Ruby on Rails and XML

Generate a Rails stub to manipulate an XML document

Skill Level: Intermediate

Daniel Wintschel (daniel@humandoing.net)
Software developer
Helium Syndicate

17 Apr 2007

You've very likely heard of Ruby on Rails. Maybe you've actually used it; perhaps it is your new programming mistress. Whatever the case, it looks like Rails is here to stay, and to everyone's benefit. Ruby plays very nicely with XML -- read further for the details.

Section 1. Before you start

This tutorial is for a general programming crowd interested in learning the basics of setting up a skeleton Rails application and using Ruby and Rails to process XML. Beginner and intermediate programmers or people who have a little bit of exposure to Rails will likely benefit the most. It will spend a brief amount of time discussing Rails in general, and Ruby syntax as necessary, but these topics are covered in much better detail elsewhere. Please see Resources for additional information.

What is this tutorial about?

Today you will build a Rails application, discuss some basics about the way that Rails works, how it's structured and how to use it, and then you'll move on to working with XML. There are a number of ways to both generate and parse XML in Ruby, and you'll look at a few of them, including REXML (Ruby Electric XML), Builder and Hpricot (Technically Hpricot is an HTML parser -- but it's fast, and works on XML, too).

Prerequisites

The following tools are needed to follow along with this tutorial:

- Ruby -- If you run Windows, your best bet is to download the One-Click Ruby Installer. If you use some variation of Linux or Mac OS X, you might already have Ruby installed. If not, you can download it from http://www.ruby-lang.org. The installation instructions are straightforward. Version 1.8.4 or 1.8.5 is recommended.
- RubyGems -- Get the gems you need, and install Rails, Builder and Hpricot if you haven't already.
- Rails -- You can install Rails through RubyGems. While not really part of this discussion, you figure all that out at http://www.rubyonrails.com/down. You'll use version 1.2.2 for this tutorial.
- Builder -- Install through RubyGems.
- Hpricot -- Install through RubyGems.

One of the beautiful things about Rails is how easy it is to perform object persistence and relational mapping. Since you'll only deal with XML in this tutorial, you won't actually use a database for anything.

To actually run the demo application, it's worth noting that the only thing you should have to do is to start the application. You can see how to do that in Start up the server.

Also, you'll be provided with a list of all of the files that have been explicitly created or modified aside from those that will be generated by Rails. These include:

- app/controllers/main_controller.rb
- views/layouts/main.rhtml
- views/main/index.rhtml
- public/stylesheets/reset-fonts-grids.css
- public/stylesheets/style.css

Section 2. Introduction

Over the past five to seven years, XML has become the de-facto standard (or at least one of the de-facto standards) for inter-application communication. Whether it's an application running on the same machine, or some Web service out in the ether, a payment gateway, a shipping gateway or another random data source, XML is something that you've likely come to know, like it or not. With that said, it seems a little ironic to write a tutorial that deals with XML processing on a framework that became famous with words like: "Learn how to use the open-source Web framework Rails to create real-world applications with joy and less code than most frameworks spend doing XML sit-ups." (Cited from Loud Thinking). It's worth noting that none of the actual XML processing done in this tutorial is dependent on Rails. All of the XML APIs you'll use are straight up Ruby and can be used in any Ruby application. You will do it in a Rails context for the sake of providing a way to upload files to process, and an easy way to interact with the code that you write.

Ruby

Ruby, if you didn't already know, is a dynamically typed scripting language that was created by Yukihiro "Matz" Matsumoto. He began developing the language in 1993, and it was released to the public in 1995. With all sorts of great features like closures, callbacks and duck typing, it allows for some amazingly flexible frameworks to be built on top of it, which leads to Rails.

Rails

Rails is a framework built by David Heinemeier Hansson and released to the public in 2004. Rails is a framework for building Web applications using the model, view, controller (MVC) architecture and emphasizes programming by convention over configuration. It has all sorts of amazingly wonderful features like fully automated object-relational mapping and persistence. In this tutorial, you demonstrate XML parsing with some of the basic features of Rails, namely simple controller and view features.

Ruby and XML

Ruby has many APIs for parsing and generating XML. REXML is the standard API that ships as part of the Ruby core, whereas frameworks like Builder and Hpricot are available as gems. Gems are essentially third-party APIs or libraries that conform to a specific format. They are easily downloadable and installable (conceptually similar to a third-party Java API available in a JAR file).

REXML

REXML is the standard Ruby library for parsing and creating XML. It is bundled as part of the Ruby Core language distribution, so you know it will always be available.

Builder

Builder is a third-party gem that is becoming more and more popular for the purpose of generating XML content. The syntax is very clean, easy to use and read, utilizing Ruby's method_missing callback for generating XML.

Hpricot

Hpricot is a fast HTML parser (but you can parse XML with it, too!) written by "why the lucky stiff" that is gaining in popularity as well.

#{random_xml_util}

Ruby has several other APIs for parsing XML, but I will not cover them in this tutorial. If you want to know more, go to RubyForge and search around.

Section 3. Building the skeletal Ruby on Rails app (in preparation for XML processing)

Now, you'll move on to the good stuff. In this section, you'll take advantage of a bunch of the bundled scripts that generate all sorts of Rails goodness for you. I will discuss some Rails basics in this section before I get to the meat of XML processing.

Generate the Rails application stub

I did all of the work in this tutorial in Mac OS X from the terminal, so if you use Windows, feel free to follow along using the command line. All of the commands should work exactly the same.

The first thing you do is create the Rails application by executing the command in Listing 1.

Listing 1. Command to generate a Rails application stub

```
rails xml_tutorial -f
```

This runs the Rails script and tells it that you want to create a new Rails application called xml_tutorial. It also used the `-f` flag, which freezes the Rails version to the current version that you have installed. This will copy the current version of the Rails framework into the vendor/plugins directory, and will hopefully help you have less trouble when you try to run the application, in the event that you have a different version of Rails installed.

The script runs and generates the application structure for you. See the partial output in Listing 2.

Listing 2. Partial output from Rails application creation script

```
monkey:~/Work/Rails_XML_Tutorial daniel$ rails xml_tutorial -f
      create
      create  app/controllers
      create  app/helpers
      create  app/models
      create  app/views/layouts
...
...
rm -rf vendor/rails
mkdir -p vendor/rails
cd vendor/rails
mv activesupport-1.4.1 activesupport
mv activerecord-1.15.2 activerecord
mv actionpack-1.13.2 actionpack
mv actionmailer-1.3.2 actionmailer
mv actionwebservice-1.2.2 actionwebservice
cd -
froze
monkey:~/Work/Rails_XML_Tutorial daniel$
```

Now you can start to see the beginnings of Rails 'convention over configuration' by seeing the default application structure created for you. Within a Rails application, you place files in certain locations depending on what they are. Controllers, helpers, models, views, -- all of these go within their respective directories, and thus Rails knows what they are, and what to do with them. As mentioned previously, the Rails directory layout will not be covered in depth, but you can find lots of good information in Resources.

Create the controller

As you might expect, Rails comes with a script to generate stubs for certain types of objects. Some of the object stubs that Rails can generate for you include controllers, models, mailers, Web services, and so on. You just want to create a controller, so let's do that now. First you need to change into the Rails application directory. (see Listing 3).

Listing 3. Creating a Rails controller

```
monkey:~/Work/Rails_XML_Tutorial daniel$ cd xml_tutorial/
monkey:~/Work/Rails_XML_Tutorial/xml_tutorial daniel$ ruby script/generate controller main
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/main
      exists  test/functional/
      create  app/controllers/main_controller.rb
      create  test/functional/main_controller_test.rb
      create  app/helpers/main_helper.rb
monkey:~/Work/Rails_XML_Tutorial/xml_tutorial daniel$
```

Listing 3 calls the Ruby interpreter, asks it to run a file called generate from the script directory, and passes the arguments `controller main` to it. Rails generated a controller for you called 'main' that you can now use for your actions. As per the 'convention over configuration' mantra, each public method defined within a controller is referred to as an action and usually has an associated view. These are all accessed based on a URI convention.

For example, now you have a controller called 'main'. If you define a public method within the controller called 'index', you can now access that controller and action respectively at http://localhost:3000/main/index.

Rails has all sorts of nifty ways you can extend this routing, but suffice it to say that this is all you will look at in this tutorial. There are more interesting things to discuss -- like XML.

Create the layout and view

You don't need any fancy views here, so let's whip up something to get this running as quickly as possible. Check out the code download for some sample files.

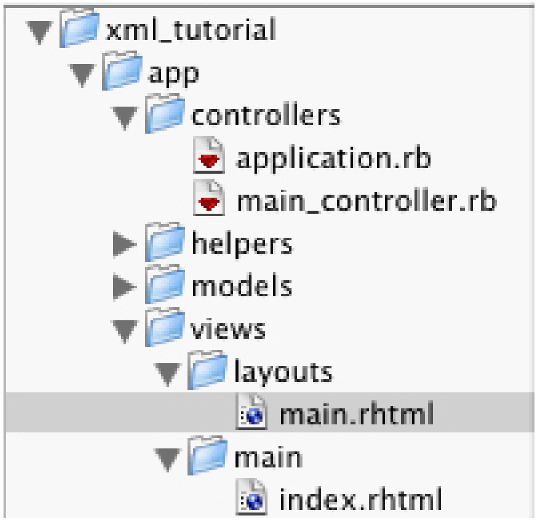
The difference between layouts and views is that a layout is an overall page layout. It's reused over and over for the common elements of your application (for example: navigation, headers, footers, and so on). Views are designed to be used within the layouts.

Create a file called main.rhtml within your layouts directory, again following convention. You name it with the same name as your controller, so Rails automatically knows that this is the layout you want to use with this controller. Then you'll create an index view that has a basic form upload field so you can upload and parse an XML document. (To save some time, copy index.rhtml from the code download.)

One last look

Take one last look at the layout of the important directories and files that you have here in Figure 1.

Figure 1. Important files and partial directory structure



```
▼ xml_tutorial
    ▼ app
        ▼ controllers
              application.rb
              main_controller.rb
        ▶ helpers
        ▶ models
        ▼ views
            ▼ layouts
                  main.rhtml
            ▼ main
                  index.rhtml
```

You have your xml_tutorial directory, which is the root directory of your Rails application. Then you have the main_controller.rb which was generated using Rails built-in generate script. You created a layout called main.rhtml and a view called index.rhtml. Notice that the view is in a directory called main. This tells Rails that all views in this directory belong to the controller called main. It's all that convention over configuration stuff again. And it's pretty great. Now, when you browse to http://localhost:3000/main/index, Rails knows that you want the following:

- Go to main_controller.rb
- Execute the index method
- Use the layouts/main.rhtml layout in the resulting display to the user
- Include the content of main/index.rhtml in the layout that you return to the user

Start up the server

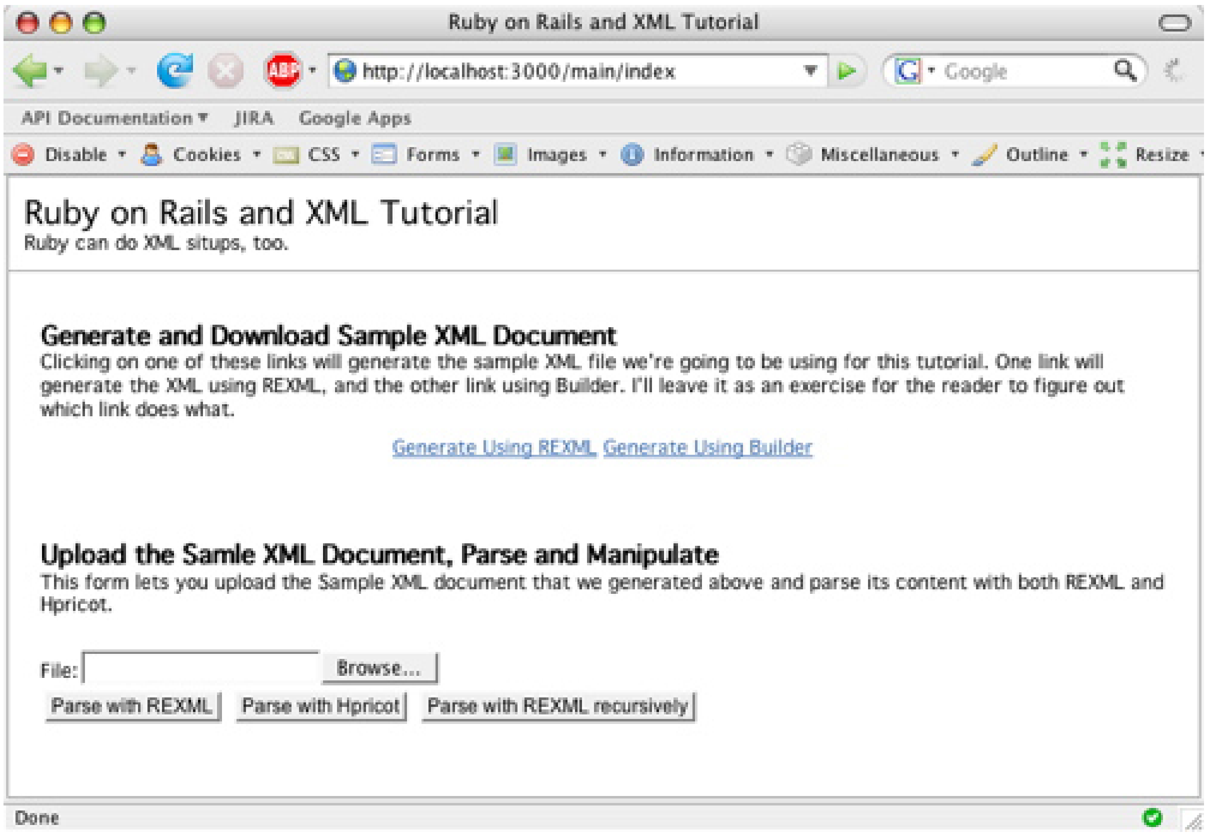
Once you've created your views, you can start up the server (depending on what software you have installed, this might start WEBrick, Lighttpd or Mongrel, none of which are discussed here). To start up the server, return to the command line (see Listing 4).

Listing 4. Starting the Rails application

```
monkey:~/Work/Rails_XML_Tutorial/xml_tutorial daniel$ ruby script/server
=> Booting lighttpd (use 'script/server webrick' to force WEBrick)
=> Rails application started on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server (see config/lighttpd.conf for options)
```

As you can see, you ran the command ruby script/server and your server is now running. Take a peek at what you get in Figure 2.

Figure 2. Your user interface



Ruby on Rails and XML Tutorial

Ruby can do XML situps, too.

Generate and Download Sample XML Document

Clicking on one of these links will generate the sample XML file we're going to be using for this tutorial. One link will generate the XML using REXML, and the other link using Builder. I'll leave it as an exercise for the reader to figure out which link does what.

Generate Using REXML Generate Using Builder

Upload the Samle XML Document, Parse and Manipulate

This form lets you upload the Sample XML document that we generated above and parse its content with both REXML and Hpricot.

File: Browse...

Parse with REXML | Parse with Hpricot | Parse with REXML recursively

How was that for some foreshadowing for the rest of the tutorial? Let's carry on!

Section 4. Creating a new XML document

You can create XML documents in several ways, anywhere from using APIs to writing it by hand. This section demonstrates how to use both REXML and Builder to generate XML documents. You'll use both APIs to generate the same document, which will give you a feel for how the syntax differs between the two APIs.

Your sample XML document

You will use a sample document that lists food for your generating and parsing examples. Listing 5 shows the basic structure of the XML document that you'll create and use.

Listing 5. Sample XML to be used throughout the tutorial

```
<Food>
  <Dish rating="2" category="singaporean">
    <DishName>Fish Head Curry</DishName>
    <WhereToBuy>Little India</WhereToBuy>
  </Dish>
```

```
...
  <Dish rating="8" category="mexican">
    <DishName>Pork Enchilada</DishName>
    <WhereToBuy>Iguana Cafe</WhereToBuy>
  </Dish>
</Food>
```

The actual XML document that you generate will have about a dozen `Dish` elements, from some of my favorite places in Singapore. This list includes a couple dishes that sound terrible just so I could give a low rating to a few dishes.

You'll generate this document based on hard-coded data that you'll just store in the controller. The data will be stored as an `Array` of `Hash` in the controller (`main_controller.rb`) so that it's easy to access while you generate the XML using REXML and Builder. Just for your piece of mind, Listing 6 contains a small chunk of how the hard-coded data in the controller has been stored.

Listing 6. Ruby data structures (Array and Hash) used to generate the sample XML

```
class MainController < ApplicationController

DISHES = [
  { :rating =>2, :category => "singaporean",
    :dish_name => "Fish Head Curry", :where_to_buy => "Little India" },
  { :rating =>8, :category => "western", :dish_name => "Cowboy Burger",
    :where_to_buy => "Brewerkz" },
  # ... Other hashes here...
]
...
```

Next, look at how you can use REXML to turn this data into XML.

REXML

REXML, as I already mentioned, is the standard XML API that is bundled with the core Ruby distribution. So you can count on it being present. It's pretty straight forward to work with, although for large pieces of XML creation programmatically, a lot of people prefer Builder, which you'll get to shortly. You can find all the sample code from the listings in this section in main_controller.rb. Let's see how to go about creating a new XML document using REXML in Listing 7.

Listing 7. Creating a new XML document using REXML

```
...
    { :rating =>8, :category => "mexican", :dish_name => "Pork Enchilada",
      :where_to_buy => "Iguana Cafe" }
  ]

  private

  def generate_rexml
     doc = REXML::Document.new
  end
```

That's not really very hard, is it? How about adding a root element? Listing 8 shows how.

Listing 8. Creating the root node in a REXML document

```
...
  def generate_rexml
     doc = REXML::Document.new
     root = doc.add_element( "Food" )
  end
end
```

Now that you have the basic structure of your sample XML document, let's iterate over your hard-coded data structure to generate the rest of the elements.

Listing 9. Using REXML to generate the sample XML content

```
...
root = doc.add_element( "Food" )
DISHES.each{ |element_data|

  dish_element = root.add_element( "Dish" )

  dish_element.add_attribute( "rating", element_data[:rating] )
  dish_element.add_attribute( "category", element_data[:category] )

  dish_name_element = dish_element.add_element( "DishName" )
  dish_name_element.add_text( element_data[:dish_name] )

  where_to_buy_element = dish_element.add_element( "WhereToBuy" )
  where_to_buy_element.add_text( element_data[:where_to_buy] )
}
...
```

Hopefully you're familiar with this Ruby syntax of iterating over an `Array`. As you can see in Listing 9, your code goes through each `Hash` within the `Array`, and creates an XML `Dish` element for each Hash. You give the `Dish` element two attributes ("rating" and "category") and two child elements ("DishName" and "WhereToBuy"). To make things clean and easy to read, you're a little verbose with your variable names and syntax. Now that you've successfully generated an XML document, you need to write it to an object so you can send it back to the client. One of the nice things about the `REXML::Document` class is that the `write` method will write its contents to any object that responds to `<< string`. Your luck continues then, because Ruby's `String` class responds to `<< string`; you just write the XML document to a `String` then (see Listing 10).

Listing 10. Writing the contents of a REXML document to a String

```
...
      where_to_buy_element.add_text( element_data[:where_to_buy] )
    }

    doc.write( out_string = "", 2 )
    return out_string

  end
end
```

Note that in Ruby, you can declare variables on the fly, which is what you've done in Listing 10. You've declared the variable `out_string` and assigned it an initial value as part of the parameters you send to `doc.write`. (The second parameter represents the indentation.) Many other languages, like Java for instance, would require this to be done in two steps, which would look something like Listing 11.

Listing 11. Alternative method for writing the contents of a REXML document to a String

```
out_string = ""
doc.write( out_string, 2 )
```

You can do it this way in Ruby as well, but it's nice to get rid of the extra line of code. In a moment, to actually see the document, you'll look at the code that hijacks the Rails response, and writes the XML document that you generated back to the client's browser as a file download. But first, see how to generate the same XML document using Builder instead of REXML.

Builder

Builder makes it unbelievably easy to write beautiful code that generates XML markup. So much so, that after you read the rest of this section, you might even want to take the rest of the week off and ponder how wonderful Builder is.

In Listing 12, look at how you create a document.

Listing 12. Creating a new XML document with Builder

```
  def generate_builder
    doc = Builder::XmlMarkup.new( :target => out_string = "",
        :indent => 2 )
  end
```

Notice here that Builder takes a target as one of the constructor parameters. With REXML, you chose a target output object after you had used the API to generate some XML. Builder just asks for the object you want to write your generated XML to up front. Notice you do the same little inline variable declaration here that you did with the REXML example.

Now is where you start to see the real power of Ruby's `method_missing` callback, and how APIs can take advantage of it. Witness how to create the `Food` element in Listing 13.

Listing 13. Creating the root element of your sample XML document with Builder

```
  def generate_builder
    doc = Builder::XmlMarkup.new( :target => out_string = "", :indent => 2 )
    doc.Food
  end
```

At this point, you might say something like - -"Wait a minute, there isn't a Food method in an XML API -- that's silly!" - and you'd be correct. What happens here is Ruby says "Whoa? I don't know about a Food method in this object, I will dispatch this call to method_missing, and let the API handle it if it wants too, if not, I'll raise an exception." So in essence, what your XML document now contains is shown in Listing 14.

Listing 14. XML result based on previous method call on Builder document

```
<Food/>
```

Now, see how you create nested elements, and attributes and text nodes in Listing 15.

Listing 15. Generating your sample XML document using Builder

```
  def generate_builder
    doc = Builder::XmlMarkup.new( :target => out_string = "", :indent => 2 )
    doc.Food {
      DISHES.each{ |element_data|
      doc.Dish( "rating" => element_data[:rating],
              "category" => element_data[:category] ){
        doc.DishName( element_data[:dish_name] )
        doc.WhereToBuy( element_data[:where_to_buy] )
        }
      }
    }
    return out_string
  end
```

That's it. This nice little chunk of code is all it takes to generate your sample XML document with Builder.

You can see that as you nest code blocks, you create nested XML elements. If you want to assign an attribute to an XML element, you can pass a Hash as a parameter, and the attributes are created. For example, in the above code, the call in Listing 16 will result in an XML element like Listing 17.

Listing 16. How to create an XML element with attributes using Builder

```
doc.Dish( "rating" => element_data[:rating],
          "category" => element_data[:category] )
```

Listing 17 shows the resulting XML element.

Listing 17. XML result based on previously listed method call

```
<Dish rating="8" category="western">
```

One scenario that your sample code doesn't cover is the case where you might want to create an element that has textual data and attributes. Imagine you wanted to create an XML element something like Listing 18.

Listing 18. A sample XML element with both attributes and text

```
<Dish rating="8" category="western">Cowboy Burger</Dish>
```

You can do this in Builder by passing a `String` argument, followed by a `Hash` argument, like this in Listing 19.

Listing 19. How to create an XML element with attributes and text using Builder

```
doc.Dish( "Cowboy Burger",
          "rating" => 8, "category" => "western" )
```

As you can see, the syntax for Builder is incredibly easy to use, and amazingly intuitive. In addition to I mentioned earlier, Table 1 cites a few special methods that are a part of Builder's `document` object.

Table 1. Special methods from Builder's document object

Method name	Action
`cdata!`	Inserts a CDATA section into the XML markup
`comment!`	Inserts an XML comment into the markup
`declare!`	Inserts an XML declaration into the markup
`instruct!`	Inserts a processing instruction into the markup
`target!`	Returns the documents target object (the object that the XML is written to)

For more information on Builder, check out the RDoc at RubyForge. Now let's take a look at how to get this generated XML back to a browser.

Section 5. Downloading the XML document

Generally speaking, the purpose of an action method within a Rails controller is to render a view back to the requesting client. The view is usually something like an RHTML or RJS file. But what if you want to grab the response directly, and send something else entirely. That's what you'll do here.

Hijacking the Rails response

Sometimes you want to render your own content back to a client, like in the case of a generated file (in this example XML). Sometimes it might be a PDF document, comma delimited values, or some other format entirely. Rails allows you to this in just a tiny fraction of code. Look at the method in main_controller.rb that's responsible for sending your generated XML file back to the client's browser as a downloaded file called sample.xml (see Listing 20).

Listing 20. Hijacking the Rails response to send custom data to the client

```
def hijack_response( out_data )
  send_data( out_data, :type => "text/xml",
:filename => "sample.xml" )
end
```

It really is that easy. In this case, `out_data` is the string instance that contains your generated XML, "text/xml" is the mime type to be set in the response, and "sample.xml" is the filename that will be presented to the client as the name of the file it attempts to download. It's worth noting that `send_data` is a protected instance method of the module `ActionController::Streaming`, which is included as a mixin by `ActionController::Base`, which you in turn inherit from as part of the class hierarchy (see Listing 21).

Listing 21. Rails controller class hierarchy

```
ActionController::Base (Part of the Rails framework)
|
 -- ApplicationController (application.rb - generated for you)
    |
     -- MainController (main_controller.rb - your controller)
```

Modules and mixins are beyond the scope of this tutorial, but check Resources for more information about these. Now that you have some XML, let's do something with it.

Section 6. Uploading a file in Rails (in this case, XML)

Imagine that you're pretty excited about this XML tutorial. Maybe you followed along, and you generated the sample XML document by clicking on some of the links. You're so excited that you show your boss the XML that was generated. But the reaction from your boss is one of disappointment. He's extremely perturbed at the very poor food ratings given to the Fish Head Curry and the Pig Organ Soup. Your boss demands that you immediately build him a Web application whereby he can upload the XML document and assign more favorable ratings to his favorite food dishes. You're in luck, because that's exactly what the rest of this tutorial is about! Let's see how you can use Rails to upload a file for processing.

View code

For the sake of being complete, let's take a quick look at the snippet of code from the view that handles the file upload form (see Listing 22).

Listing 22. RHTML form used for uploading an XML document

```
<% form_tag( {:action => 'upload'}, {:multipart
=> true} ) do -%>
<p>
  <label for="file">File:</label><%=
file_field_tag "xml_file" %><br/>
  <%= submit_tag "Parse with REXML" %>
  <%= submit_tag "Parse with Hpricot" %>
</p>
<% end -%>
```

You use the standard `form_tag` and the `helper` method that is part of the Rails framework, along with the `file_field_tag`, which generates an HTML file input tag, and `submit_tag`, which generates an HTML submit button. You'll use the values of the `submit_tag` calls to determine which handler in the controller will parse the uploaded XML document. Not very elegant, but it serves the purpose in this example. Note that you don't use the helper tags that support backing by a model, as the tutorial has no models.

Controller code

To get the contents of an uploaded file in a Rails controller is a single line of code (see Listing 23).

Listing 23. Retrieving an uploaded file from the params object in your Rails controller

```
  def upload
     uploaded_file = params[:xml_file]
  end
```

That's it. If the user uploaded a file in that file box, the resulting variable will be either an instance of `StringIO` or `File`. If the user didn't specify a file, then the result of the upload will be an empty `String`. You don't need to care about what the class of the `uploaded_file` object is, just whether or not it responds to the read method. You can conditionally declare and assign the contents of the file to a variable, if the `uploaded_file` instance responds to the `read` method (see Listing 24).

Listing 24. Reading the contents of an uploaded file into a String variable

```
def upload
  uploaded_file = params[:xml_file]
  data = uploaded_file.read if uploaded_file.respond_to? :read
end
```

After you have that, you can just delegate to whomever you want to parse the uploaded data. In this case, you either pass it off to REXML or Hpricot, depending on what button the user clicked in the browser (see Listing 25).

Listing 25. Delegation to appropriate parse method depending on user input

```
def upload
  uploaded_file = params[:xml_file]
  data = uploaded_file.read if uploaded_file.respond_to? :read

  if request.post? and data
    case params[:commit]
      when "Parse with REXML" : parse_with_rexml( data )
      when "Parse with Hpricot" : parse_with_hpricot( data )
      else parse_recursive( data )
    end
  else
    redirect_to :action => 'index'
  end

end
```

The only piece of code worth mentioning from Listing 25 is the first line of new code. The first if statement just checks to make sure that the request is a `POST` (as opposed to a `GET`) and that you actually have some data that was uploaded (make sure that the user actually selected a file to upload). That's about it! Now let's see what sort of code you need to write to actually parse and manipulate this uploaded XML!

Section 7. Parsing and manipulating XML

Now that your boss is upset about the low-scoring Fish Head Curry, you're anxious to get on with the ability to parse and manipulate XML. But you also know from previous experience that these XML APIs seem to be user-friendly. So you're ready to jump in and get some higher ratings for those dishes, to make your boss happy.

The goal here is to parse the uploaded XML document, search and iterate as needed to find the Fish Head Curry and Pig Organ Soup dishes, and upgrade their ratings to a 6. You hope that will improve the humor of your boss, and with any luck, he'll start to notice the great code, instead of the low-scoring food dishes.

Parse with REXML

In this section you parse to your sample document with REXML and update the relevant elements appropriately. With REXML it is extremely easy to navigate through an XML document as all the elements and attributes are accessible to you array-style, or through XPath queries, or by simple iteration. I list some other great REXML articles in the Resources. But for now, let's get to work. First you'll use an XPath query to search for all of the `DishName` elements in the document (see Listing 26).

Listing 26. Using REXML's XPath class to search for DishName elements

```
  def parse_with_rexml( xml_data )
    doc = REXML::Document.new( xml_data )

    REXML::XPath.each( doc, "//DishName" ){
|dish_name_element|
      #... your code here ...
      #... do work with elements ...
    }

    doc.write( out_string = "", 2 )
    hijack_response( out_string )
  end
```

That simple bit of code will find all `DishName` elements in the document and iterate over them, allowing you to put whatever custom code you want into the block, to do whatever work you need to do with the elements. The text `//DishName` is an XPath query that basically means "get me all DishName elements in this document."

In this particular case, you want to check the text value of the element to see if it is equal to "Fish Head Curry" or "Pig Organ Soup" (see Listing 27).

Listing 27. Retrieving the text value of a given XML element

```
if dish_name_element.text == "Fish Head Curry" or
   dish_name_element.text == "Pig Organ Soup"
  #... your code here ...
  #... do work with elements ...
end
```

Calling the `text` method on a `REXML` element will return the String value of the first child text element (if one exists) or nil otherwise.

Once you have the correct `DishName` element, you want to get the parent. Remember your document structure (see Listing 28).

Listing 28. Partial snippet from sample XML

```
<Food>
  ...
  <Dish rating="2" category="singaporean">
    <DishName>Fish Head Curry</DishName>
    <WhereToBuy>Little India</WhereToBuy>
  </Dish>
  ...
</Food>
```

The parent element is the `Dish` element, and to access it, simply call the `parent` method (see Listing 29).

Listing 29. Retrieving an XML element's parent element using REXML

```
parent = dish_name_element.parent
```

Once you have the parent element, you can access the `rating` attribute, and assign it the new value (see Listing 30).

Listing 30. Assigning a new value to an attribute in an XML element using REXML

```
parent.attributes["rating"] = 6
```

And voilà! You just made your boss happier! Take a quick look at the entire block of code in Listing 31.

Listing 31. Code block to navigate, parse and modify XML elements using REXML

```
XPath.each( doc, "//DishName" ){ |dish_name_element|
  if dish_name_element.text == "Fish Head Curry" or
     dish_name_element.text == "Pig Organ Soup"
    parent = dish_name_element.parent
    parent.attributes["rating"] = 6
  end
}
```

Now, because you feel really cool, and want to impress your boss after that rating fiasco, you decide to try to write this rate-changing XML manipulating snippet again, in a single line of code. By now you feel like a Ruby hero, so it's a snap (separated onto multiple lines for readability). See Listing 32.

Listing 32. An alternative approach to parse and manipulate the sample XML in a single line of code

```
doc.root.each_element( "//DishName" ){ |e|
  e.parent.attributes["rating"] = 6
    unless ["Fish Head Curry", "Pig Organ Soup"].index( e.text ).nil? }
```

This method doesn't use an XPath explicitly; rather it uses the `each_element` method, which is part of REXML's `Element` class. If you wanted to read this code as an English sentence, left to right, it would read something like this (important words bolded):

"Find **each DishName** element, and **assign** the parent element's **rating** attribute a value of **six, unless** the text of the DishName element is **not** 'Pig Organ Soup' or 'Fish Head Curry'".

Now let's have some fun with Hpricot!

Parse with Hpricot

Hpricot is technically an HTML parser, not an XML parser, but because HTML and XML use essentially the same syntax, Hpricot happily parses any XML document you send to it. Many people choose Hpricot over REXML for its speed (the scanner is written in C) and *syntactical sugar* as many Rubyists like to say.

A caveat to working with Hpricot on the XML front is that if you have case-sensitive data, you might have some issues. Version 0.5 (the latest) even when parsing a document as XML explicitly, converts all element names to lowercase. A ticket in the Hpricot Trac (#53) asks to address this, but it hasn't happened yet. You want to be aware of this if you start to play with Hpricot yourself. Look at the entire snippet to parse the document and assign the new ratings, it's very similar to REXML (see Listing 33).

Listing 33. Parsing, navigating and manipulating an XML document using Hpricot

```
  def parse_with_hpricot( xml_data )
    doc = Hpricot.XML( xml_data )
    (doc/:dishname).each{ |dish_name_element|
      if dish_name_element.inner_html == "Fish Head Curry" or
         dish_name_element.inner_html == "Pig Organ Soup"
        parent = dish_name_element.parent
        parent.attributes["rating"] = "6"
      end
    }
  end
```

(Don't forget to require rubygems and hpricot at the top of the main_controller.rb file!)

The key piece of interest in Listing 33 is this call in Listing 34.

Listing 34. Hpricot's divisor method

```
(doc/:dishname)
```

In Ruby, everything is an object. Ruby has no primitives, which is one reason why you can actually use the divisor (/) as a method name. The divisor method in Hpricot is just an alias for the search method, so another way to do exactly the same thing as in Listing 34 is shown in Listing 35.

Listing 35. Hpricot's search method (also aliased as /)

```
doc.search(:dishname)
```

It's good to remember that parenthesis to method calls are optional in Ruby. You can take them or leave them depending on what you find more readable. Other than that difference, you'll notice that the code for manipulating the XML document is nearly identical.

You can do all sorts of impressive things with Hpricot's CSS and XPath selectors, and for more advanced usage, I recommend that you look at all the examples on the Hpricot Web site.

Parsing XML recursively with REXML

Sometimes you need to parse an XML document, and you might not know the structure or content or what you're looking for ahead of time. In that case you're not necessarily looking for specific elements, or you might just want to build up some in-memory data structure. Using REXML you can do this with unbelievable ease (see Listing 36).

Listing 36. Iterate over an entire XML document recursively

```
  def parse_recursive( xml_data )
    doc = REXML::Document.new( xml_data )
    root = doc.root
    root.each_recursive{ |element|
      logger.info "Element: #{element}"
    }
    redirect_to :action => 'index'
  end

end
```

(To see the output, check the `log/development.log` file. You'll see it amongst the other messages.)

Technically, the code that actually recurses the entire document and logs each element is a one-liner. It has been split into three lines to make it look like it actually took some work, but alas -- you can reduce the core to this single line of code (see Listing 37).

Listing 37. Iterate over an entire XML document recursively in one line of code

```
                      root.each_recursive{ |element| logger.info "Element: #{element}" }
```

The REXML API includes many other convenient methods. For example, if you just wanted to iterate over a given element's direct descendents instead of the entire document recursively, you can just do the following from Listing 38.

Listing 38. Iterate over a given XML elements direct descendants (immediate children)

```
doc = REXML::Document.new( xml_data )
root = doc.root
```

```
root.each_element{ |child| logger.info "Child Element: #{child}" }
```

And that brings you to the end of your XML, Ruby and Rails goodness.

Section 8. Summary

Wrap up

In this tutorial, you generated a Rails application stub and created one controller to handle requests that generate and manipulate an example XML document. You saw how to generate XML content with REXML and Builder, and how to navigate and manipulate XML content with REXML and Hpricot. Additionally, you looked briefly at how to handle file uploads in Rails, and how to hijack a Rails response object, in order to serve something to the client other than a Rails view (such as generated XML data).

Downloads

Description	Name	Size	Download method
Tutorial source code	x-rubyonrailsxml-source.zip	2038KB	HTTP

Information about download methods

Resources

Learn

- Ruby on Rails: Visit the official Ruby on Rails Web site.
- Ruby: Also, visit the official Ruby Programming Language Web site.
- Processing XML in Ruby (Koen Vervloesem, XML.com, November 2005): In this great article, find plenty of examples of creating, parsing and manipulating XML using REXML.
- The Poignant Guide to Ruby: Dig into an excellent and free online Ruby resource.
- Humble Little Ruby Book: Walk through the basics of working with Ruby and much more in another excellent and free online Ruby resource.
- XML Matters: The REXML Library (David Mertz, developerWorks, March 2002): Read how to process XML with the REXML library and tailor the library to your programming language as you develop an XML application.
- Four cool libraries for Ruby (Pat Eyler, developerWorks, January 2006): Improve your Ruby code as you learn to use four members of Ruby's standard library -- RDoc, WEBrick, dRuby, and REXML -- more effectively.
- What's the secret sauce in Ruby on Rails? (Bruce Tate, developerWorks, May 2006): Read about the Ruby on Rails framework in this introductory discussion.
- Ruby on Rails and J2EE: Is there room for both? (Aaron Rustad, developerWorks, July 2005): Compare Rails and J2EE in the technology industry.
- Introduction to XML (Doug Tidwell, developerWorks, August 2002): Learn the basic concepts behind XML in this popular tutorial.
- Understanding DOM (Nicholas Chase, developerWorks, updated March 2007): Learn to refer to, retrieve, and change items within an XML structure through the Document Object Model (DOM).
- Ajax for Java developers: Exploring the Google Web Toolkit (Philip McCarthy, developerWorks, June 2006): Develop Ajax applications from a single Java codebase in this introduction to GWT's comprehensive set of APIs and tools for creating dynamic Web applications almost entirely in Java code.
- Build an Ajax application using Google Web Toolkit, Apache Derby, and Eclipse: Read the developerWorks article series by Noel Rappin:
    - Part 1: The fancy front end: (December 2006): See how to build the front end of a sample delivery system.
    - Part 2: The reliable back end (January 2007): Read how to create a relational database using Derby, and a bare-bones mechanism to convert the database rows to Java objects.

    - Part 3: Communication (February 2007): With the Remote Procedure Call (RPC) framework within GWT, get the client and server talking to each other.
    - Part 4: Deployment (February 2007): Learn to deploy your GWT app within a Java Web application server and find tips on using the Apache Derby database to drive the GWT.

- YouTube's developer APIs: Learn how to add online videos from YouTube into your application.
- Amazon Web Services: Explore this suite of web services to enrich your applications.
- IBM XML certification: Find out how you can become an IBM-Certified Developer in XML and related technologies.
- XML technical library: See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- developerWorks technical events and webcasts: Stay current with technology in these sessions.
- developerWorks XML zone: Explore hundreds of articles and tutorials about XML.
- The technology bookstore: Browse for books on these and other technical topics.

Get products and technologies

- Ruby Gems: Check out the official Ruby Gems Web site.
- Hpricot: At the official Hpricot Web site, find links to good documentation and examples.
- Builder: Get the RDoc pages for the Builder API, including good documentation.
- REXML: Explore the official REXML Web site, including tutorials and documentation.
- IBM trial software: Build your next development project with trial software available for download directly from developerWorks.

Discuss

- developerWorks XML forums: Communicate with other XML developers trying to solve the same problems you are.
- developerWorks blogs: Get involved in the developerWorks community.

About the author

Daniel Wintschel
Daniel Wintschel is a technology agnostic coffee drinker who loves solving problems for people and businesses. He's done a whole lot of Java programming (~7 years), and is starting to do a whole lot of Ruby programming (~1.5 years). He is the co-founder of Helium Syndicate, a company dedicated to building best of breed software solutions for small to medium sized businesses. When he's not writing software, he's likely eating, drinking coffee, or wishing he were writing software.