Web Development Using the Ruby on Rails

Articles

Posted by:

Posted on : 2008/7/27 23:54:31

Ruby on Rails is probably the most talked about and most controversial Web framework since the Internet was invented. After reading this gentle introduction, you will finally know what people all over the world are raving about!

Author: Nico Mommaerts

Introduction

Ever since companies discovered the merits of writing applications for the Web, people have been writing frameworks for making the work of a Web developer easier. The popularity of Web applications is due to the fact that a Web browser is all that is needed for a client to access a Web application. This makes deployment and maintenance a lot easier as a Web application only needs to deployed or updated on one server, instead of on thousands or possibly millions of computers. The downside of Web applications is that they are generally more complex to develop as a rich client application. Web applications are always written using a multitude of technologies at once: HTML, CSS and JavaScript for the presentation; Java or .NET for the application logic; a multitude of server environments and browsers each with their own quirks.

Given this complex environment to develop for, it is very easy to write unmaintainable code. A Web developer needs to understand and control multiple languages/environments, and it is not easy to automate the testing of such a heterogeneous system. To help overcome these issues with Web development, various frameworks have been written over the years, always promising to make Web development more productive and less error-prone. A framework is a collection of libraries and utilities that should facilitate the construction of an application. This doesn't have to be a Web application as frameworks exist for any kind of application. Just like not every car is suited for all terrain types, and not everybody has the same taste, there are hundreds of frameworks out there.

There is however one architectural pattern that most user-interface related frameworks implement, and that is the MVC pattern. MVC stands for Model-View-Controller and is basically the separation of these three concerns into three different layers.

* Model: the information the application works with, the model is usually persisted to a database but that is not necessary
* View: a representation of the model, multiple views are possible for one model, in fact that is one of the benefits of using an MVC pattern. In a Web application this is usually an HTML page but can also be a Flash page or something else
* Controller: the controller defines what needs to happen on different events, like the user clicking on a button, usually changes one or multiple models and chooses the correct view for the model By separating these concerns into different layers, changes to one of them don't have an impact on the other two layers, resulting in faster development cycles and easier maintenance.

What is Rails?

Ruby on Rails is a full stack, cross-platform open source framework. David Heinemeier Hansson originally wrote it, but in the meantime there is already a small group of core developers working on it. It often is abbreviated to 'RoR' or 'Rails' and the latter will be used throughout this article. 'Full stack' means that all the parts needed to build a Web application are present. Instead of having to learn different frameworks and tools and trying to get them to work together, the different Rails layers fit together seamlessly, which makes it a joy to work with! It runs on all major operating systems like Linux, OS X and Windows; it supports the most common open source databases like MySQL and PostgreSQL but also SQL Server, Oracle and others. It uses any Web server that supports CGI, the most popular being Apache and Lighttpd.

The entire framework is written in Ruby, a language which has been around for a while but whose popularity has exploded since Rails hit the scene. The language originated in Japan in '95 by the hands and brains of Yukihiro "Matz" Matsumoto, sort of as better, more object-oriented version of Perl. It harbors the same power as Perl does and yet is as fully object-oriented as SmallTalk. Its syntax also closely resembles these two languages. Ruby is an interpreted language, meaning it doesn't need to be compiled before execution. Everything in Rails is done in Ruby: from the configuration (the little there is) to the domain model and the Web pages, you only need to know one language. Given the dynamic nature of Ruby there is no development environment for it like Eclipse for Java or Visual Studio for .NET. You can develop in any text editor you like, although an editor with the notion of a project and syntax coloring is recommended. On Windows Eclipse with the Ruby plugin, FreeRIDE or Arachno Ruby (commercial) are popular, TextMate is the reigning editor on OS X, but no one keeps you from using vi, Emacs or even Notepad. There is even an open source editor specifically for Rails, called RadRails.

One of the selling points of Rails is that it is built with the DRY principle in mind. DRY stands for Don't Repeat Yourself, meaning that every piece of your system is described once and only once, which should make development and maintenance a lot easier since there is no need to keep multiple parts of the code in sync. Hand in hand with DRY goes 'Convention over Configuration', another one of Rails' core philosophies. Rails uses a set of code and naming conventions that when adhered to eliminates the need for configuring every single aspect of your application. Only the extraordinary stuff needs to be configured, like legacy database schemas or other resources you don't control. Using these two philosophies, DRY and 'Convention Over Configuration', Rails lets you write less code AND more features in the same time as with a typical Java or .NET application, with easier maintenance afterwards.

Rails is an MVC framework, which means your application is divided into three layers: the Model, the View and the Controller. Rails consists of several separate libraries. The two most important, which implement the MVC paradigm, are ActionPack, which takes care of the View and the Controller layers and ActiveRecord for the Model layer.

The Model

The model in a database-driven Web application is contained in the database. ActiveRecord lets you 'wrap' a class around each table, to which you can add your own business logic, and voila,

you have your own domain model. Let's see how this 'wrapping' happens when we have a table called 'employees' that we want to access. The ActiveRecord class would like this:class Employee end
Would you believe it if I said this class lets you access all the columns in the table 'employees' and enables you to read, write, update and delete records? Do you remember the philosophy of 'Convention over Configuration'? This is a very clear example of that principle: it lets you write less code (and fewer bugs). Let's take a look at how this works under the hood.

How does Rails know we are interested in the table 'employees'?

Convention: the name of the model is the singular of the database table name.

How can you access the different columns in the table?

Convention: Rails adds attributes to the Employee class with the same name as the columns in the table. For example, the 'employees' table has a column named 'name', so the Employee class would have a 'name' attribute. The attributes are actually added at runtime, using Ruby's powerful dynamic capabilities, this is why you don't see them in the source code.

All the code required to load/save the data from/to the table is inherited from the ActiveRecord::Base class, all for free! You can add your own business methods to this class if you want.

Storing and loading Employees won't get you very far though. It would be nice if we could define a relation with some other tables. After all, we are working with a 'relational' database. Let's suppose we have an ActiveRecord class 'Company' and a database table 'companies' (remember that the classname is the singular of the tablename). In order for us to be able to ask an Employee object what company he works for, we need to tell Rails that an employee belongs to a company:class Employee belongs_to :company
end
If 'employee' is an instance of the Employee class, we can ask it what company is attached to it:employee = Employee.find_by_name('David Heinemeier Hansson')
note that Rails has find_by_ method for each column!
puts employee.company # -> 37 Signals
Note: everything after # is considered commentary

Again, this magic works by using certain code conventions. By saying 'belongs_to :company' Rails expects to find an ActiveRecord model named Company, which in turn expects a table called Companies. The relationship between the two is figured out by the 'belongs_to' statement, which expects to find a foreign key column in the Employee table, with the name of the table it points to. The name of the foreign key in this example would be 'company_id'.

This is only one example of a possible relationship, the others are :has_one, :has_many, and :has_and_belongs_to_many. They all work using the same principle, the convention of using primary and foreign keys with a certain name. All the metadata is extracted at runtime from the database

tables and there is no explicit mapping present. It is therefore not possible to generate automatically a database schema like for example Hibernate does, a well-known object-relational mapping framework in the Java world.

That doesn't mean there is not any help at all from ActiveRecord to manage your database(s). There are two classes that help you: ActiveRecord::Schema, a class that allows you to define your table and ActiveRecord::Migration, a class that helps you manage the evolution of your schema. Instead of writing SQL DDL statements to create your database, you can describe your tables or modifications in Ruby using a domain specific set of classes and methods, just like we defined relationships between various tables. Lets see how we could define our 'employees' table using a Schema class:ActiveRecord::Schema.define do
create_table :employees do |t|
t.column :name, :string, :null => false
t.column :birthdate, :date
t.column :email, :string
end

add_index :employees, :name, :unique
end
As you can see the domain specific language used here is so clear, that no real explanation is needed: a table called 'employees' is created, a few columns and an index are defined. Since this is just plain Ruby, the database creation is completely database independent and supports every database that Rails supports, except for DB2.class AddSSNToEmployees def self.up
add_column :employees, :ssn, :string
end

def self.down
remove_column :employees, :ssn
end

end
This class has two methods: 'self.up' and 'self.down'. The first one is used when upgrading to a new version of the database, the latter for downgrading if needed. Again, you don't need to understand Ruby at all to see what is going on here, a new table called 'employees' is created and some columns are added. A 'schemainfo' table is automatically created where the current version of the database is stored. The default Rails installation puts a script in your project directory that executes all migrations in the correct order and brings your database up to date. Using Rails Migrations alleviates the pains often found when pushing changes to multiple development databases and other environments like test or production that may be multiple versions behind. Needless to say, this is only a small portion of what you can do with ActiveRecord, but it did demonstrate the most important aspects. Conventions and a domain specific macro language let you describe your models and express relationships between them. When you can't adhere to these conventions due to external constraints (an existing database schema for example) or whatever the reason is, you can override this wherever you want to.

The Controller

The classes that make up the Controller layer are bundled together in the ActionController module. The Controller layer in a Web application is like a policeman regulating traffic, but instead of regulating car traffic, it controls incoming and outgoing Web/database traffic. In Rails, you typically have a whole bunch of these policemen (or ActionController classes). Each one has its own area to control and it is entirely up to the developer to specify these areas. You could have one that is in charge of the contracts and another one that is in charge of your employees:class ContractsController end

class EmployeesController end

As its name implies, a Controller class controls something, so it has to be able to 'do something' with the contracts or employees. The Controller classes in the previous example don't seem very capable of doing anything like storing a new contract, or firing an employee. They are powerless policemen. In Rails terms, they still need 'actions'. Actions define what a Controller is able to control, like a policeman can 'give a ticket', or 'stop a car'. Without actions there would not be much happening in your application. You specify actions for a Controller simply by defining methods in a Controller class:class EmployeesController def show
@employee = Employee.find(params[:id])
end

end

This is not much code, but a lot of stuff is done for you behind the scenes, a testimony of the ease with which you write Rails Web applications. Most other popular Web application frameworks would require a lot more code and configuration to achieve the same result. Let us lift the curtains and peek at what is happening in the EmployeeController. The play begins with a user clicking on a link ' http://www.myrailsapp.com/employees/show/1'.

Now what happens first is that Rails starts dissecting the url to see which action Controller needs to perform. We notice here another convention: the url needs to comply with a specific format. In this case the format is 'http://www.myrailsapp.com/:controller/:action/:id'. Rails will simply look at the strings separated by a forward slash and will confer meaning to them based upon their order. The first part is 'employees', which maps to the Controller name. The second part is 'show', which maps to the name of the action the Controller needs to execute. The third and final part is the id of a record in our database. This very simple url convention lets you write clean and logically constructed urls. As a developer it is really easy to figure out what is going to happen when looking at the url without having to trace through various mapping files as is the case in many other Web frameworks. Just by looking at the url Rails knows to go to the EmployeesController and calls the public method 'show', without writing any code yet! Once inside the method 'show' we see one line of code:@employee = Employee.find(params[:id])

Employee is the class we saw in the previous chapter about the Model layer. It class method 'find' takes various parameters, in this case params[:id]. The parameter 'params' is a map that contains all incoming parameters, be it from an url, a GET or a POST. Rails collects these on each incoming request, another annoyance the developer doesn't have to worry about anymore! The right part of the assignment can now be explained as: find the employee whose id is in the value of params[:id], which is 1 as specified in the example link.

On the left part we have '@employee'. The '@' in front of the variable means that we want to make it a class variable. In Ruby we don't have to declare the variable first, it exists as soon as we put something in it. When you put something in a class variable, it means you want to make the variable available to the view layer. We want to show the data of an employee on a Web page (the view) so the result of the Employee.find method is put in the class variable @employee.

It looks like we're done now because there is no code left, but there is still one thing left to do: show to the user the information we loaded from the database. If we don't state explicitly in the action what has to happen after executing the action, Rails assumes, correctly in this case, that we want to load a new Web page in the application. Rails needs an html page (well it's a bit more as plain html but we'll see that in the next part) to render the Web page. This is already been taken care of by yet another naming convention. All Web pages reside in a directory that Rails knows about, just like the directory that contains the models, the configuration, etc. In this directory, Rails looks for a subdirectory with the same name as the controller, 'employees' in this case, and then looks for a file with the same name as the action, 'show'. This is an overview of the file layout (I omitted the irrelevant folders):/app

```
  /controllers
 employees_controller.rb-> the controller + .rb extension (Ruby source)

/models
 employee.rb -> the model + .rb extension (Ruby source)

/views

  /employees -> the name of the controller
show.rhtml > the name of the action + .rhtml extension (Rails html file)
```

These directories are generated by Rails at the time you setup the project.

Another example to illustrate these principles:

```
class ContractsController
  def destroy

  contract = Contract.find_by_number(params[:number]
        contract.destroy()
# removes the row that contains the contract            redirect_to &#39;show&#39;
end

  show
@contracts = Contract.find(:all)
 end

  end
```

We have a ContractsController with an action 'destroy'. Obviously this action deletes a certain contract. Lets presume there is a form on a Web page that lets the user delete a contract by pressing a DELETE button. The action 'destroy' starts with looking up the contract to

delete using the Contract model: Contract.find_by_number(params[:number]).

The Contract model seems to have a method with the name of the column of interest, 'number' in it. How is it possible? Well, just like Rails adds attributes to the model class based upon the names of the columns of the table it belongs to, it also adds 'find_by_' methods for each column name. As with the automatically added attributes, it doesn't matter if you decide to rename a database column later on, since all this code is added by Rails at runtime. You never have to maintain this generated code! See how Rails gives you a powerful framework to work and yet doesn't require you to write much code?

Back to our example, after the contract is retrieved, it is removed from the database by calling the method 'destroy', nothing earth-shocking there. The next line is something we have not seen before: redirect_to 'show'. In the previous example, we assumed we wanted to render the loaded information, but in this case we didn't load any information, quite the opposite even! We want to show the user an overview of all remaining contracts after the user deletes one. We will add a separate action 'show' for that. This method can be used by visiting the url 'http://www.myrailsapp.com/contracts/show'. The 'redirect_to' statement lets Rails execute the 'show' action immediately after executing the 'destroy' action. The 'show' action doesn't state anything explicitly after loading all the contracts so it assumes it needs to render a Web page (located in /views/contracts/show just like in the employee example). We could explicitly say to render a Web page if we wanted to by saying: render 'show'.

The View

Our ride through Rails is almost done, only the View layer is left. It can be found in the ActionView bundle and is the easiest one of the three. The View is comprised of all the rhtml pages found in the 'views' directory, as shown in the Controller examples. There is not much difference between a plain html file and a rhtml file, except that a rhtml file may contain small pieces of embedded Ruby code, called ERb.

Rails defines some helper methods for use on rhtml pages that make it easy to generate links, forms etc in your pages. There is also support for AJAX, that nifty JavaScript technology that lets you make Web pages that behave almost like a rich client. You can create typical AJAX effects, like submitting a form without refreshing the entire page, without even having to write any JavaScript yourself.

There is one other important thing to know about rhtml pages in Rails: it is possible to use a template system, called Layout. It would be really cumbersome if we would have to write each rhtml page beginning with the opening tag until the closing tag, especially given the fact that most pages will probably have a lot of html in common, for example a navigation bar. The template system takes care of this:

The '' line is an example or ERb. The denotes that we want to execute a piece of Ruby code and means we want to render the output of the embedded Ruby code on the html page. The 'yield' method returns the content of the currently rendered page. This very simple

example layout dismisses the need to write the boilerplate html every page. A page may look like this:

```
 no need to write surrounding html tags
```

The final rendered page would have an html source like this:

```
   no need to write surrounding html tags
```

How does Rails know which template to apply, if any? Again, it looks in certain directories and expects certain filenames. Continuing on the previous file layout, it could look like this:/app

/controllers
 employees_controller.rb

/models
employee.rb

/views

/employees
show.rhtml

/layouts
application.rhtml
employees.rhtml
The default directory for storing layouts is in views/layouts. When rendering a view, for example the employees/show.rhtml view, Rails will look in the layouts folder and search for a file with the same name as the controller that renders the view, 'employees.rhtml' in this case. If the employees.rhtml file contains a line , it will be replaced with the contents of the employees/show.rhtml file.

You might also notice another (optional) file in the layouts directory, the 'application.rhtml' file. This is a special template, as it will be used as the default template for each controller unless explicitly overridden.

Summary

If you are already familiar with Web frameworks that use the MVC pattern, you might have noticed that Rails isn't anything revolutionary. Rails' strength is rather in the synergy between its various parts, ActiveRecord, ActionPack, ActionMailer and ActionWebService. I have not touched the last two modules but they are not as important as the former two. By fully utilizing the dynamic power of Ruby and using conventions instead of tedious configuration files, Rails became one the most beloved Web frameworks of the last years. Since it is so beloved by some, and loathed by others, discussions about it often become very emotional and its hard to know what to believe or not. Is it a framework that will solve all our problems and make all other Web frameworks obsolete? Or is it only suited to build small Web applications, nothing up to the challenges found in the 'Enterprise' world? The correct answer is none of the two. Rails is a solution for a certain

range of problems and it is the responsibility of the developer to choose the appropriate tools for the job.

Links

* http://rubyonrails.org/
* http://weblog.rubyonrails.org/
* http://www.loudthinking.com/
* http://www.ruby-lang.org/en/
* http://www.rubycentral.com/
* http://www.radrails.org
* http://freeride.rubyforge.org
* http://www.ruby-ide.com/ruby/ruby_ide_and_ruby_editor.php
* http://en.wikipedia.org/wiki/Ruby_programming_language
* http://en.wikipedia.org/wiki/Model-view-controller
* http://en.wikipedia.org/wiki/Framework
* http://en.wikipedia.org/wiki/Web_applications Originally published in the Fall 2006 issue of Methods & Tools