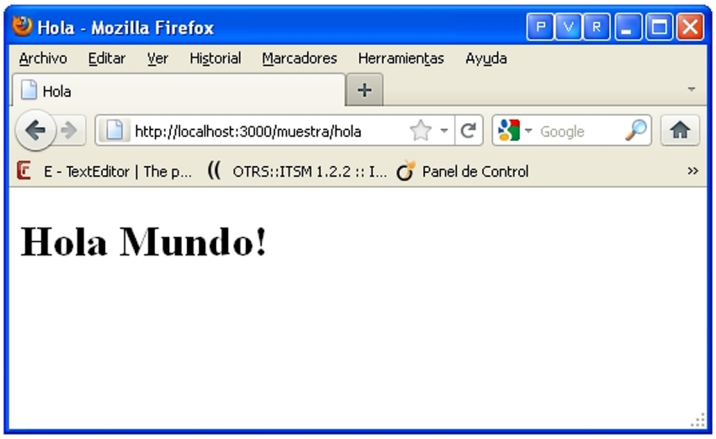


Como se puede observar en la figura 12, la vista contiene las etiquetas HTML que se muestran en el navegador web. Para lograr este objetivo, se tiene que sustituir el código anterior por el siguiente:

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

Si se actualiza la ventana del navegador, aparecerá lo siguiente (figura 13):

Figura 13. Página web de la vista `hola.html.erb`



Con ello se ha obtenido el resultado esperado. Así pues, se va a dar un paso más añadiendo a la vista código Ruby como el que hemos estudiado en el subapartado anterior.

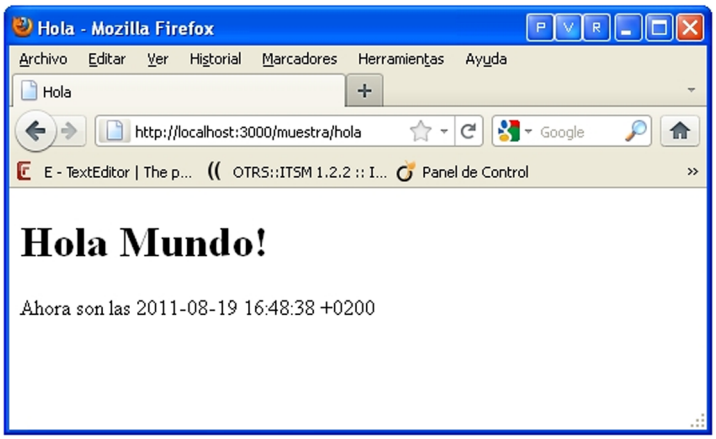
Para empezar, se añadirá a la vista anterior el siguiente código:

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

Con el anterior código se ha insertado código Ruby en la vista que llama al método `now` de la clase `Time` (el intérprete detecta el código, ya que está insertado entre los símbolos "`<%`" y "`%>`").

Si se actualiza el navegador, se obtiene el resultado siguiente (figura 14):

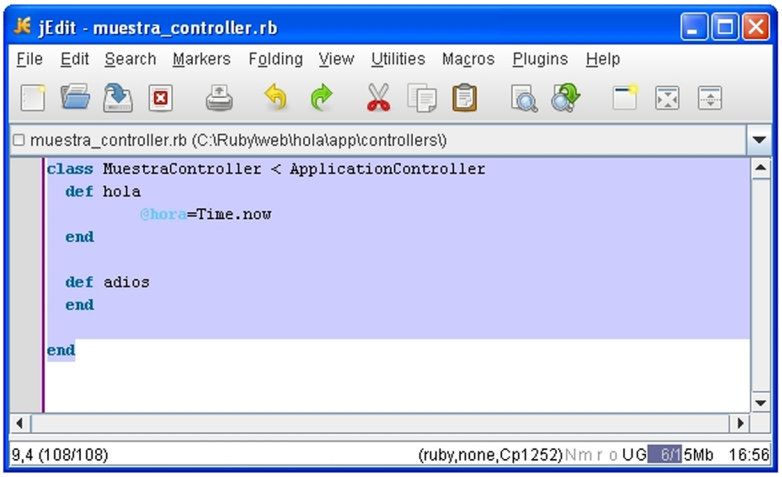
Figura 14. Página web actualizada de la vista `hola.html.erb`



El código anterior es correcto, pero no sigue el patrón MVC. Esto es debido a que toda la lógica de la aplicación debe programarse en el controlador, dejando que la vista solo tenga la responsabilidad de mostrar información. Para ello se modificará el código anterior introduciendo en el método `hola`, una variable de instancia que contiene el valor de la hora en curso. Esta variable será consultada por la vista y se mostrará en la página web.

Con el argumento anterior, el código del controlador quedará de la forma siguiente (figura 15):

Figura 15. Código del controlador `Muestra`



Se ha definido la variable de instancia `@hora`. A continuación es necesario modificar la vista para que esta utilice el valor de esta variable, sustituyendo el uso de la clase `Time` por la variable:

```
Ahora son las <%= @hora %>
```

De esta manera se ha obtenido el mismo resultado, pero se ha aplicado el patrón MVC de forma estricta.

2. Conceptos clave de una aplicación Ruby on Rails

El patrón MVC existe en Ruby on Rails gracias a un conjunto de clases implementadas en Ruby que facilitan el cumplimiento del modelo. Por este motivo se va a estudiar sin entrar en detalle cada uno de los componentes de Rails.

En la web de Ruby on Rails está disponible la documentación completa del API de Rails, donde se puede consultar con el mayor detalle cada una de las clases, los métodos y las propiedades que definen el conjunto de Rails.

2.1. `ActiveRecord`

Se trata de la clase que implementa la parte "Modelo" del patrón MVC.

Sus principales funciones son la implementación del mapeo entre los objetos de Ruby on Rails y las tablas de las bases de datos relacionales (también conocido como ORM); y en segundo lugar, proporciona los métodos CRUD[12] de forma muy cercana al lenguaje natural (sin utilizar el lenguaje SQL).

(12)Sigla de la expresión inglesa correspondiente a *crear*, *leer*, *modificar* y *eliminar*.

`ActiveRecord` evita que el programador de Ruby on Rails trabaje directamente con la base de datos, con las tablas y las columnas. Evita que tenga que utilizar sentencias complejas del lenguaje SQL y esto es gracias a que todo el trabajo lo implementa la propia clase de forma transparente.

En los subapartados siguientes se presentan los conceptos básicos de `ActiveRecord`, aunque además de realizar la función de enlace entre la instancia de una clase y la fila de la tabla donde se encuentran los datos, también proporciona métodos que permiten gestionar las relaciones entre objetos, métodos que permiten controlar el ciclo de vida de los objetos y métodos que gestionan transacciones (agrupaciones de cambios en la base de datos que deben realizarse de forma conjunta, es decir, todos los objetos se actualizan en una misma transacción o esta se cancela).

Reflexión

Algunos de estos puntos sobrepasan el alcance de este módulo, por lo que, si se desea ampliar conocimiento, se puede consultar la página oficial de la API tal como se ha planteado en la introducción del apartado.

2.1.1. De la clase a la tabla

El programador define un modelo con sus atributos o propiedades y `ActiveRecord` transforma este modelo en una tabla de una base de datos relacional.

Ruby on Rails establece el convenio en el que los modelos o clases se nombran en singular y siempre con la primera letra en mayúscula, mientras que las tablas asociadas se pluralizan y cambian a minúscula la primera letra.

Ejemplo

Si se define la clase `Menu`, se definirá la tabla asociada `menus`.

Cada instancia de la clase (cada objeto) corresponde a una fila de la tabla y cada columna de la tabla corresponde a un atributo. A continuación se creará la primera clase en un nuevo proyecto que se llamará `Project` y que se utilizará para practicar los conceptos presentados a lo largo del apartado.

La clase que vamos a crear será `Equipo`, cuyas propiedades o atributos serán `nombre`, `estadio` e `historia`, de forma que los dos primeros son del tipo cadena de texto y la última será un campo con mayor capacidad.

Para crear la clase se va a utilizar un andamio[13], que no es más que un ayudante que a partir de unos pocos parámetros es capaz de crear las estructuras y ficheros necesarios para la aplicación. Para ello vamos a crear un nuevo proyecto `laboratorio` sobre el que realizaremos todas las pruebas de estos subapartados:

(13)En inglés, *scaffold*.

```
rails new laboratorio
```

A continuación nos situamos en la carpeta raíz del proyecto recién creado y ejecutamos la sentencia siguiente:

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

Si se observa la salida del comando, este invoca `activeRecord` y crea un conjunto de ficheros. El primero, creado en el directorio `db/migrate`, tiene el contenido siguiente:

```
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 fichero define la clase `CreateEquipos` como subclase de `Migration`, que es a su vez una subclase de `ActiveRecord`. Este fichero implementa el proceso ORM, ya que se define la tabla `equipos` con sus tres columnas que corresponden a la clase `Equipo`.

Por otra parte, se ha creado el fichero `equipo.rb` en la carpeta `app/models`, que define la clase `Equipo`, aunque inicialmente está vacía de contenido:

```
class Equipo <ActiveRecord::Base
end
```

También se ha creado el controlador `equipos_controller` y una vista `equipos`.

Reflexión

Los ficheros `equipos_controller` y `equipos` serán objeto de estudio en los siguientes apartados.

El siguiente paso es la ejecución del comando que efectúa la migración en la consola de comandos abierta:

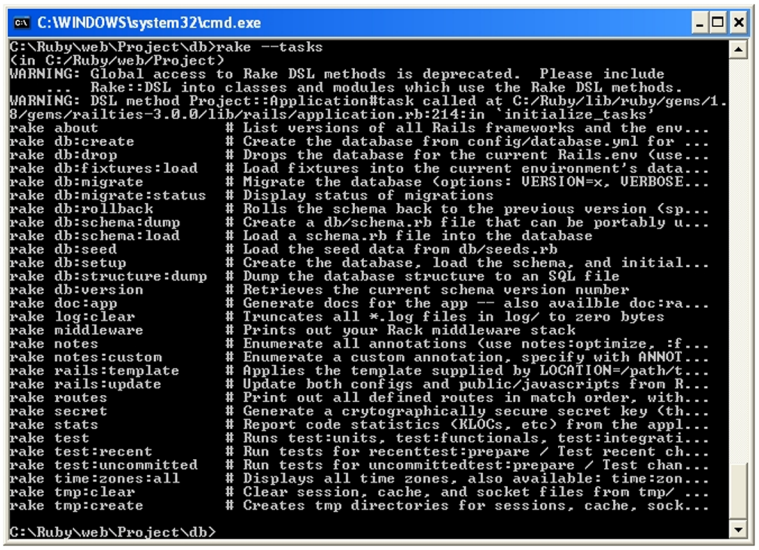
```
rake db:migrate
```

La herramienta `Rake`

`Rake` es una utilidad de Ruby on Rails similar al comando de Unix `make` y utiliza un fichero `Rakefile` para crear una lista de tareas. En Rails, `Rake` se utiliza para implementar tareas de administración de una cierta complejidad.

Es posible obtener una lista de las tareas `Rake` ejecutando el comando `rake -tasks`, de forma que se obtiene una lista con la descripción de cada una de las tareas (pero esta depende del directorio donde se ejecute el comando).

Figura 16. Salida del comando `rake -tasks`

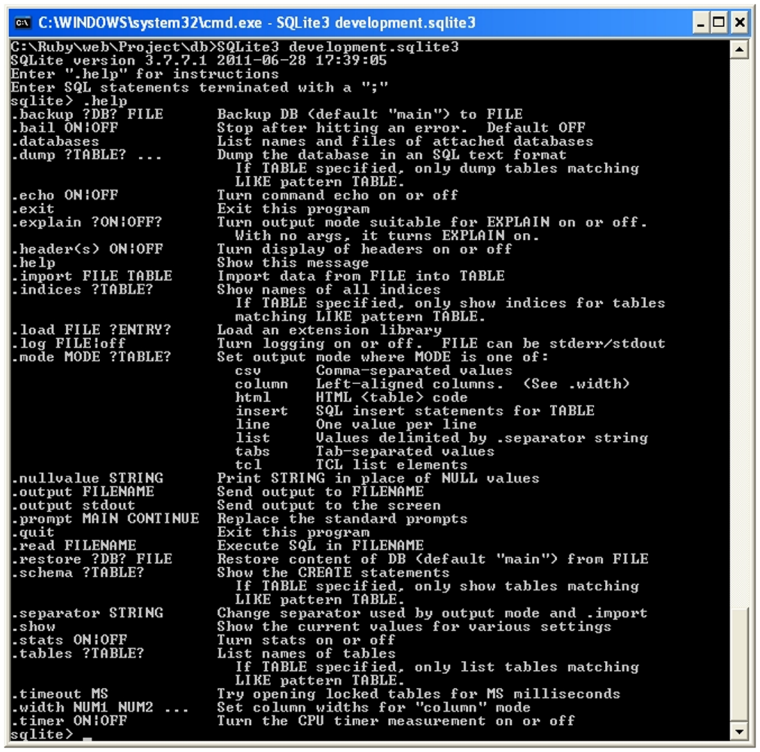


La ejecución del comando anterior ha creado una base de datos SQLite3 en el directorio `db` del proyecto, el nombre del fichero es `development.sqlite3`. Además se ha creado una tabla `Equipo` con las columnas `nombre`, `estadio` e `historia`.

Se puede consultar la base de datos con un par de comandos básicos de SQLite3. Situando la consola en el directorio `db`, al ejecutar el comando siguiente, el gestor SQLite3 selecciona la base de datos recién creada y dentro del gestor se puede ejecutar el comando `.help`, que muestra las distintas opciones disponibles (figura 17).

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

Figura 17. Opciones disponibles en la consola de SQLite3



El comando `.tables` lista el conjunto de tablas que existen en la base de datos, de forma que al ejecutarlo se obtiene la tabla `equipos`. Por otra parte, si se ejecuta la sentencia `.schema equipos`, esta devuelve la sentencia SQL que se ejecutó para crear la tabla `equipos`.

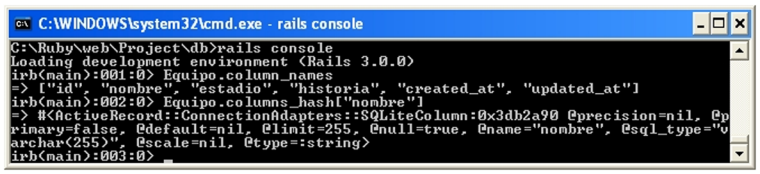
Hasta este momento, con dos simples comandos se ha creado la clase `Equipo` con sus tres propiedades (mediante el comando `rails generate scaffold`), a continuación se ha creado la base de datos y la tabla `equipos`, que almacena las instancias de la clase `Equipo` utilizando la sentencia `db:migrate`.

Una alternativa para consultar los atributos creados es la consola de Rails, de forma que se consultan los atributos de una clase e incluso las características de estos en la base de datos:

```
rails console
```

En el momento en que se abre la consola, el comando `Equipo.column_names` devuelve un *array* con los distintos campos de la tabla. Además el comando `Equipo.columns_hash["nombre"]` devuelve la lista de las características que definen la columna `nombre` de la tabla `equipos`.

Figura 18. Consulta de las clases con la consola de Rails



Si se observa la definición de la tabla, esta dispone de tres columnas adicionales que han sido creadas automáticamente:

- `id`: que se trata de una columna autonumérica que es la clave principal de la tabla.

- `created_at` y `updated_at`: que son campos que almacenan el momento de creación de la fila y el de la última actualización.

2.1.2. Crear, modificar, leer y borrar

Tal como se ha presentado en el subapartado anterior, una de las principales características que ofrece `ActiveRecord` es la posibilidad de manipular registros de la base de datos sin necesidad de utilizar sintaxis SQL. A continuación se presentan las distintas técnicas que hacen posible implementar las acciones de crear, modificar, leer y borrar registros de una base de datos.

Crear

En primer lugar, la creación de un registro en la tabla es equivalente a la acción de crear una instancia de la clase; es decir, al crear un objeto, se crea una fila de la base de datos. En la consola de Rails arrancada en el anterior subapartado se introduce el código siguiente:

```
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
```

Este código provoca la creación de un objeto `equipo`, que se puede consultar a través de la siguiente URL `localhost:3000/equipos` (recordemos que se debe arrancar el servidor web en el directorio `laboratorio` para poder ejecutar la nueva aplicación).

Figura 19. Creación de instancias de una clase

En Ruby existen distintas alternativas sintácticas para el código anterior. En primer lugar, utilizando un bloque de código se evita el uso de 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 válida es utilizar un *array* asociativo, de modo que esta forma sintáctica es útil si se están almacenando valores a partir de un formulario 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
```

Para finalizar, Ruby dispone del método `create` para crear nuevos objetos. La diferencia con `new` es que no necesita la llamada al método `save` para almacenar los valores en la base de datos. Con el método `create` los valores se almacenan de manera automática.

Leer

La lectura de registros de la base de datos o de una tabla en concreto se basa en la identificación de los registros que se quieren obtener. Para ello, Ruby on Rails dispone del método `find`. A continuación se revisan las distintas maneras en las que se puede utilizar este método.

Ejemplo

En el siguiente ejemplo se busca el registro o el objeto cuyo valor en la clave principal es el valor 1:

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

Con el código anterior, la consola devuelve la asignación siguiente:

```
=> #<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 es nada habitual buscar un registro a partir de su clave principal, ya que esta es creada por el sistema. Normalmente las búsquedas se basan en los valores de ciertos atributos del objeto que el usuario conoce.

Las búsquedas a partir de valores de atributos se implementan a partir del método `find_by_`*`atributo`*, donde *`atributo`* se sustituye por el nombre de este. Se obtiene el mismo resultado que en el ejemplo anterior si se utiliza la sintaxis siguiente:

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

Esta sentencia asigna a la variable `equipo_Ondara` el registro o instancia de clase u objeto cuyo nombre tiene el valor "Ondarense". Además, es posible añadir una nueva cláusula mediante el comando `and` junto con la nueva condición.

Ejemplo

La sentencia siguiente almacena en la variable `equipo_Ondara2` el equipo cuyo nombre tiene el valor "Ondarense" y cuyo estadio tiene el valor "Vicente Zaragoza":

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

Cuando se busca un registro y este no existe, Ruby on Rails crea y devuelve la excepción `RecordNotFound`, aunque es posible modificar este comportamiento introduciendo el carácter "`!`" entre la llamada al método `find` y el paréntesis que contiene los parámetros.

Ejemplo

La sentencia siguiente provoca que se asigne a la variable `equipo_5` el valor `Nil` si no existe ningún registro cuya clave principal es el valor 5:

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

Por su parte el método `find_last_by_` devuelve el último de los registros u objetos si en la búsqueda se han encontrado varios.

Ejemplo

La sentencia siguiente asigna el último de los equipos cuyo nombre es "Ondarense" a la variable `equipo_u`:

```
equipo_u = Equipo.find_last_by_nombre("Ondarense")
```

Pero si se quieren obtener todos los objetos que cumplen una condición se utiliza la setencia `find_all_by_`.

Ejemplo

Si se crea una nueva instancia con un nuevo estadio del equipo "Ondarense", la sentencia siguiente asignará a `estadios` un *array* con los objetos o instancias cuyo nombre es "Ondarense".

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

Aunque Ruby on Rails permite utilizar sintaxis SQL en las búsquedas de registros no es recomendable, ya que puede ser un foco de errores en el código. Además con el uso de la sintaxis anterior se asegura que el código sea funcional en todos los sistemas de base de datos, independientemente de estos.

Modificar

La modificación de objetos se basa en la composición de las acciones anteriores. En primer lugar se localiza el objeto a modificar (utilizando el método `find`), a continuación se asigna el nuevo campo o atributo con el nuevo valor y el proceso finaliza con el método `save`.

Ejemplo

En el ejemplo siguiente se selecciona el objeto cuyo valor en el atributo nombre es "Ondarense", se modifica el valor del campo nombre por "Valencia" y se guardan los cambios:

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

También es posible realizar modificaciones utilizando el método `update`, que actualiza un registro, y el método `update_all`, que actualiza todos los registros que cumplen una cierta condición.

Ejemplo

Siguiendo con el ejemplo anterior, si el registro con el atributo `nombre` con el valor "Ondarense" tiene el valor 1 en la clave principal, se podrá sustituir por la siguiente sentencia:

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

En el caso de que se quieran sustituir todos los objetos con valor "Ondarense" en el atributo nombre por "Barcelona" se utilizará la siguiente sentencia:

```
equipoM = Equipo.update_all("nombre = 'Barcelona'", "nombre like 'Ondarense'")
```

Eliminar

La eliminación de registros u objetos se puede realizar de dos formas distintas:

- Con los métodos `delete` y `delete_all` se realiza la eliminación de los registros de la base de datos.

- Los métodos `destroy` y `destroy_all` ejecutan el método `destroy` del objeto o de los objetos que se van a destruir.

La ventaja de la segunda sintaxis es que en la definición de los métodos de los objetos se pueden incorporar ciertas acciones o validaciones que pueden añadir una cierta seguridad al proceso de eliminación (uno de los más críticos en cualquier aplicación).

Ejemplo

En el siguiente ejemplo se elimina el objeto con el atributo estadio con el valor "Terra" y a continuación todos los objetos con el nombre "Barcelona":

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

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

Por lo que se han eliminado los dos registros que se tenían creados originalmente con el nombre "Ondarense".

2.2. `ActionDispatch`

`ActionDispatch` es el primero de un triplete de elementos que definen lo que se conoce como `ActionPack`. En los próximos tres apartados se estudian los tres componentes cuyo objetivo se define brevemente a continuación:

- `ActionDispatch` es el responsable de hacer llegar cada petición realizada desde el navegador del usuario al controlador adecuado.

- `ActionController` recibe las peticiones, ejecuta las acciones y, cuando finalizan, devuelven el control a la vista.

- `ActionView` recibe el control desde el controlador y aplica el formato adecuado a la respuesta que es enviada de respuesta al usuario.

Por lo que son responsables de la lógica del proceso MVC, definiendo los flujos de trabajo de estos. En los dos siguientes apartados se estudiarán `ActionController` y `ActionView`, siguiendo de este modo la misma secuencia que se produce en el flujo de trabajo de una aplicación Ruby on Rails.

2.2.1. Enrutamiento estándar

En los subapartados anteriores se ha creado una aplicación "Hola Mundo". A continuación se ha aplicado la técnica que ha hecho posible ejecutar cada una de las acciones del controlador.

Ved también

Podéis ver la aplicación "Hola Mundo" en el subapartado 1.4, y las técnicas de ejecución de las acciones del controlador en el subapartado 1.4.3 de este módulo didáctico.

¿Dónde se encuentran las instrucciones que le indican a `ActionDispatch` que ejecute el código X cuando se realiza la llamada Y? La respuesta es sencilla. Existe en el directorio `config` del proyecto el fichero `routes.rb`, que contiene las siguientes sentencias, donde se indica que el controlador `Muestra` dispone de dos acciones definidas por el programador: `hola` y `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 código anterior permite que el usuario pueda ejecutar cada una de las acciones desde el navegador a partir de la composición de la URL indicada en el navegador (para cargar las siguientes páginas es necesario arrancar el servidor web en el directorio `hola`):

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

El mecanismo anterior define las rutas de forma simple y comprensible, pero estas pueden ser modificadas por el propio programador.

2.2.2. Enrutamiento a medida

El enrutamiento se basa en la combinación de los métodos HTTP (`GET`, `PUT`, `PUSH` y `DELETE`) con los distintos nombres que identifican los recursos de la aplicación.

Utilizando la aplicación de los equipos de futbol, se va a analizar algunas de las posibilidades que ofrece Ruby on Rails para la personalización del enrutamiento. Si consultamos el fichero `routes.rb` del proyecto laboratorio se obtiene:

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

El comportamiento de las rutas es el siguiente:

- Utilizando `GET` y la ruta que apunta hasta el controlador o recurso, se obtiene una lista de todos los objetos de ese recurso (`http://localhost:3000/equipos`).

- Utilizando `GET` y la ruta que apunta hasta el controlador pero indicando a continuación la clave principal de un objeto, se obtiene el contenido de uno de los equipos (`http://localhost:3000/equipos/1`).

- Utilizando `POST`, la ruta que apunta hasta el controlador y los datos que definen un equipo, se crea un nuevo equipo con los atributos indicados.

- Utilizando `PUT`, la ruta que apunta hasta el controlador y los datos que identifican y modifican el objeto, este será modificado con los nuevos valores.

- Utilizando `DELETE`, la ruta que apunta hasta el controlador y la clave principal de un objeto, este será eliminado.

Conocidas las bases del protocolo que permite la comunicación de las distintas acciones, se va a profundizar en las distintas opciones que aparecen en la página que lista todos los equipos.

La diferencia entre las rutas del proyecto `hola` y `laboratorio` es que este último dispone de una única sentencia:

```
resources :equipos
```

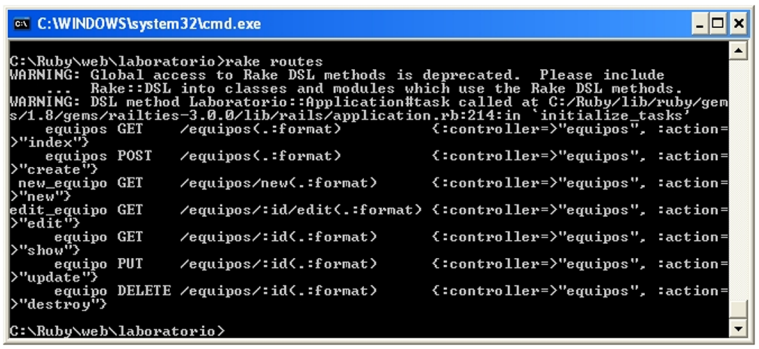
Esta sentencia especifica que el controlador `equipos` dispone de siete rutas estándar, que se pueden consultar con el comando `rake routes` desde el directorio raíz de la aplicación laboratorio (figura 20).

La respuesta de este comando es fácil de interpretar: muestra cada una de las rutas creadas en la aplicación. En cada una de las rutas se identifica el controlador y la acción que se lleva a cabo. Todo esto ha sido creado por el andamio, que generó automáticamente el código en el controlador con las siete acciones estándar.

Ved también

Podéis ver el andamio que se utilizó en el subapartado 2.2.2 de este módulo didáctico.

Figura 20. Resultado del comando `rake routes`



El nombre de las acciones estándar identifica cuál es su función, por lo que son predecibles:

- `index`: muestra el listado con los objetos existentes.
- `create`: crea un nuevo objeto a partir de la información en la llamada `POST`.
- `new`: crea un objeto nuevo pero no lo almacena, lo pasa al cliente. Sería análogo a crear un formulario vacio para que sea rellenado.
- `show`: muestra el contenido de un objeto identificado por el parámetro `id`.
- `update`: actualiza el contenido del objeto identificado por el parámetro `id`.
- `edit`: muestra el contenido del objeto identificado por el parámetro `id` en un formulario listo para ser editado por el usuario.
- `destroy`: elimina el objeto identificado por el parámetro `id`.

Estas acciones estándar se pueden modificar. Por ejemplo, si las dos últimas no fueran necesarias, se podría indicar en el fichero `routes.rb` de la siguiente manera:

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

Conocido el mecanismo por el que se realiza el enrutamiento de las acciones de un controlador concreto, es imprescindible que estas acciones se encuentren definidas en el propio controlador.

En el fichero `equipos_controller.rb` se encuentra el código creado automáticamente que define cada una de las siete acciones estándar totalmente funcionales:

```
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 mayoría de las acciones hay un bloque de código `respond_to`, donde, dependiendo de la variable indicada por el cliente, `format`, se dará respuesta en formato html o xml, con lo que se han definido dos tipos de respuesta html y xml.

En la primera de las acciones definidas se observa lo siguiente:

```
def index
   @equipos = Equipo.all

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

En la línea `@equipos = Equipo.all` se asigna a la variable `equipos` todos los objetos, de forma que se convierte en un *array* con cada una de las instancias creadas de la clase `Equipo`. A continuación, si el formato de la respuesta indicada en la variable `format` es HTML, se llama a la vista `index.html.erb`, cuyo contenido es el siguiente:

```
<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 código crea una tabla HTML y utiliza código Ruby en la creación de un bucle, en el que se va creando cada fila de la tabla con la información de cada registro de la tabla (o cada atributo de los objetos).

Es posible añadir una nueva acción utilizando una extensión en la definición en el fichero `routes.rb` con las sentencias siguientes, de forma que se está indicando que existirá una nueva acción `goles_a_favor`, donde el parámetro `member` indica que la acción se aplica a cada uno de los objetos:

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

Evidentemente la acción deberá estar definida en el controlador.

Además de la opción anterior es posible definir una acción que aplique a todos los equipos en conjunto, en este caso se utilizaría `collection` en la definición.

En los subapartados siguientes se introduce con cierto detalle la estructura de los controladores y de las vistas, y se finaliza de esta forma el estudio de los tres componentes del `ActionPack`.

2.3. `ActionControler`

En este subapartado se presenta el comportamiento y la sintaxis que se utiliza en los controladores. Se explica la relación de estos con las vistas, cómo se procesan plantillas, cómo se envían ficheros desde el servidor al usuario y cómo se puede redireccionar a una URL desde el controlador.

Reflexión

Para profundizar en la clase `ActionControler` deberéis consultar la API de Ruby on Rails.

2.3.1. Comportamiento del controlador

En primer lugar, cuando el controlador recibe una petición, prepara el entorno de ejecución poniendo a disposición todos los parámetros de la llamada y del entorno para que puedan ser utilizados en la ejecución de la acción.

Entre estos parámetros se encuentran los siguientes: el nombre de la acción y los parámetros de esta; las *cookies*, si existen; los detalles de la llamada (método `http`, `ip` y puerto de origen y características ssl), y las cabeceras http.

En la mayoría de aplicaciones solo se utilizan los parámetros de la llamada, pero es importante conocer que también hay la posibilidad de disponer de cierta información sobre la llamada realizada por el usuario.

En segundo lugar, cuando un controlador recibe una petición, se produce el siguiente flujo en la ejecución:

- se busca el método con el mismo nombre que el que recibe el controlador. Si este método existe, se le pasa el control, pero si no existe, el método:
- se busca un método identificado como `method_missing`. Si se encuentra, se ejecuta, pero en caso contrario:
- se busca una plantilla identificada con el nombre del controlador; si esta es encontrada, se devuelve al usuario; pero en caso contrario:
- el controlador devuelve el error `Unknown Action` al usuario.

En tercer lugar se ejecutará el código definido en la acción definida y se generará la respuesta, que puede ser de alguno de los siguientes tipos:

a) En el 99% de los casos, la respuesta es una llamada a una plantilla o una vista que utiliza datos procesados por el método o la acción para construir el resultado que se envía al usuario.

Ejemplo

En el ejemplo del final del apartado 1.4.3 de este módulo se realiza una llamada a la vista `hola.html.erb` que utiliza el valor de la variable calculada en el controlador.

b) En algunos casos la notificación de errores se realiza con la devolución de una cadena de texto simple al cliente.

c) Cuando se está programando con el paradigma AJAX, en algunas ocasiones una petición del cliente no dispone de una respuesta por parte del navegador. Por tanto, también es posible no devolver contenido al cliente; pero como es imprescindible una respuesta, se devuelven un conjunto de cabeceras HTTP.

d) Existen escenarios donde se tiene que entregar un fichero al cliente. Sería el ejemplo de ficheros pdf o jpg creados o almacenados en el servidor web.

2.3.2. Procesamiento de plantillas

Las plantillas son los ficheros que definen la respuesta del servidor. Ruby on Rails dispone de tres tipos de plantillas:

- erb: que está formado por código html y código Ruby embebido.
- builder: que es un mecanismo con el que se crean documentos XML.

- rjs: que se encarga de generar código JavaScript.

Los nombres de las plantillas son creados siguiendo la estructura siguiente:

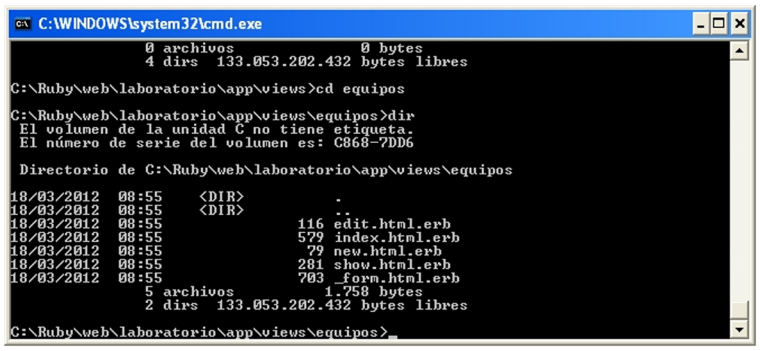
```
app/views/id_controlador/id_accion.tipo.xxx
```

Esta estructura sigue realmente el siguiente convenio estipulado por Ruby on Rails:

- `id_controlador`: identifica el controlador.
- `id_accion`: identifica la acción.
- `tipo`: identifica el tipo de fichero final (html, js o atom).
- `xxx`: define el tipo de fichero final (erb, builder o rjs).

En el caso del controlador equipos se tienen las plantillas siguientes:

Figura 21. Listado de las plantillas del controlador `equipos`



La llamada a una vista o plantilla se realiza utilizando el método `render`, que dispone de un conjunto de parámetros opcionales. En general, el comportamiento de este método es el siguiente:

a) `render()`: si no se especifica ningún parámetro, el método `render` llama a la vista del controlador asignada a la acción donde se ejecuta la sentencia `render`.

Ejemplo

En el siguiente ejemplo, como no se ha establecido ningún método sobre la acción `index` que hay en la segunda línea, se interpreta que se debe ejecutar `render` sobre la vista con el mismo nombre que la acción:

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

Es decir, Ruby on Rails considera que si no aparece ningún método en la definición de la acción, el método por defecto es `render` y lo ejecutará (aunque no se encuentra en la acción). Por el contrario, en la tercera línea, donde se estipula la acción si el formato es xml, sí se hace referencia al método `render` porque se tienen que introducir parámetros, ya que la respuesta no será la vista estándar.

b) `render(:text => string)`: en esta llamada se envía al navegador el texto plano definido en la variable `string` sin realizar ningún tipo de interpretación del tipo HTML.

Ejemplo

Si el código del ejemplo anterior se modifica por el siguiente, cuando se ejecute la acción `index` del controlador `Equipos` la respuesta del usuario será una página web con el texto "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 este tipo de llamada se indica que la cadena especificada en el parámetro `inline` es el código de una plantilla o vista cuyo tipo se especifica mediante el parámetro `type` y finaliza con la indicación de la posibilidad de utilizar variables locales.

Ejemplo

El siguiente ejemplo define un método por defecto de un controlador. El comportamiento es que si el controlador es llamado con un identificador de una acción que no existe, devuelve una respuesta al usuario con el nombre de la acción y sus parámetros.

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

d) `render(:action =>nombre_accion)`: devuelve al usuario la plantilla o vista por defecto de la acción que es pasada en el parámetro.

e) `render(:file => path, [ :use_full_path => true|false ], [:locals => hash])`: devuelve la plantilla que se indica en la ruta especificada por el parámetro `file`. Por defecto, se define una ruta absoluta.

f) `render(:template => name, [:locals => hash])`: devuelve la plantilla identificada en el parámetro `template` (que debe contener el nombre del controlador y de la acción).

Ejemplo

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

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

Reflexión

Las plantillas parciales se explicarán más adelante en este mismo módulo.

h) `render(:nothing => true)`: devuelve una página HTML vacía al navegador del usuario.

Como se puede observar, la sintaxis del método `render` es muy amplia, pero con las opciones presentadas se contemplan la mayoría de las necesidades que van a surgir en una aplicación web de complejidad media.

2.3.3. Envío de ficheros

Existen dos métodos que permiten el envío de ficheros desde el servidor al cliente:

- `send_data`, que envía una cadena con datos binarios.
- `send_file`, que envía un fichero.

La sintaxis básica de cada uno de estos métodos se presenta a continuación.

a) `send_data`

El método tiene la siguiente sintaxis, donde la variable *`cadena`* es el fichero binario (por ejemplo, una imagen jpg o cualquier tipo de fichero binario):

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

Pero, además, hay una serie de opciones que definen las características de la transmisión del fichero:

- `:disposition` *`inline/attachment`*: cuando la opción seleccionada es `inline`, el navegador mostrará el fichero cargado, mientras que si es la opción `attachment`, el navegador muestra la ventana de descarga del fichero.

- `:filename` *`nombre`*: especifica el nombre por defecto que el navegador dará al fichero cuando este sea almacenado en el equipo cliente.

- `:type` *`cadena`*: define el tipo de fichero, que por defecto es: `application/octet-stream`.

b) `send_file`

La sintaxis del método es la siguiente, donde la variable *`ruta`* indica la ruta del fichero que se quiere enviar:

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

Y de la misma forma que el método anterior, `send_file` dispone de las siguientes opciones:

- `:buffer_size` *`número`*: donde se indica el tamaño del fichero enviado al navegador en cada escritura, si la opción de *streaming* está habilitada.

- `:disposition` *`inline/attachment`*: cuando la opción es `inline`, el navegador mostrará este fichero; y si la opción es `attachment`, indica que el navegador ofrecerá la descarga del fichero.

- `:filename` *`nombre`*: especifica el nombre por defecto que el navegador dará al fichero cuando este sea guardado. Si no se especifica, se utilizará el mismo nombre que tiene en la ruta original.

- `:stream` *`true/false`*: activa el *streaming* en el envío de ficheros.

- `:type` *`cadena`*: define el tipo de fichero, que por defecto es `application/octet-stream`.

Ejemplo

Si se modifica la acción `index` del controlador `equipos` con el siguiente código, el navegador envía al usuario el fichero `rails.png` que se encuentra en el directorio `public/images` del proyecto:

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

2.3.4. Redirección a una URL

En Ruby on Rails, la redirección se utiliza cuando la acción solicitada por el cliente no puede ser procesada y se necesita la ayuda de otra acción. Por ejemplo, puede darse el caso de que por algún motivo en la ejecución del código de una acción se necesite ejecutar otra acción.

La sintaxis del método `redirect_to` es la siguiente:

- `redirect_to(:`*`action => ..., opciones...`*`)`: realiza una redirección a la acción especificada en el parámetro *`action`*.

- `redirect_to(`*`path`*`)`: realiza la dirección a la ruta indicada como parámetro.

- `redirect_to(:back)`: realiza una redirección a la URL indicada en la cabecera `HTTP_REFERER` de la llamada.

Este método se utiliza en la definición de las acciones `create`, `update` y `destroy` del controlador `equipos`. Por ejemplo, en la acción `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
```

Después de crear el equipo en la primera línea de código de la acción:

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

se comprueba que se ha guardado mediante el condicional:

```
if @equipo.save
```

y se realiza la redirección:

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

donde se llama a la acción `show`, que muestra el equipo recién creado y además el mensaje: "Equipo was successfully created.".

Se trata de una forma rápida de ejecutar la sentencia siguiente, pero que además de implementar la redirección, añade la cabecera con el mensaje que indica que el equipo se ha creado con éxito:

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

2.4. ActionView

`ActionView` proporciona la funcionalidad necesaria para crear las plantillas que son transformadas en página HTML, ficheros XML o JavaScript y que son enviadas al cliente como respuesta a la acción solicitada. En este subapartado se estudian algunas de la opciones disponibles en las plantillas, algunos ayudantes de Rails con los que se crean sencillos formularios, mecanismos para subir ficheros del cliente al servidor y el uso de parciales que proporcionan la funcionalidad AJAX en Ruby on Rails.

2.4.1. Plantillas eRB

Uno de los convenios básicos de Ruby on Rails es la localización de los ficheros en la estructura de directorios. Las plantillas se encuentran en el directorio :`app/views`, pero además, dentro de este directorio se crea un directorio por cada uno de los controladores de la aplicación. En el interior de estos directorios cada plantilla es un fichero erb con la siguiente especificación en el nombre *`nombreaccion`*`.html.erb`.

En la figura 21 se observa cada una de las plantillas de las acciones del controlador `equipos` situadas en `app/views/equipos`.

Las plantillas están formadas generalmente por código HTML y Ruby, aunque –como ya se ha visto– existe la posibilidad de crear vistas cuyo contenido sea JavaScript o XML.

El código Ruby se delimita utilizando el siguiente delimitador `<%= ....%>`, de forma que se crean páginas HTML con sentencias Ruby insertadas en su interior y que son interpretadas antes de ser enviadas al cliente.

Una de las características importantes de las plantillas es que el código Ruby de una vista tiene acceso al entorno del controlador, es decir:

- Puede acceder a todas las variables definidas en el controlador. Este es el principal mecanismo de comunicación entre la vista y el controlador. En el controlador se implementa la lógica de la aplicación y el resultado se almacena en variables que son utilizadas por la vista para componer la página de resultados.

- Además, el objeto controlador es accesible desde la vista, lo que permite ejecutar desde la vista cualquiera de los métodos públicos definido en el controlador.

En el ejemplo "Hola Mundo" que hemos presentado anteriormente, se creó una vista con código Ruby insertado:

Ved también

Ved el ejemplo "Hola Mundo" en el subapartado 1.4.3.

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

En el código Ruby insertado se ejecuta el método `now` de la clase `Time`, de forma que el resultado es convertido en una cadena y aparece en la página HTML.

Se trata de un ejemplo sencillo, pero es posible insertar código tan complejo como sea necesario. Sin embargo, tal como se comentó, es una buena práctica imponerse la lógica de negocio y utilizar para la implementación de los controladores y las vistas el mínimo código Ruby que permita presentar los resultados.

En el ejemplo anterior el símbolo = es el responsable de la conversión a cadena y el retorno como texto del resultado del código que tiene a continuación, por lo que si se quiere ejecutar código Ruby que no devuelve ningún valor, simplemente se omite el símbolo =.

Si en la vista `hola.html.erb` del ejemplo "Hola Mundo" se sustituye su contenido por el siguiente, la primera sección de código no devuelve ningún valor al cliente y es equivalente a realizar la asignación de la variable en el controlador, mientras que en la segunda se devuelve la fecha al usuario.

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

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

Sin embargo, tal como se ha comentado, es importante que un código como el anterior se encuentre en el controlador y no en la vista.

Una característica muy potente es la posibilidad de insertar etiquetas HTML entre código Ruby.

Ejemplo

El siguiente código implementa un bucle de dos pasos en el que en el navegador se escribe "Ei Ei".

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

Se puede crear un problema de seguridad debido a que Ruby envía directamente código que es interpretado por el navegador y como este interpreta código JavaScript además de las etiquetas HTML, puede darse el caso de que se envíe al navegador código JavaScript o etiquetas erróneas que han sido introducidas por un usuario intencionadamente (por ejemplo, en un campo de texto de un formulario).

En el caso anterior, como el contenido es enviado directamente a la página HTML, este sería interpretado por el navegador y podría crear ciertas acciones que modificaran el comportamiento o los objetivos iniciales de la aplicación.

Para evitar este problema, se utiliza el método `sanitize`, cuya traducción al español 'desinfectar', ya da unas pistas de cuál es su función: elimina etiquetas `<form>`, `<script>`, enlaces a código JavaScript, etc.

Por tanto, es recomendable desinfectar un resultado del código Ruby si el contenido de este ha sido generado inicialmente por un cliente.

2.4.2. Formularios a partir de ayudantes

En todas las aplicaciones web interactivas, el usuario o cliente en algún momento tiene que introducir información en un formulario. Ruby on Rails proporciona un conjunto de métodos que facilitan la creación de estos formularios.

Se utilizará el siguiente ejemplo para mostrar el potencial de los ayudantes de formularios.

Figura 22. Formulario de ejemplo creado con ayudantes

El código Ruby que construye el formulario es el siguiente:

```
<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 código anterior aparecen un conjunto de métodos que se conocen como *ayudantes* y cuya función es simplificar la creación de un formulario a partir de la estandarización. La descripción básica de función se explica a continuación:

- En la segunda línea `form_for` crea el formulario y lo asigna a la variable `form`, que es utilizada en el resto del código.

- En el ejemplo, las etiquetas se definen con el ayudante `label`, que se acompaña de la cadena que aparecerá en la etiqueta.

- Los campos de texto se definen con los ayudantes `text_field` y con `text_area` en caso de tratarse de un campo de texto con múltiples líneas. Además, el atributo `placeholder` define el texto que aparecerá por defecto en el interior del campo de texto.

- Los campos tipo `radio` (botones de opción) y `check` (casillas de selección) se crean con los ayudantes `check_box` y `radio_button`.

- Es interesante observar cómo se ha creado la lista desplegable `Prioridad`, por la sencillez con la que se especifican los valores de esta.

- Mediante el ayudante `date_select` se crean tres listas que permiten elegir el día, el mes y el año. De forma análoga, el ayudante `time_select` crea dos desplegables que recaban la hora y minutos para asignar la alarma.

Existen otros ayudantes para la construcción de formularios que no se han utilizado en el ejemplo anterior, como `search_field`, `telephone_field`, `url_field`, `email_field`, `number_field` y `range_field`, cuyas características pueden ser experimentadas de forma muy sencilla.

Posiblemente, la principal característica que aportan los ayudantes es que proporcionan una forma simple de implementar programación *cross-platform*, ya que nos aseguramos de que se construye un formulario que va a funcionar en todas las plataformas y que aprovechará las ventajas de cada una de estas.

¿Cómo y cuándo se utilizan los formularios? ¿Qué ocurre con los datos introducidos en el formulario? A continuación se presenta un ejemplo de flujo de ejecución en el que se produce la modificación de un objeto o instancia de clase:

- El controlador recibe una petición para editar un objeto y con la información recibida busca la instancia de la clase que representa el objeto.

- Localizado el objeto, el controlador llama una vista formada por un formulario cuyos campos contienen los valores del objeto que se quiere modificar. Se trata de la clásica `edit`.

- Cuando el usuario modifica los campos y pulsa el botón de enviar, los valores del formulario son enviados de nuevo al controlador en busca de la acción que guardará los nuevos valores. El controlador recibe los valores, los extrae y los asigna al *array* `params`.

- La acción encargada de guardar el objeto utiliza los parámetros de `params` para obtener a modificar y guardará los nuevos atributos.

Se debe tener en cuenta que los datos enviados en el método `POST` son transferidos al *array* `params`, pero si el nombre del parámetro tiene corchetes, se asume que este forma parte de una estructura compleja y se construye un nuevo *array* para almacenar este parámetro.

Por ejemplo si el formulario tiene el campo `:id = 123`, el *array* `params` tendrá el siguiente contenido: `{ :id => "123" }`, y en caso de que se tenga el campo `[nombre] = Mestalla`, el *array* `params` tendrá el siguiente contenido: `{ :campo => { :nombre => "Mestalla" }}`.

En el ejemplo del controlador `equipos`, se dispone de la acción `edit`, donde la vista `edit.html.erb` hace referencia a un parcial en el que se observa la creación del formulario utilizando ayudantes, de forma que el formulario del ejemplo es único y es utilizado por las vistas `new` y `edit`, ya que en el interior de estas se utiliza el método `render` para insertar el formulario definido como una vista parcial:

Ved también

Los parciales se estudian en el subapartado 2.4.3 de este módulo didáctico.

```
<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 siguiente código corresponde a la vista `new.html.erb`:

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

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

El siguiente código corresponde a la vista `edit.html.erb`:

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

<%= render 'form' %>

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

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

2.4.3. Uso de *layouts* y parciales

En cualquier aplicación web de cierta complejidad existen partes de esta como menús o zonas donde el código se repite en distintas partes de la aplicación. Pero con los conceptos presentados hasta ahora, cada vista deberá contener todo el código que se presenta al usuario, por lo que se tendrá que repetir todo el código que define las cabeceras, los menús o la "cesta de la compra" y esto complica el mantenimiento de la aplicación (un pequeño cambio en el menú implicará cambiar todas las plantillas donde este se haya insertado). El uso de *layouts* y parciales aportan un mecanismo para resolver este problema.

Ejemplos de código que se repite

En la mayoría de aplicaciones existen cabeceras o menús que se mantienen a lo largo de la aplicación. En el caso particular de aplicaciones de comercio electrónico, la "cesta de la compra" se repite en las distintas páginas por las que el cliente va navegando mientras realiza las diferentes transacciones.

Layouts

Los *layouts* son páginas que contienen el diseño de una página HTML, es decir, incluyen las secciones `html` y `css` que definen el aspecto de la página que va a ser enviada al cliente. Pero en la sección `body` se inserta la vista de la acción que ha sido llamada por el controlador. El flujo de ejecución es el siguiente:

- Cuando finaliza la ejecución de una acción, el controlador llama a la vista asociada.
- La vista es renderizada y se llama al *layout* asociado, donde la vista es insertada.
- Se devuelve al cliente el *layout* en cuyo cuerpo se encuentra la vista.

Este funcionamiento no es nuevo. Aunque no hemos hecho referencia a él con anterioridad, ya se ha utilizado, por ejemplo, cuando se ha creado la clase `Equipo` mediante el andamio.

Ved también

Podéis ver el andamio que se utilizó en el subapartado 2.2.2 de este módulo didáctico.

En el directorio `app/views/layouts` existe un fichero `application.html.erb` que define el *layout* por defecto, cuyo contenido es el siguiente:

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

      <%= yield %>
```

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

El objetivo de este *layout* es definir la página HTML en cuyo cuerpo se mostrará el resultado de cada una de las plantillas asociadas a las acciones de la clase `Equipo` (`index`, `edit`, `show` y `new`). El método `yield` es el encargado de realizar la carga de la plantilla de la acción en el cuerpo del *layout*.

Cada *layout* es específico de un controlador, por lo que un controlador busca en el directorio `app/view/layouts` un *layout* con su mismo nombre. Pero si el controlador no lo encuentra, buscará un *layout* identificado con `application`, que es el *layout* que se utiliza por defecto en toda la aplicación.

El comportamiento descrito en el párrafo anterior se puede modificar. Para ello se puede especificar en la definición del controlador el nombre del *layout* que se quiere utilizar:

```
layout "standard"
```

Incluso es posible especificar un *layout* para cada una de las acciones del controlador. Para realizar esta especificación se utiliza el atributo `only`:

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

Como el *layout* define la apariencia de la respuesta del controlador, es posible definir la apariencia dependiendo de ciertas condiciones. En el ejemplo siguiente, el *layout* aplicado depende de si el equipo pertenece a la primera o a la segunda división:

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

Incluso cada acción puede elegir el *layout* indicándolo en la sentencia `render`. En el primer ejemplo no se llama a ningún *layout* y en el segundo a uno específico:

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

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

Una característica fundamental de los *layouts* es que estos tienen acceso, al igual que las plantillas, a las variables del controlador, y esto permite implementar una personalización en ciertas partes de la página, como las cabeceras, los menús, etc.

Existe una alternativa más estructurada basada en la incorporación de cierto contenido a la vista (que no se muestra pero que el *layout* sí que utiliza posteriormente). A continuación se muestra un ejemplo sencillo de esta técnica:

```
<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 continuación se plantea un *layout* que hace uso de la vista anterior:

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

La clave está en la definición del segmento de código en la vista con `content_for` y en la llamada a este código mediante la sentencia `yield: menu` en el *layout*.

Parciales

Con los parciales es posible la inserción de pequeños fragmentos de código en vistas, lo que permite funcionalidades como las comentadas al inicio de este apartado (cesta de la compra, menú compartido, etc.).

Los **parciales** son vistas que son llamadas desde otras vistas, que reciben como parámetro un objeto y que cuando finalizan devuelven el control a la vista que la ha llamado.

Los parciales se identifican fácilmente porque su nombre empieza con el signo de subrayado (_). Por ejemplo, en la clase `Equipo` se creó un parcial `_form.html.erb` que se encuentra junto con el resto de plantillas en `app/views/equipos` y cuyo código es:

```
<%= 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 código anterior se define un formulario para la gestión de los atributos de la clase `Equipo`, utilizando la sintaxis que ya hemos estudiado. Este parcial es llamado desde la vista `edit` y `new` a través de la siguiente sentencia:

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

En el caso de que la llamada se produzca desde otro controlador (distinto del que contiene el parcial), es necesario un mayor detalle en la sentencia `render`, ya que se tiene que especificar cuál es el objeto en el que está el parcial:

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

En ocasiones, ciertas páginas HTML están formadas por listas de elementos (lista de la compra, lista de equipos de primera división, etc.), donde un parcial es encargado de aplicar formato a cada elemento individual de la lista.

Para implementar una vista a cada uno de los elementos de una colección, se utiliza el parámetro `collection` junto con el parámetro `:partial`.

Ejemplo

Para mostrar una lista de artículos utilizando el parcial `_articulo.html.erb` (donde se define cómo se mostrará cada artículo) se utiliza la sintaxis siguiente:

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

Se puede aplicar una cierta separación entre los distintos artículos, esta se puede especificar con el parámetro `spacer_template` en la llamada:

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

Con la sentencia anterior se aplica en primer lugar el parcial `articulo` y a continuación se aplica el parcial `separador`. Esta combinación se realiza en cada uno de los artículos de forma iterativa.

En las llamadas a los parciales se asume que estos se encuentran en el mismo directorio donde están las vistas del controlador, pero en ocasiones no tiene por qué ser así. Es el caso de parciales que son utilizados por distintos controladores; en estos casos se puede especificar la ruta donde se encuentra el parcial teniendo en cuenta que la ruta base es `app/views`.

Por otra parte, los parciales también pueden utilizar *layouts* que definen su diseño. Para implementarlo, se definen utilizando el parámetro *layout* en la llamada.

Para finalizar el apartado, uno de los usos más interesantes de los parciales es que los controladores pueden utilizar parciales directamente con el objetivo de actualizar partes de código de las páginas (sin actualizar toda la página), y proporcionar de esta manera la técnica necesaria para implementar AJAX en Ruby on Rails.

2.5. Migraciones

En los subapartados anteriores se ha presentado la estructura de cada uno de los componentes de las aplicaciones Ruby on Rails. Además, se ha visto cómo se pueden modificar de forma que a partir de la estructura inicial creada con el andamio se va personalizando la aplicación.

Pero a medida que se progresa en el desarrollo de la aplicación, el esquema de la base de datos va a necesitar evolucionar y adaptarse a las particularidades que van surgiendo. La herramienta que Ruby on Rails proporciona para estas tareas es la migración.

Las **migraciones** son ficheros Ruby que se encuentran en el directorio `db/migrate`, cuyo nombre es especial, ya que está formado por la fecha y hora de creación del fichero. Esta identificación define la versión del fichero y la secuencia temporal en la que se aplican las migraciones. La identificación del intervalo temporal está seguida del nombre de la clase sobre la que se aplicarán las modificaciones.

Como es de esperar, el comando `rails generate` creó un fichero `migration` que se encargaba de crear la base de datos y la tabla que almacenaba los valores de la clase `Equipo`. En el mismo apartado se llamaba al comando `rake :migrate` que ejecutaba el fichero anterior.

Ved también

Véase el el comando `rails generate` en el subapartado 2.1.1 de este módulo didáctico.

El fichero de migración anterior tiene el siguiente nombre:

```
20111001184529_create_equipos.rb
```

y su contenido es:

```
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 subapartado siguiente presenta las características básicas que permiten la modificación del esquema de la base de datos de una forma simple y que además permite adaptarla a las necesidades que surgen durante el desarrollo de la aplicación.

2.5.1. Creación y ejecución simple

Todas las bases de datos de las aplicaciones Ruby on Rails disponen de una tabla en la que se especifica la versión del esquema aplicado. Este número de versión coincide con el valor que identifica la fecha y hora de los ficheros de migración.

El mecanismo anterior es la clave del flujo de ejecución de una migración, ya que a lo largo de la creación de una aplicación se pueden crear distintos ficheros de migración: donde se crean tablas, se modifican, se crean columnas, se modifican, se eliminan, etc. Cada uno de estos ficheros de migración tiene un identificador único y se ejecutan de forma secuencial siguiendo el orden temporal definido en la fecha que forma parte del nombre del fichero.

Con la lógica anterior, al ejecutar el comando `rake db :migrate`, se consulta la versión del esquema que tiene la base de datos y a continuación se ejecutan todos los ficheros de migración cuyo identificador es posterior a la versión en curso. Es importante destacar que se realiza de forma secuencial y ordenada por la fecha-hora indicada en el fichero de migración.

Cuando la migración finaliza, el nuevo número que identifica la nueva versión de la base de datos se almacena en la tabla *schema_migrations*.

Pero si no se quieren ejecutar todas las migraciones, o si se quiere volver a una versión anterior del esquema, se puede especificar la versión de la base de datos que se va a conseguir:

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

La vuelta a una versión anterior se realiza a partir de la ejecución del código de la acción `down` definida en cada uno de los ficheros de migración.

Ejemplo

En el siguiente ejemplo se define una migración que añade el campo `telefono` en la tabla `Equipos`:

```
class TelefonoToEquipo < ActiveRecord::Migration
   def self.up
      add_column :equipos, :telefono, :string
   end

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

Ruby on Rails es independiente del sistema gestor de base de datos, por lo que dispone de unos tipos de datos propios y realiza la traducción a los tipos específicos de cada base de datos de forma automática. Los tipos existentes son: `binary`, `boolean`, `date`, `datetime`, `decimal`, `float`, `integer`, `string`, `text`, `time` y `timestamp`.

Cada uno de los tipos de datos anteriores dispone de las tres opciones siguientes:

- *`null`* `=> true/false`, de forma que con el valor asignado *`true`* la columna creada no puede almacenar valores nulos.

- *`limit => size`*, que define el tamaño del campo y que es utilizado mayoritariamente para definir el tamaño de las cadenas de texto.

- *`default => valor`*, que especifica el valor por defecto de la columna en caso de que no se especifique ninguno.

Los tipos numéricos disponen de dos opciones adicionales: `precision` y `scale`, que definen en primer lugar el número de dígitos que se almacenarán, y en segundo lugar, dónde se situará la coma decimal en estos dígitos.

Las acciones disponibles son las siguientes: crear, renombrar y eliminar tanto columnas como tablas, y además también es posible crear y eliminar índices.

Se utiliza el método `rename_column` para implementar los cambios de nombre en las columnas.

Ejemplo

El siguiente código define una migración que cambia el nombre de la columna `pueblo` por `población`. Como se observa, hay dos métodos: `up` que aplica el cambio y `down` que deshace el cambio:

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

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

El método `change_column` se utiliza para cambiar el tipo o ciertos atributos de una columna. Este método se utiliza de forma similar al método `add_column`.

Ejemplo

En el siguiente ejemplo, la columna `nombre`, que es del tipo `integer`, es cambiada al tipo `string`:

```
def self.up
   change_column :equipos, :nombre, :string
end

def self.down
   change_column :equipos, :nombre, :integer
end
```

Los cambios de tipos de datos son complejos. Esto es así ya que en ocasiones no todos los datos pueden ser almacenados en otro tipo distinto (por ejemplo, una cadena como “Hola, Mundo” no puede almacenarse como un tipo `integer`).

Los métodos `create_table` y `drop_table` son utilizados para la creación y eliminación de tablas.

Ejemplo

En el siguiente ejemplo se utilizan estos dos métodos:

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

      t.timestamps
      end
   end

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

En este ejemplo se crea una nueva tabla `equipo_copas`, con dos campos `equipo_id` y `descripcion`, pero no se ha definido la clave principal. Realmente no es necesario, ya que Ruby on Rails define por defecto el campo `id` como autonumérico y además el método `timestamps` crea los campos `created_at` y `updated_at`.

El método `create_table` dispone de un conjunto de parámetros opcionales:

- `force: true`, que en caso de existir una tabla con el mismo nombre, esta es eliminada.

- `temporary: true`, que crea la tabla en formato temporal, es decir, se eliminará cuando la aplicación finalice.

Por otra parte, el método `rename_table` permite cambiar el nombre de la tabla.

Ejemplo

A continuación se muestra un ejemplo de uso del método `rename_table`:

```
class RenameEquipoCopas <ActiveRecord::Migration
   def self.up
      rename_table :equipo_copas, :equipo_trofeos
   end

   def self.down
      rename_table :equipo_trofeos, :equipo_copas
   end
end
```

El cambio de nombre en las tablas puede provocar problemas, ya que puede romper la relación ORM, es decir, la relación entre las clases y las tablas que se han creado anteriormente.

Para finalizar el subapartado, los métodos `add_index` y `drop_index` proporcionan la funcionalidad que permite añadir índices en una tabla. Se crean índices cuando se dan búsquedas de registros por un cierto campo cuyo rendimiento es adecuado.

Ejemplo

En el siguiente ejemplo se añade un índice en el campo `nombre` de la tabla `equipos`:

```
class AddCustomerNombreIndexToEquipos <ActiveRecord::Migration
   def self.up
      add_index :equipos, :nombre
   end

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

3. Creación de una aplicación: Restaurante UOC

En este apartado se va a desarrollar una aplicación web donde se podrán registrar las reservas de un restaurante. La aplicación será accesible para clientes y empleados del restaurante donde se podrán dar de alta, de baja y se podrán modificar reservas.

Los empleados del restaurante dispondrán de vistas para visualizar las reservas previstas para los próximos días y así planificar el comedor con anterioridad.

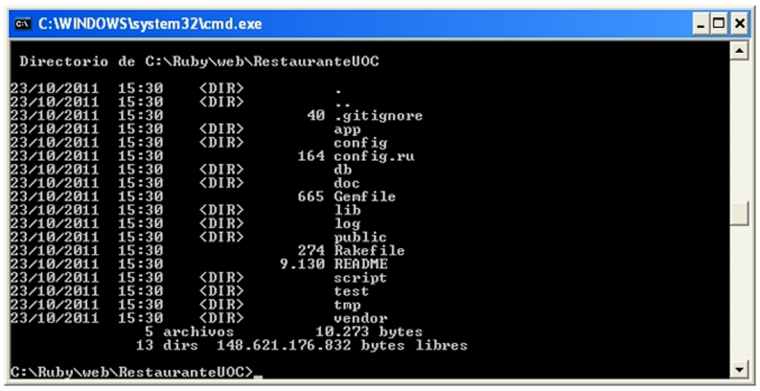
3.1. Creación de la aplicación `RestauranteUOC`

En primer lugar se creará un proyecto llamado RestauranteUOC. Para ello, en la carpeta `C:\Ruby\web` de la consola de Windows, se ejecuta el siguiente comando:

```
rails new RestauranteUOC
```

Para que las siguientes sentencias se ejecuten, es necesario situar la línea de comandos en el directorio `RestauranteUOC` (figura 23).

Figura 23. Estructura de los directorios del proyecto

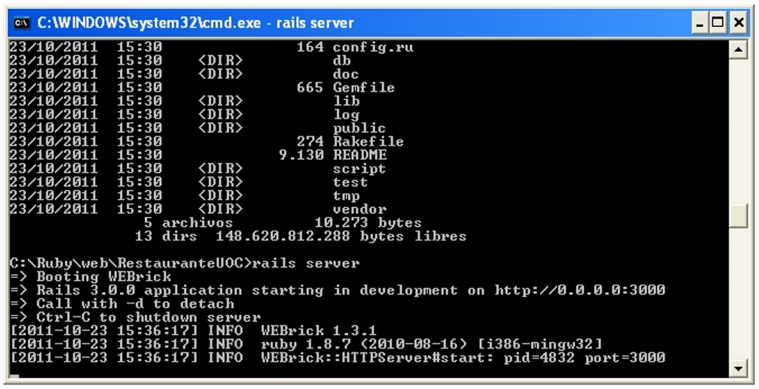


Arrancar el servidor web

Al crear el proyecto, se crea de forma automática un servidor que hospedará la aplicación.

Para probar la aplicación recién creada, será necesario iniciar un servidor que hospede la aplicación y permita ejecutarla. Para ello se ejecutará `rails server` desde la línea de comandos en el interior del directorio recién creado `RestauranteUOC` (figura 24).

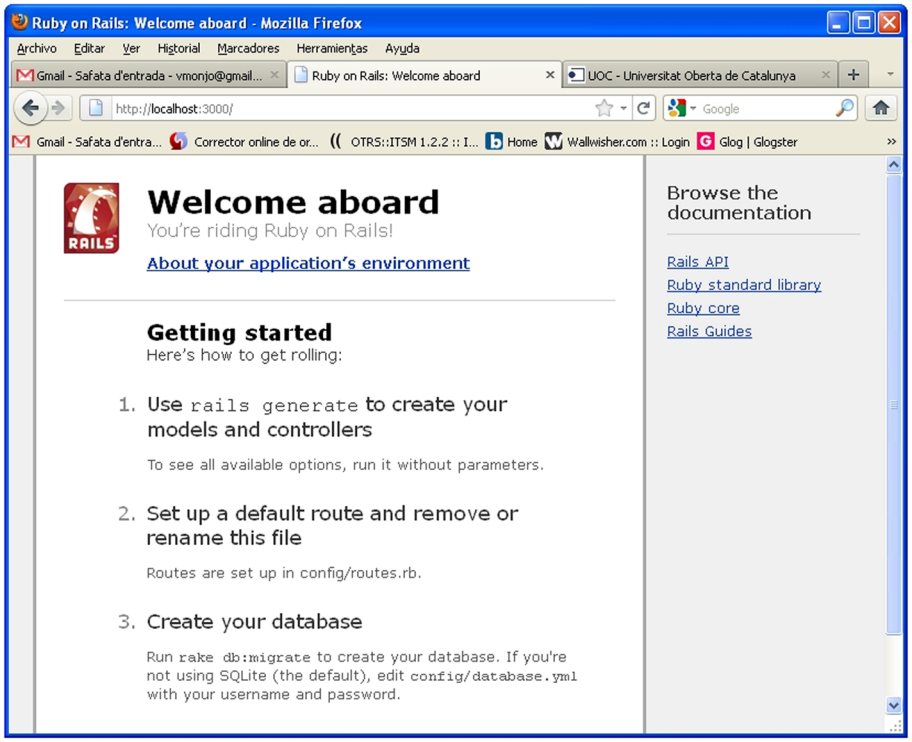
Figura 24. Arranque del servidor web WEBrick



Al ejecutar la aplicación en el sistema local, esta estará accesible indicando la URL`http://localhost:3000` en cualquier navegador web.

La figura 25 muestra la pantalla de bienvenida del nuevo proyecto.

Figura 25. Pantalla inicial del proyecto recién creado



El siguiente paso es la creación de la base de datos, para ello se utiliza el siguiente comando en el directorio `RestauranteUOC`:

```
rake db:create
```

Con el anterior comando ya se ha creado la estructura necesaria para construir la aplicación, pero todavía no tiene ningún contenido específico; este se irá ampliando en los subapartados siguientes.

3.2. Fase 1. Introducción de reservas

El requisito principal de la aplicación es la gestión de reservas, que debe contar con altas, bajas y modificaciones. Una reserva dispondrá de los siguientes campos:

Campo	Tipo	Descripción
Nombre	*String*	Nombre del cliente que realiza la reserva.
Apellidos	*String*	Apellidos.
Teléfono	*String*	Teléfono de contacto.
Fecha	Fecha-Hora	Fecha y hora que desea disponer de la mesa disponible.
Comensales	Numérico	Número de comensales.
Comentarios	*String* largo	Comentarios adicionales.

Para disponer de gestión de reservas en la aplicación, deberán crearse los siguientes elementos:

a) El **modelo de la reserva**: una clase que represente una reserva y permita también realizar diferentes operaciones, como la búsqueda de una reserva a partir de unos criterios, la actualización y eliminación de reservas, etc.

b) Las **vistas**: la aplicación debe contar con las vistas necesarias para visualizar la lista de reservas, el formulario de alta de una reserva, etc.

c) Los **controladores**: se encargarán de gestionar las peticiones realizadas.

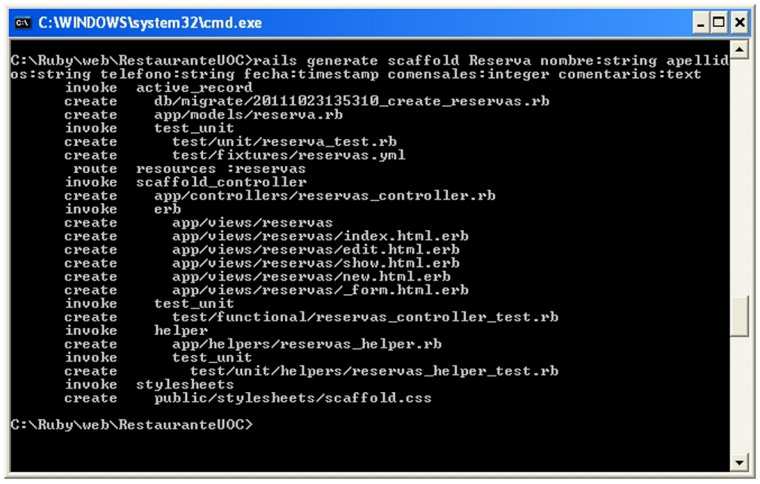
3.2.1. Creación del MVC

Se ejecutará el siguiente código para la creación mediante un andamio del modelo `Reserva`:

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

Se obtendrá el resultado que se observa en la figura 26. En estos momentos se dispone de la infraestructura necesaria para dar de alta, baja, modificar y eliminar reservas.

Figura 26. Creación del modelo mediante el andamio



a) El modelo `Reserva`

El andamio ha creado el modelo `Reserva` declarado en el fichero `app/models/reserva.rb`. El contenido del fichero es el siguiente:

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

Estas dos líneas definen la clase `Reserva`, que hereda de `ActiveRecord`. Esta herencia provee la funcionalidad necesaria para el acceso y la manipulación de reservas en la base de datos.

ActiveRecord

`ActiveRecord` proporciona el mapeo entre los objetos de la clase `Reserva` y la tabla `reservas` de la base de datos.

Por convención, se tomarán los atributos de la clase a partir de las columnas de la tabla `reservas`.

Para crear y destruir esta tabla, también se ha creado la clase `CreateReservas` en el fichero `db/migrate/create_reservas.rb`:

CreateReservas

`CreateReservas` contiene el nombre del modelo indicado en el generador `scaffold` y se le añade el plural de forma automática por convenio.

```
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` hereda de `ActiveRecord::Migration` y define los métodos tal como se ha estudiado:

Ved también

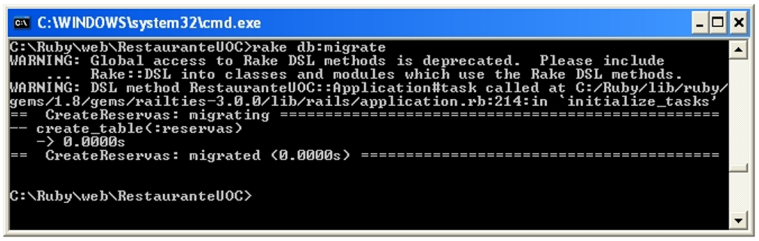
Podéis ver la definición de métodos en el apartado 2 de este módulo didáctico.

- `up`: que se ejecuta para crear la tabla que representa una reserva en la base de datos configurada en el proyecto. Las columnas son las mismas que se especificaron en los parámetros del andamio.
- `down`: elimina la tabla reservas de la base de datos.

El siguiente paso es la creación de la tabla `reservas`, esta se realiza utilizando el siguiente comando:

```
rake db:migrate
```

Figura 27. Creación de la tabla con la herramienta `rake`



Después de la ejecución del comando anterior se puede comprobar que se ha creado la tabla `reservas` con las siguientes columnas:

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

Para ello se utilizará la consola de `rails`:

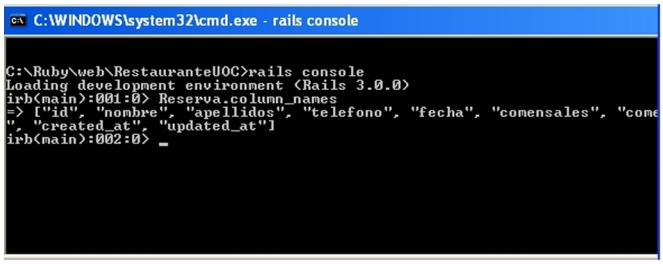
Ved también

Podéis ver la consola de `rails` en el subapartado 2.1.1 de este módulo didáctico.

```
rails console
```

Y dentro de la consola se consultan las columnas de la tabla `Reserva` con `Reserva.column_names` (figura 28).

Figura 28. Creación de la tabla con la herramienta `rake`



Se puede observar que se han añadido automáticamente columnas adicionales a las indicadas en la generación: `id`, `created_at` y `updated_at`. En particular, `id` es un identificador único, autoincremental y clave principal de la tabla. Por su parte, `created_at` y `updated_at` contienen la fecha y hora de creación y última actualización, respectivamente.

Los atributos de una reserva podrían estar declarados en ambas clases: en el modelo `Reserva` y en la clase `ReservaMigration`; pero ello entraría en contradicción con el principio "Don't repeat yourself". Por este motivo, inicialmente solo están en la clase `ReservaMigration`.

Ejecución de la aplicación

A continuación se ejecutará la aplicación para comprobar los resultados de la generación del nuevo modelo, controlador y vistas. Se debe acceder a la siguiente URL:

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

La figura 29 muestra una lista de reservas que inicialmente estará vacía, pero el enlace `New reserva` realiza una llamada a la acción `new` del controlador `reservas`, de forma que este llama a la vista `new` que en es enviada al usuario.

Esta vista tiene un parcial en su interior, que es el formulario (figura 30), de forma que se carga una página que contiene un formulario con los campos `Nombre`, `Apellidos`, `Telefono`, `Fecha`, `Comensales` y `Comentarios`. Al rellenar los campos y hacer clic en el botón `Create`, se llama a la acción `create` del controlador que guarda los valores y redirige el resultado mediante la vista `show`.

Figura 29. Página de listado de reservas

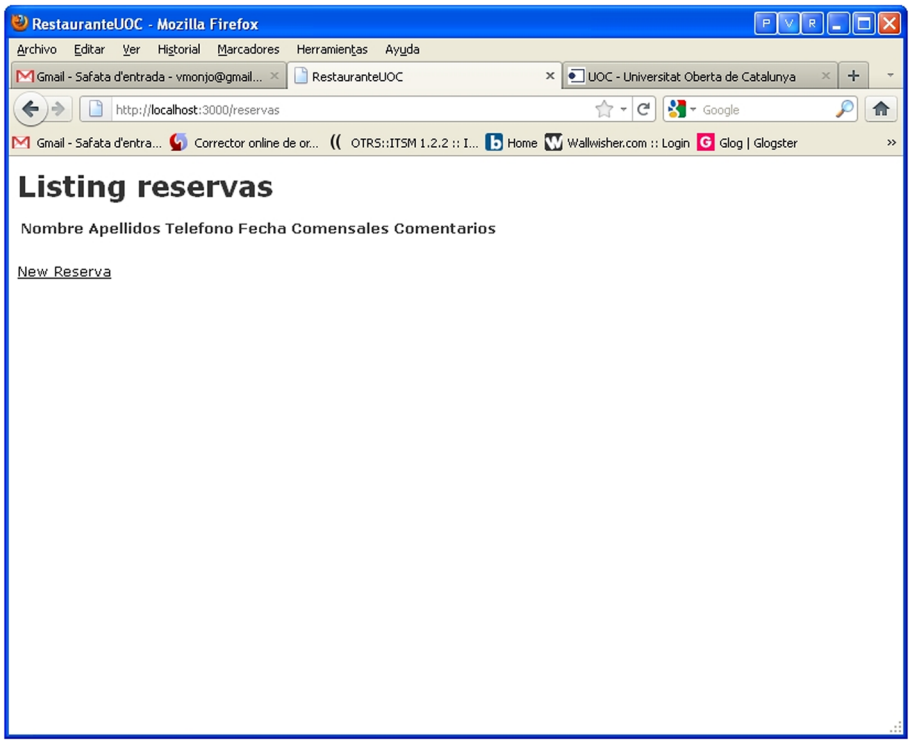
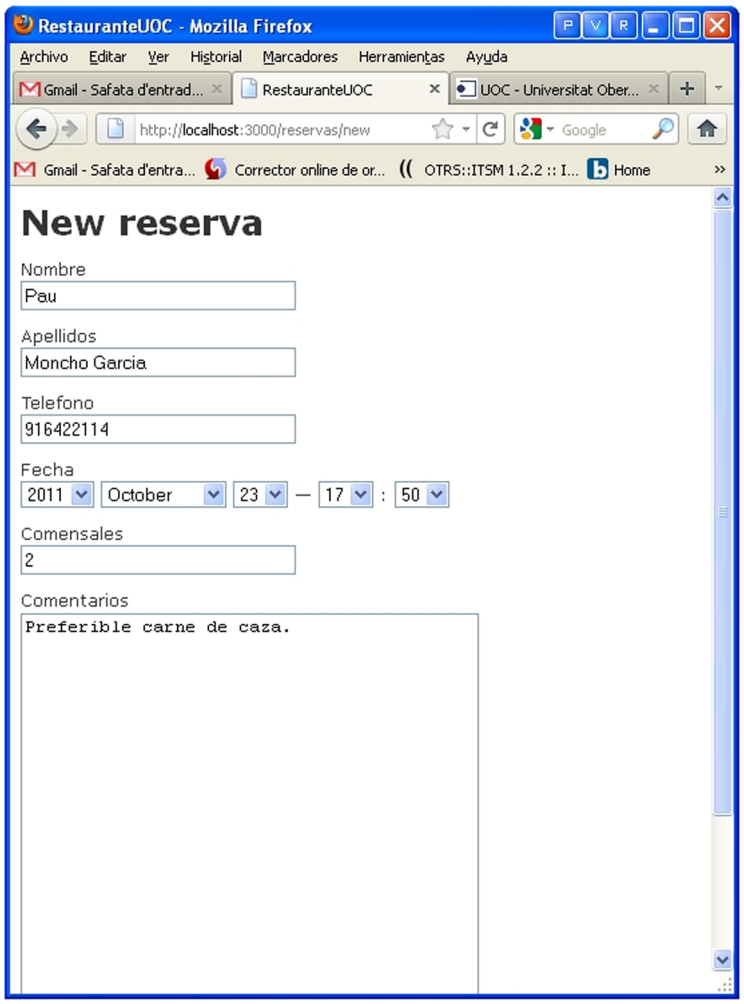
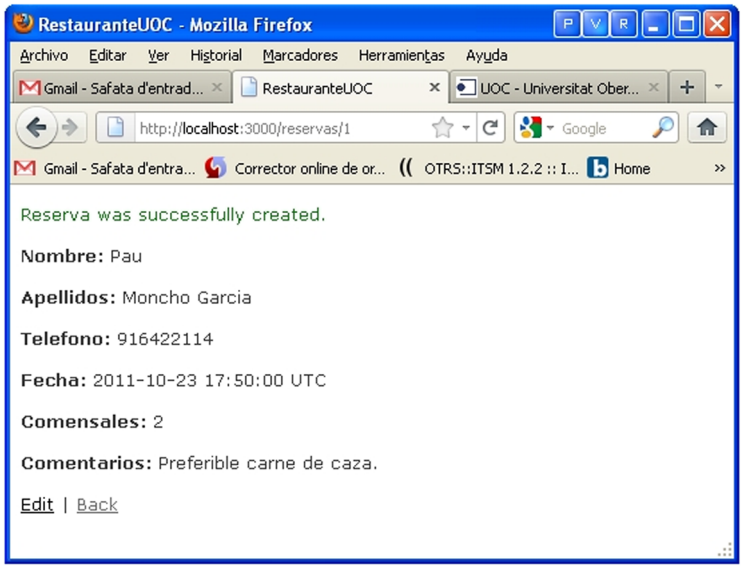


Figura 30. Formulario de creación de una reserva



Al crear la reserva, se muestra una ficha (figura 31) con la reserva en la URL `http://localhost:puerto/reservas/1`. El último número de la URL indica el `id` de la reserva asignado al crear el registro. En el pie de la página aparecen dos links: el primero `Edit`, que vuelve al formulario anterior para modificar la reserva, mientras que el segundo, `Back`, vuelve a la lista de reservas inicial.

Figura 31. Vista de una reserva



Si se vuelve al listado inicial, puede verse el nuevo registro de reserva (figura 32). En la misma fila aparecen tres enlaces: `Show`, `Edit` y `Destroy`, que sirven para mostrar la reserva, editarla y eliminarla, respectivamente.

Figura 32. Lista inicial con la reserva creada recientemente

3.2.2. Adaptación de la página inicial

Si se desea acceder al listado de reservas con la URL inicial de la aplicación, como por ejemplo: `http://localhost:3000`, se deberá modificar el archivo `routes.rb` ubicado en el directorio `config` añadiendo la línea siguiente:

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

Esta línea redirecciona cualquier petición dirigida a la raíz hacia la acción `index` del controlador `Reservas`. También debe eliminarse la página inicial anterior situada en el directorio `public` con el nombre `index.html`. Para que los cambios sean efectivos se tiene que reiniciar el servidor web.

Además de seleccionar la lista de reservas, como página inicial se le va a aplicar un cierto formato utilizando una hoja de estilo CSS. La asignación de la hoja de estilo se realiza en el *layout* `application`, ya que de esta forma se va a aplicar a todas las vistas de la aplicación. Si se consulta el código de este, se obtiene:

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

      <%= yield %>

   </body>
</html>
```

La línea `<%= stylesheet_link_tag :all %>` indica que se van a cargar todas las hojas de estilo que se encuentren en el directorio `public/stylesheets`. En este directorio solo existe el fichero `scaffold.css`, que es la hoja de estilo que crea el andamio por defecto.

Como la instrucción indica que va a cargar todas las hojas de estilo, la aplicación de un nuevo estilo es muy simple, ya que simplemente almacenando la nueva hoja de estilo en el anterior directorio, esta se aplicará.

Además se modificará el fichero `index.html.erb` para asignarle el estilo, quedando de la siguiente manera:

```
<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 %>
```

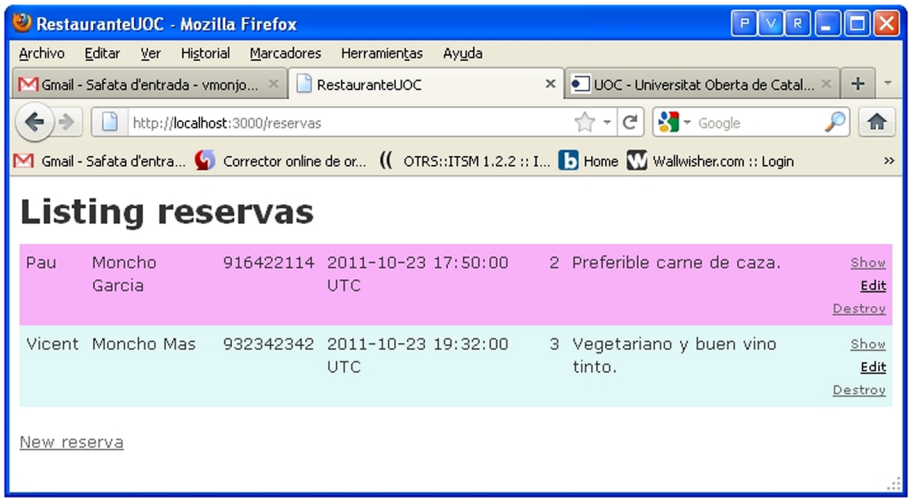
La sentencia `cycle` es la responsable de aplicar los estilos de forma alternativa. Para acabar, en el directorio `public/stylesheets` se insertará un fichero CSS con el siguiente código:

```
#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 resultado de la aplicación del estilo se puede apreciar en la figura 33:

Figura 33. Lista con el formato aplicado



3.2.3. Creación del formulario con validaciones

El lugar indicado en el patrón MVC para validar las entradas de valores es en la definición del modelo, de modo que cuando son guardados en la base de datos ya llegan con valores validados.

Enlace de interés

Se puede consultar el detalle de las validaciones disponibles en "Active Record Validations and Callbacks", RailGuides [en línea].

En primer lugar se consulta cuál es el código del modelo `Reserva`:

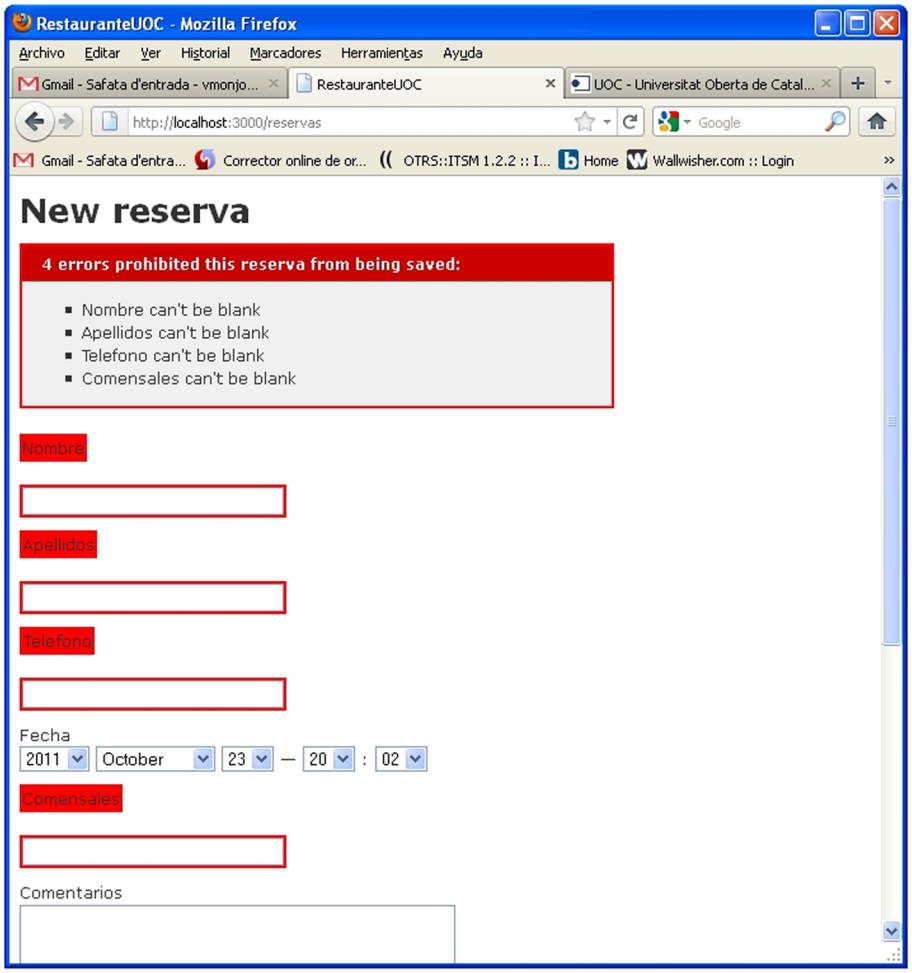
```
class Reserva < ActiveRecord::Base
end
```

El método `validates` es utilizado para realizar validaciones del texto introducido. A continuación se probarán ciertas variantes de este método. El siguiente código se debe introducir en el interior de la definición de la clase `Reserva`:

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

El código anterior comprueba que los campos no están vacíos y en caso de que alguno de estos esté vacío, muestra el siguiente error (figura 34):

Figura 34. Página con los avisos de las validaciones

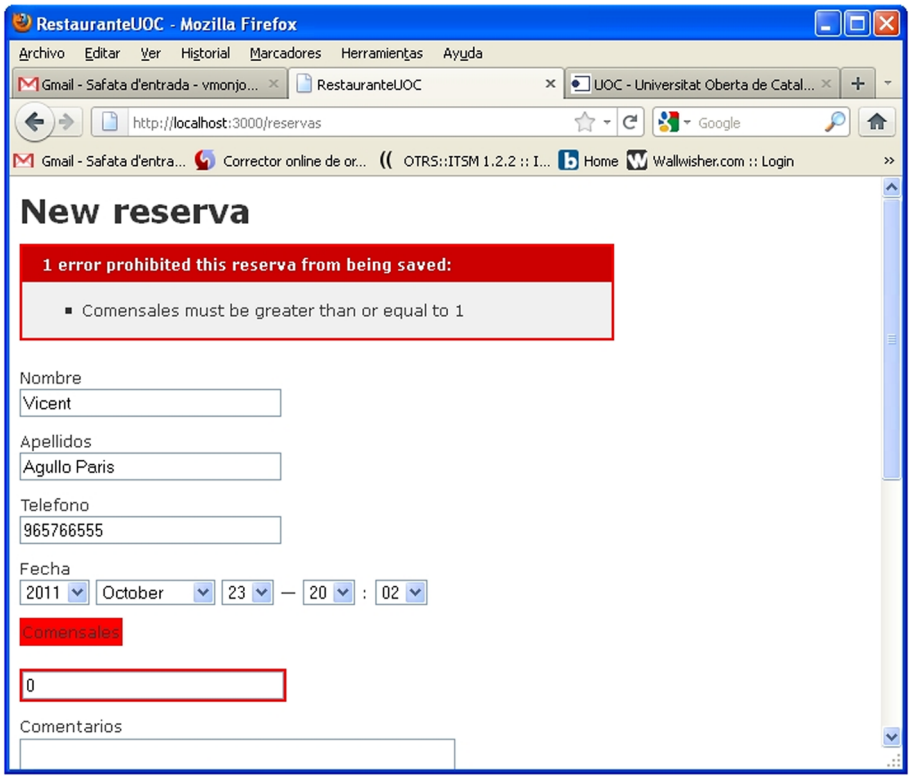


A continuación, el siguiente código valida que el número de comensales es mayor o igual que 1:

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

Con la línea anterior, si no se introduce un número mayor o igual que 1 aparece el siguiente código de error (figura 35):

Figura 35. Página con la validación del número de comensales



Con las dos validaciones anteriores, el código del modelo `reserva` queda de la siguiente manera:

```
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. Gestión de reservas

3.3.1. Creación del MVC

El siguiente paso es crear un controlador y una vista que permita gestionar las reservas de los próximos días. El objetivo se basa en poder consultar las reservas de los próximos días a través de la siguiente URL:

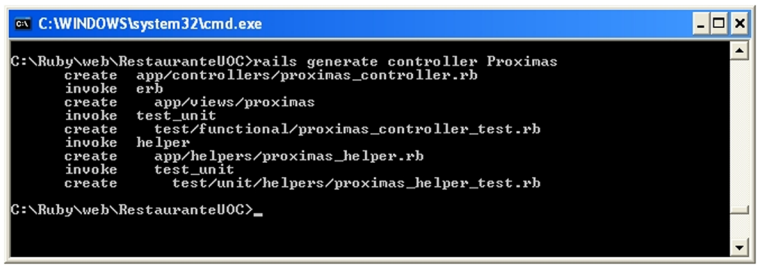
```
http://localhost:3000/proximas?dias=5
```

El parámetro `dias` será opcional y permitirá especificar los días de antelación en los que se desea visualizar las reservas. Si no se especifica este parámetro, se visualizarán las reservas de las próximas 24 horas por defecto.

Para crear el controlador se ejecuta la siguiente sentencia (figura 36):

```
rails generate controller Proximas
```

Figura 36. Creación del controlador `Proximas`



El generador ha creado un fichero llamado `proximas_controller.rb` y en su interior la clase `ProximasController`, que todavía no dispone de ninguna acción.

```
class ProximasController < ApplicationController
end
```

Se van a definir dos acciones `index` y `obtenerReservas` en el controlador anterior, que tendrán el siguiente código:

```
  def obtenerReservas
    @dias = params[:dias] || 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
```

La acción `obtenerReservas` realiza las siguientes tareas:

- Obtiene de la URL el parámetro `dias` en la variable `@dias`. En el caso de que no exista el parámetro, se establecerá un día por defecto.

- Crea una segunda variable llamada `inicio`, con la fecha y hora corrientes.

- Crea una tercera variable llamada `fin`, donde se establece la fecha resultante de sumar a la fecha corriente los días obtenidos en el parámetro.

- Finalmente, se declara una cuarta variable llamada `@reservas`, que contendrá las reservas cuyas fechas estén comprendidas entre la fecha de inicio y la fecha de fin.

El conjunto de reservas se obtiene a partir de la clase `Reserva`, se realiza el filtrado por fechas mediante el método `where` y el resultado es un conjunto de reservas con fecha en el período indicado.

La acción `index` llama a la acción `obtenerReservas`. `Index` es el método por defecto del controlador, por lo que no es necesario indicar la acción en la URL, simplemente `http://localhost:3000/proximas`.

En estos momentos se va a crear la vista asociada con la acción `index` que permitirá visualizar la lista de reservas contenidas en la variable `@reservas`. Para ello, se utilizarán parciales, ya que de esta forma se podrá reutilizar el código en distintas vistas.

Así, se va a crear `parcial` para mostrar la lista de reservas, por lo que se creará un fichero `_mostrarReservas.html.erb`, que se guardará en el directorio `/app/views/proximas`.

El contenido del fichero será el siguiente:

```
<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.nombre %></td>
         </tr>
         <tr>
            <td><b>Apellidos</b></td>
```

```
            <td><%= reserva.apellidos %></td>
         </tr>
         <tr>
            <td><b>Telefono</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 %>
```

La etiqueta `<style>` define un estilo para los elementos `<div>` de la página. Posteriormente se recorren todas las reservas mediante código Ruby embebido. Cada una de las reservas está contenida en un `<div>` en el que se aplica el estilo definido. Dentro de la capa se define una tabla que muestra toda la información de una reserva accediendo a los diferentes miembros de estas.

Pero todavía no se dispone de la vista, ya que el código anterior es el parcial que creará el contenido de esta. La vista asociada a la acción `index` del controlador `Proximas` se creará en el directorio `/app/views/proximas`, con el nombre `index.html.erb` y el siguiente contenido:

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

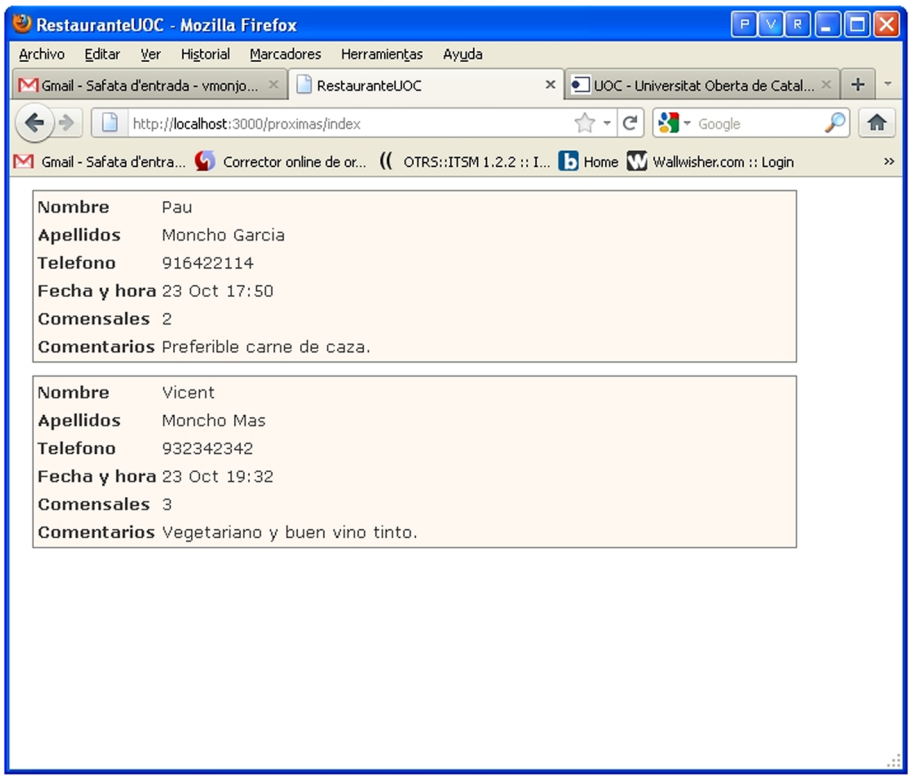
Cuando se ejecute el método `index` del controlador `Proximas`, se renderiza el fichero `index.html.erb`, que a su vez llama al parcial `_mostrarReservas` y muestra todas las reservas contenidas en la variable `@reservas` proporcionada por el controlador.

Para enrutar el controlador `Proximas` con su acción `index` se debe añadir la siguiente línea en el fichero `routes.rb`:

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

Para probar la nueva acción, es necesario reiniciar el servidor y acceder a la URL `/proximas/index`. Es necesario crear una reserva para antes de las próximas 24 horas.

Figura 37. Lista de las próximas reservas



En caso de querer consultar las reservas de los dos próximos días se realizará la llamada de la siguiente forma:

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

A continuación se va a modificar el *layout* para mostrar una cabecera, un pie de página y se definirá una sección cuyo interior mostrará la vista en curso.

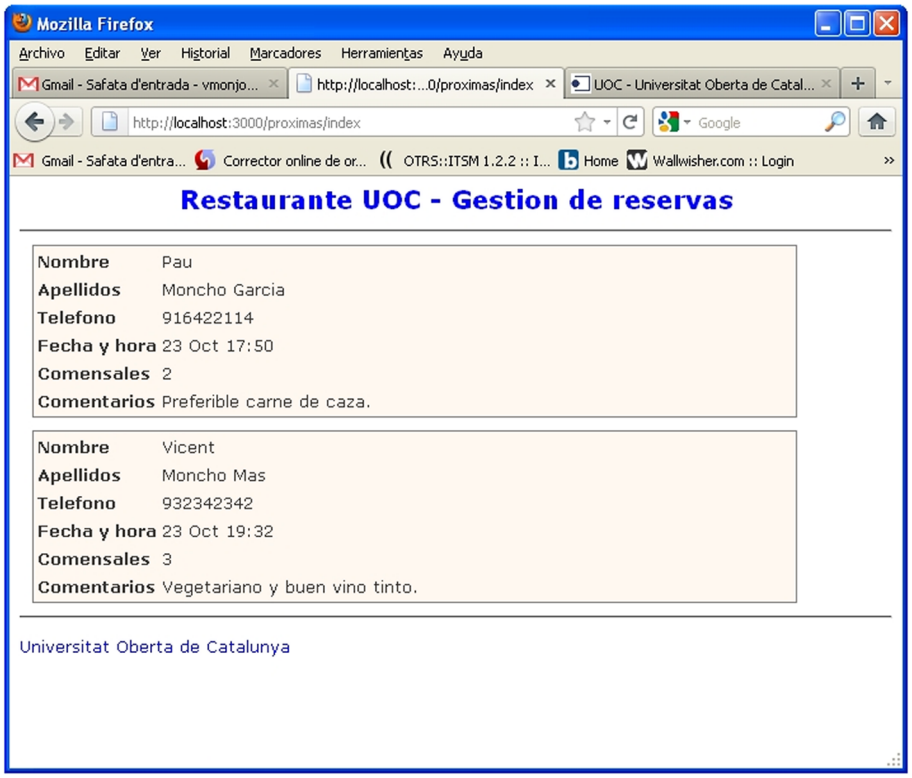
Para que la cabecera y el pie de página se apliquen a todas las vistas de la aplicación se modificará el *layout* `aplication.html.erb`, que se guardará en el directorio de *layouts* con el siguiente contenido:

```
<!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 - Gestion 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>
```

Con el nuevo *layout* se muestra el nombre del restaurante y un pie de página. En el cuerpo se encuentra la etiqueta `<%= yield %>`, que será sustituida por la vista de cada acción. El resultado de acceder a la página `/proximas/index` es la figura 38.

Figura 38. Acción index del controlador reservas con la plantilla modificada



¡Atención!

Puede ser necesario reiniciar el servidor para visualizar los nuevos cambios en la plantilla.

3.4. Aviso de reservas con AJAX

Mientras se gestionan reservas, sería interesante conocer si una reserva está próxima para que pueda prepararse el servicio. Para obtener esta información, se creará un *layout* para el controlador `Reservas` añadiendo un proceso que comprueba periódicamente mediante Ajax si existe alguna reserva próxima. En el caso de que exista alguna, se mostrará un enlace a la página que las visualiza.

Reflexión

Al añadir una función Ajax en el *layout* de `Reservas`, se obtendrán notificaciones sobre nuevas reservas en cualquier vista del controlador en la que estemos situados.

Prototype

Prototype es una biblioteca escrita en JavaScript para facilitar el desarrollo de páginas web dinámicas con Ajax. Ofrece funciones que abstraen las diferencias entre las implementaciones del DOM por los navegadores.

La plantilla `app/views/layouts/reservas.html.erb` contendrá el siguiente código:

```
<!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 - Gestion 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>
```

Las diferencias entre este *layout* y el que se definió en el subapartado anterior son las siguientes:

1) El comando siguiente importa la biblioteca JavaScript Prototype para el uso de Ajax:

```
<script type="text/javascript" src="/javas-
cripts/prototype.js"></script>
```

2) La sentencia siguiente define una función Ajax que realiza peticiones asíncronas de forma periódica:

```
new Ajax.PeriodicalUpdater
```

Dentro de la función se definen las siguientes características:

- `notificacionReserva`: Especifica la capa sobre la que se cargará el resultado de la llamada.

- `/proximas/existeReserva`: Indica el controlador y la acción que se va a llamar.

- `method`, `frequency` y `decay`: Indican el método http, la frecuencia en segundos de la llamada periódica y la velocidad a la que la frecuencia crece cuando la respuesta recibida es exactamente la misma que la anterior.

3) La línea siguiente define la capa que contendrá la respuesta de las peticiones periódicas:

```
<div id="notificacionReserva"></div>
```

La modificación anterior realiza peticiones asíncronas cada dos segundos a la acción `existeReserva` del controlador `Proximas`. El contenido de esta acción es la siguiente:

```
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
```

Con la acción anterior se obtiene el número de reservas entre la fecha en curso y los próximos días pasados en el parámetro. Este número se almacena en la variable `@number` y será utilizado por la vista `existeReserva.html.erb`.

Para que la acción `existeReserva` sea accesible tiene que incluirse en el fichero `routes.rb` la siguiente línea:

```
get "proximas/existeReserva"
```

La última instrucción indica que no debe renderizarse ningún *layout* para esta acción. La llamada realizada es asíncrona.

La vista asociada al método `existeReserva` mostrará la información en función del número obtenido, de forma que si existen reservas mostrará un enlace a `/proximas` y en caso de que no exista ninguna devolverá un texto informativo.

Se tiene que crear la vista `existeReserva.html.erb` dentro del directorio `app/views/proximas` con el siguiente contenido:

```
<% if @number == 1 %>
   <%= link_to ('Hay 1 reserva en las proximas 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 proximas 24 horas",
      {:action => 'index', :dias => '1'},:popup =>
      ['new_Window','height=600, width=650,scrollbars=yes']) %>
<% else %>
   No hay reservas en las próximas 24 horas.
<% end %>
```

El código de la vista evalúa el número de reservas obtenidas en el controlador a partir de la variable `@number`, de forma que dependiendo del valor realiza las siguientes acciones:

- Si el número de reservas próximas es 1, se devuelve un enlace que abre una ventana emergente hacia `/proximas/index`, que mostrará el listado de la próxima reserva.

- Si el número de reservas es mayor que 1, se devolverá un enlace parecido al anterior pero con el texto en plural, dado que hay más de una reserva.

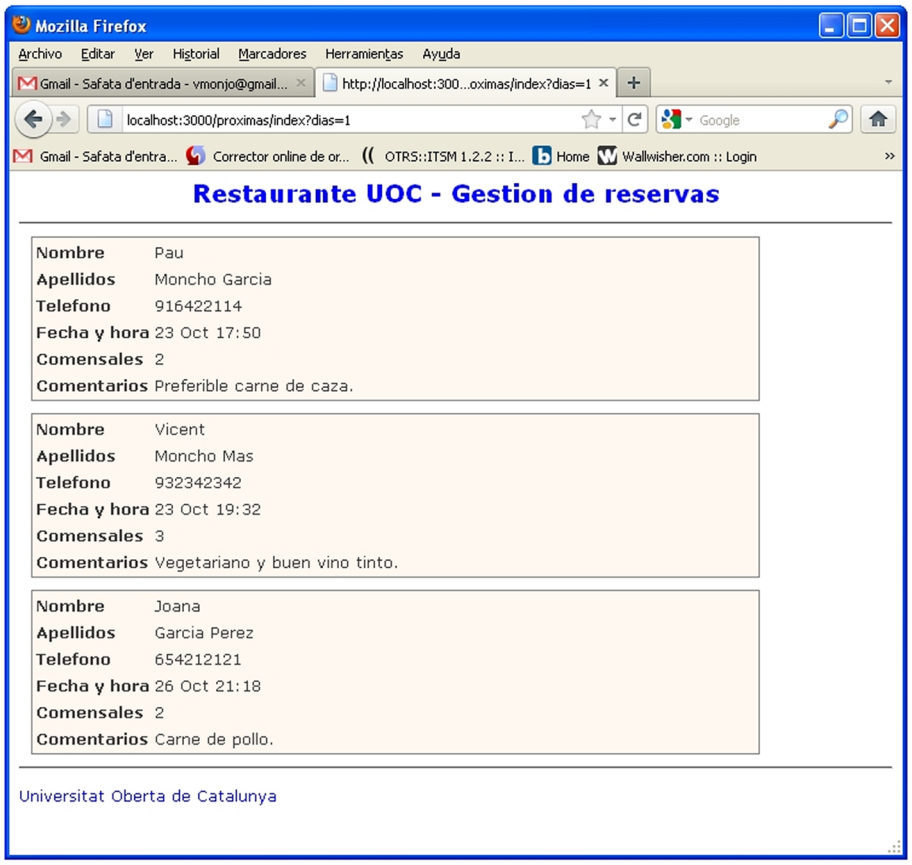
- Si no se cumple ninguna de las dos anteriores, se muestra un texto que indica que no hay reservas en las próximas 24 horas.

Para probar el anterior código se debe abrir la siguiente URL `http://localhost:3000/reservas/`. Si existe una reserva en las próximas 24 horas, la vista principal de gestión de reservas mostrará un enlace como el de la figura 39.

Figura 39. Actualización asíncrona de la página notificando que hay una reserva próxima

Al hacer clic en el enlace, se abrirá una ventana emergente mostrando la reserva (figura 40).

Figura 40. Lista de próximas reservas en una ventana emergente



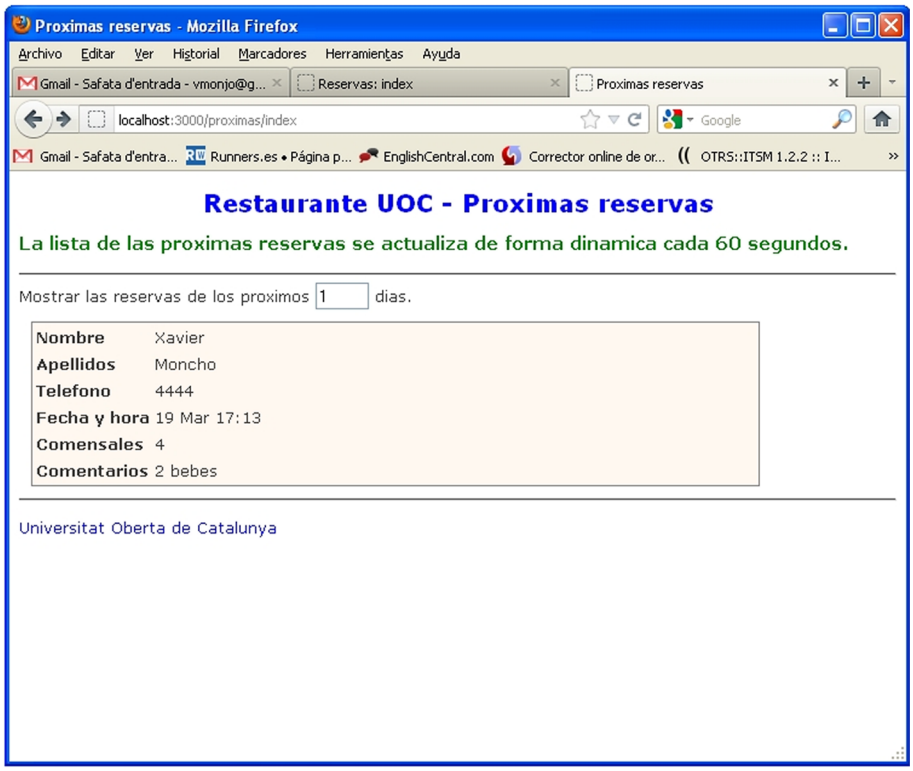
Layout para las próximas reservas

El siguiente paso es modificar el *layout* del controlador `Proximas` para que la ventana emergente como la de la figura 40 muestre los elementos siguientes:

- Información sobre el restaurante con una cabecera y pie de página, similar al *layout* del controlador `Reservas`.
- Información sobre la lista de próximas reservas de forma automática, periódica y asíncrona mediante Ajax.
- Una caja de texto para poder indicar el rango de días de próximas reservas.

La ventana emergente se mostrará de la siguiente forma (figura 41):

Figura 41. Ventana emergente con el *layout* modificado



Esta vista se actualizará a medida que pase el tiempo con las reservas que se acerquen a las próximas 24 horas o en los días introducidos en el campo de texto.

El primer paso se basa en la creación de una nueva acción `buscarNuevas` dentro del controlador `Proximas` que se encargará de obtener la lista de las próximas reservas:

```
def buscarNuevas
   obtenerReservas
   render :layout => false
end
```

De la misma forma que en la acción anterior, es necesario enrutarla para que sea accesible, por lo que se añadirá el siguiente código al fichero routes.rb:

```
get "proximas/buscarNuevas"
```

Y se reiniciará el servidor web para que los cambios en la ruta tengan efecto.

Al igual que el método `index`, `buscarNuevas` llama a `obtenerReservas` para devolver a la vista las variables `@dias` y `@reservas` que contienen el número de días de la petición y el conjunto de reservas obtenidos de la base de datos.

La instrucción `render :layout => false` evita que la vista devuelta por la llamada asíncrona muestre el *layout* definido para el controlador `Proximas`, ya que se trata de una llamada asíncrona y la vista obtenida se insertará en una capa. Dentro se creará una vista asociada a la acción `buscarNuevas` en el directorio `/app/views/proximas` llamada `buscarNuevas.html.erb`. El contenido del fichero será exactamente igual que `index.html.erb`:

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

Para finalizar se creará el *layout* `proximas.html.erb` en el directorio `/app/views/layouts`, que tendrá el siguiente código:

```
<!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 forma dinamica
         cada 60 segundos.
      </h3>
      <hr width="100%" align="center"/>
         Mostrar las reservas de los proximos

      <%= text_field_tag "numDias" ,"#{@dias}", :maxlength => 3, :size => 3 %> dias.

      <%= javascript_tag do %>
```

```
          new Ajax.PeriodicalUpdater('notificacionReserva', 'buscarNuevas',
       {
       parameters: 'dias=' + eval($('numDias').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>
```

Las etiquetas más destacables del código anterior son:

- `<%= text_field_tag"numDias",...%>`: muestra una caja de texto que contendrá el número de días obtenido por la variable `@dias`. A este control se le establece el identificador `numDias`. La lista de próximas reservas se actualizará tomando los días indicados en este control.

- `new Ajax.PeriodicalUpdater`: función Ajax que realiza consultas periódicas. Cada minuto actualiza la lista de próximas reservas contenida dentro de la capa `notificacionReserva`. La acción realizada es `buscarNuevas` del controlador `Proximas`.

Con estas modificaciones, se mantendrá la lista actualizada con las próximas reservas a partir del valor de la caja de texto. Si se quisieran obtener las reservas próximas a una semana, bastaría con cambiar el valor de la caja de texto a 7.