DZone
REFCARDZ
tech facts at your fingertips

CONTENTS INCLUDE:

Flexible Rails: Flex 3 on Rails 2

By Peter Armstrong

ABOUT FLEXIBLE RAILS

Simply put, Flex is the most productive way to build the UI of Rich Internet Applications, and Rails is a very productive way to rapidly build a database-backed CRUD application, thanks to ActiveRecord (the ORM layer of Rails) and thanks to the principles of Convention Over Configuration and DRY (Don't Repeat Yourself). This refcard shows you how to get started. It provides an overview of Flex and Rails, how they can be used together and then building a simple Flex + Rails application using XML over HTTPService to have the Flex client talk to a RESTful Rails server. Since we'll use RESTful Rails controllers, the Rails controller methods will also support the traditional HTML views.

OVERVIEW OF RAILS 2

Rails provides a standard three-tier architecture (presentation tier, model tier, persistence tier) as well as a Model-View-Controller (MVC) architecture. As shown in Figure 1, Rails takes care of everything between the web server and the database.

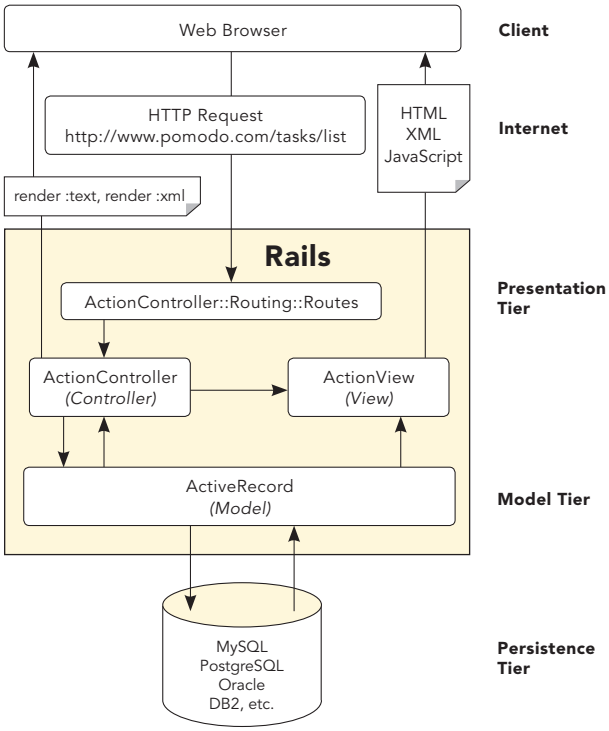


Figure 1: Rails provides a standard three-tier architecture (presentation tier, model tier, persistence tier) as well as a Model-View-Controller architecture.

The typical sequence is as follows:

1. A user visits a particular URL in their web browser (makes an HTTP request).
2. This request goes over the Internet to the web server in which Rails is running.
3. That web server passes the request to the routing code in Rails, which triggers the appropriate controller method call based on the routes defined in **config\routes.rb**.
4. The controller method is called. It communicates with various ActiveRecord models (which are persisted to and retrieved from a database of your choosing). The controller method can then do one of two things:
   1. Set some instance variables and allow a view template (a specially named .html.erb file, for example) to be used to produce HTML, XML, or JavaScript, which is sent to the browser.
   2. Bypass the view mechanism and do rendering directly via a call to the render method. This method can produce plain text (**render :text => "foo"**), XML (**render :text => @task**), and so on.

OVERVIEW OF FLEX 3

In Flex 3, you write code in MXML (XML files with a .mxml extension; M for Macromedia) and ActionScript (text files with a .as extension) files and compile them into a SWF file, which runs in the Flash player. This SWF is referenced by an HTML file, so that when a user with a modern web browser loads the HTML file, it plays the Flash movie (prompting the user to download Flash 9 if it's not present). The SWF contained in the web page can interact with the web page it's contained in and with the server it was sent from.

FLEX 3 AND RAILS 2 TOGETHER

Flex and Rails can be used together with XML over HTTPService or with Action Message Format. The XML over HTTPService approach is shown in Figure 2 below.

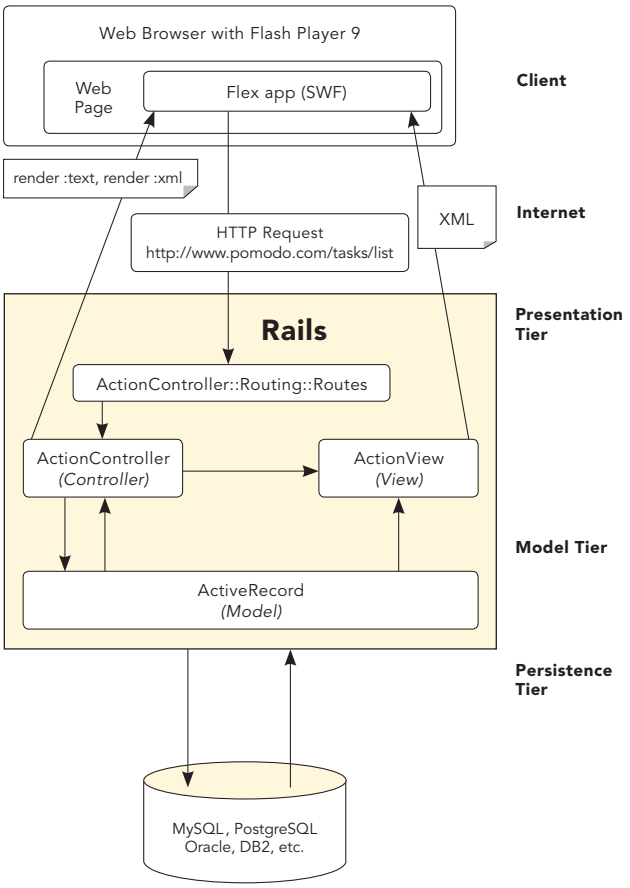


Figure 2.

The AMF waters are a bit muddier: there are currently three ways that Flex can talk to Rails using AMF and RemoteObject:

- RubyAMF
- WebORB for Rails
- BlazeDS (with Rails running on JRuby).

FYI

Flash 9? Are you kidding me?

The reference to Flash 9 earlier may have set off alarm bells in your head: "Isn't Flash 9 somewhat new? How many people will be able to run my app?" Well, while not everyone has Flash 9, most do: according to **http://www.adobe.com/products/player_census/flashplayer/version_penetration.html** Flash 9 has reached near ubiquity: 97.3% in US/Canada, 96.5% in Europe and 98.0% in Japan. This is better than Windows.

INSTALLING EVERYTHING

To get started, various software packages need to be installed. The full instructions can be found in chapter 2 of *Flexible Rails* (Manning Publications). Here's what is needed:

1. Ruby 1.8.6
2. RubyGems 1.0.0 (or higher)
3. Rails. 2.0.2 (or higher)
4. Flex Builder 3
5. SQLite
6. The sqlite3 gem, installed by running this command:
   `C:\>gem install sqlite3-ruby`

Resource	URL	Notes
RubyonRails	http://rubyonrails.org/down	
HiveLogic	http://hivelogic.com/articles/2008/02/ruby-rails-leopard	
SQLite	http://www.sqlite.org/download.html	On Windows, download *sqlite-3_5_5.zip* or higher and *sqlitedll-3_5_5.zip* or higher, and unzip both of them into C:\WINDOWS\system32

BUILDING A FLEX + RAILS APPLICATION

The world doesn't need Yet Another Todo List, but let's build one. Unlike most Rails tutorials, we will assume you are using Windows. (Rails has "crossed the chasm", so this is now becoming the correct assumption.)

Open a command prompt or Terminal window and run the following commands:

```
C:\>rails todo
```

This installs the SQLite3 gem and then creates a new Rails application which by default uses the SQLite database. (The default in Rails 2.0.1 and below was MySQL.)

Next, create a couple of directories:

```
C:\>cd todo
C:\todo>mkdir app\flex
C:\todo>mkdir public\bin
```

Next, switch to Flex Builder 3 and create the new Flex project:

1. Do File > New > Flex Project...
2. Choose to create a new project named Todo in c:\todo
3. Leave its type set to "Web application" and its "Application server type" set to None and click Next
4. Set the Output folder of the Flex project to public\bin and click Next
5. Set the "Main source folder" to app\flex, leave the "Main application file" as Todo.mxml and set the output folder to http://localhost:3000/bin and click Finish. Your new Flex project will be created. (Note that public isn't part of the path since it's the root; 3000 is the default port for the server).

We are using app\flex as the root of all our Flex code—in larger team environments it's advisable to create a Flex project as a sibling of the Rails app (say, c:\todoclient) and set its output folder to go inside c:\todo\public\bin. This way, different team members can use different IDEs for the client and server projects: for example, Aptana and Flex Builder are both Eclipse-based, and interesting things can happen when you nest projects.

Building a Flex + Rails Application, continued

Next, let's create a new Task resource using the now-RESTful scaffold command.

```
C:\todo>ruby script\generate scaffold Task name:string
```

Here we are creating a Task that has a name attribute, which is a string. Running this command generates the various Rails files, including the model, helper, controller, view templates, tests and database migration. We're going to make the simplest Todo list in history: Tasks have names, and nothing else. Furthermore, there are no users even, just a global list of tasks.

The Task model looks like this:

```
class Task < ActiveRecord::Base
end
```

Because the Task model extends (with <) ActiveRecord::Base, it can be mapped to the equivalent database tables. Because we also created the controllers and views with the script\generate scaffold command and ensured that we specified all the fields, we can use a prebuilt web interface to Create, Read, Update, and Delete (CRUD) them.

The CreateTasks migration that was created (in db\migrate\001_create_tasks.rb) looks like this:

```
class CreateTasks < ActiveRecord::Migration
  def self.up
    create_table :tasks do |t|
      t.string :name
      t.timestamps
    end
  end
  def self.down
    drop_table :tasks
  end
end
```

CreateTasks Class	Extends ActiveRecord::Migration
Up method	Creates a new tasks table with the create_table method call, which takes a block that does the work
Down method	Deletes it with the drop_table call

In the up method, we specify the data types of each new column, such as string in our case, or boolean, integer or text. These are then mapped to the equivalent database data types: for example, boolean becomes a tinyint(1) in MySQL. The timestamps call adds two columns: created_at and updated_at, which Rails treats specially, ensuring that they're automatically set. This is often a good thing to have, so we'll leave them there even though they won't be needed in this build.

The TasksController (in app\controllers\tasks_controller.rb) looks like this:

```
class TasksController < ApplicationController
  # GET /tasks
  # GET /tasks.xml
  def index
    @tasks = Task.find(:all)
    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @tasks }
    end
  end
  # GET /tasks/1
  # GET /tasks/1.xml
  def show
    @task = Task.find(params[:id])
    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @task }
    end
  end
  # GET /tasks/new
  # GET /tasks/new.xml
  def new
    @task = Task.new
    respond_to do |format|
      format.html # new.html.erb
      format.xml  { render :xml => @task }
    end
  end
  # GET /tasks/1/edit
  def edit
    @task = Task.find(params[:id])
  end
  # POST /tasks
  # POST /tasks.xml
  def create
    @task = Task.new(params[:task])
    respond_to do |format|
      if @task.save
        flash[:notice] = 'Task was successfully created.'
        format.html { redirect_to(@task) }
        format.xml  { render :xml => @task,
          :status => :created, :location => @task }
      else
        format.html { render :action => "new" }
        format.xml  { render :xml => @task.errors,
          :status => :unprocessable_entity }
      end
    end
  end
  # PUT /tasks/1
  # PUT /tasks/1.xml
```

→

Building a Flex + Rails Application, continued

```
  def update
    @task = Task.find(params[:id])
    respond_to do |format|
      if @task.update_attributes(params[:task])
        flash[:notice] = 'Task was successfully updated.'
        format.html { redirect_to(@task) }
        format.xml  { head :ok }
      else
        format.html { render :action => "edit" }
        format.xml  { render :xml => @task.errors,
          :status => :unprocessable_entity }
      end
    end
  end
  # DELETE /tasks/1
  # DELETE /tasks/1.xml
  def destroy
    @task = Task.find(params[:id])
    @task.destroy
    respond_to do |format|
      format.html { redirect_to(tasks_url) }
      format.xml  { head :ok }
    end
  end
  end
```

This new controller which was generated for us contains the seven RESTful controller methods, which are explained in the following table (inspired by David Heinemeier Hansson's *Discovering a World of Resources on Rails* presentation—**media.rubyonrails.org/presentations/worldofresources.pdf, slide 7**—as well as the table on p. 410 of *Agile Web Development with Rails*, 2nd ed. (The Pragmatic Programmers), and the tables in Geoffrey Grosenbach's REST cheat sheet **http://topfunky.com/clients/peepcode/REST-cheatsheet.pdf)**:

#	Method	Sample URL paths	Pretend HTTP Method	Actual HTTP Method	Corresponding CRUD Method	Corresponding SQL Method
1	index	/tasks /tasks.xml	GET	GET	READ	SELECT
2	show	/tasks/1 /tasks/1.xml	GET	GET	READ	SELECT
3	new	/tasks/new /tasks/new.xml	GET	GET	—	—
4	edit	/tasks/1/edit	GET	GET	READ	SELECT
5	create	/tasks /tasks.xml	POST	POST	CREATE	INSERT
6	update	/tasks/1 /tasks/1.xml	PUT	POST	UPDATE	UPDATE
7	destroy	/tasks/1 /tasks/1.xml	DELETE	POST	DELETE	DELETE

Table 1: The seven standard RESTful controller methods

FYI

What's REST?

REST (Representational State Transfer) is a way of building web services that focuses on simplicity and an architecture style that is "of the web." This can be described as a Resource Oriented Architecture (ROA); see *RESTful Web Services* published by O'Reilly Media for details. Briefly, the reason to use a RESTful design in Rails is that it helps us organize our controllers better, forces us to think harder about our domain, and gives us a nice API for free.

Next, we run the new migration that was created (CreateTasks) when we ran the scaffold command:

```
C:\todo>rake db:migrate
```

At this point we run the server:

```
C:\todo>ruby script\server
```

and play with creating, editing and deleting tasks.

1. Go to http://localhost:3000/tasks to see an empty task list.
2. Click the New link to go to http://localhost:3000/tasks/new.
3. Create a new Task with a name of "drink coffee" and click Create.
4. Go back to http://localhost:3000/tasks to see the task list with the new "drink coffee" task present.

Now, let's do something interesting and hook this up to Flex. Currently, the Todo.mxml file looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/
2006/mxml" layout="absolute">
</mx:Application>
```

The top-level tag is mx:Application; the root of a Flex application is always an Application. The mx: part identifies the XML namespace that the Application component is from. By default, an Application uses an absolute layout, where you specify the x,y of each top level container and component.

What we want to build is the following application:

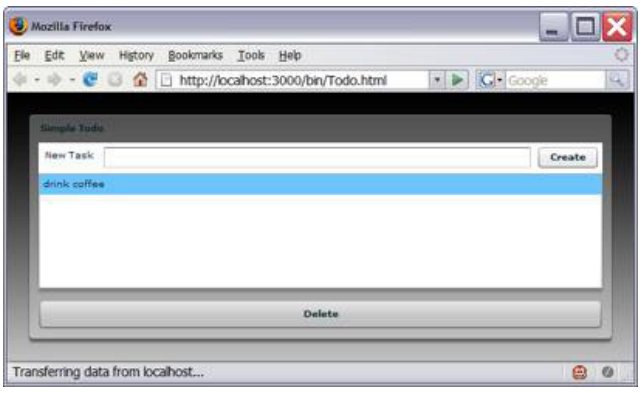


Figure 3: The Simple Todo Flex Application

Building a Flex + Rails Application, continued

We want the ability to create new tasks, delete tasks and rename them inline in the list. Furthermore, we want to do this in the least amount of code possible. Normally, I'd build this iteratively; but we'll build it all at once. Modify the Todo.mxml file to look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  width="100%" height="100%" layout="vertical"
  backgroundGradientColors="[#000000, #CCCCCC]"
  creationComplete="svcTasksList.send()">
<mx:Script>
<![CDATA[
  import mx.events.ListEvent;
  import mx.controls.Alert;
  import mx.rpc.events.ResultEvent;
  private function createTask():void {
    svcTasksCreate.send();
  }
  private function deleteTask(task:XML):void {
    svcTasksDestroy.url = "/tasks/" + task.id + ".xml";
    svcTasksDestroy.send({_method: "DELETE"});
  }
  private function updateSelectedTask(event:ListEvent):
  void {
    var itemEditor:TextInput =
      TextInput(event.currentTarget.itemEditorInstance);
  var selectedTask:XML = XML(event.itemRenderer.data);
  if (selectedTask.name == itemEditor.text) return;
  var params:Object = new Object();
  params['task[name]'] = itemEditor.text;
  params['_method'] = "PUT";
  svcTasksUpdate.url = "/tasks/"+ selectedTask.id +".xml";
  svcTasksUpdate.send(params);
}
private function listTasks():void {
  svcTasksList.send();
}
]]>
</mx:Script>
  <mx:HTTPService id="svcTasksCreate" url="/tasks.xml"
    contentType="application/xml" resultFormat="e4x"
    method="POST" result="listTasks()">
    <mx:request>
      <task><name>{newTaskTI.text}</name></task>
    </mx:request>
  </mx:HTTPService>
  <mx:HTTPService id="svcTasksList" url="/tasks.xml"
    resultFormat="e4x" method="POST"/>
  <mx:HTTPService id="svcTasksUpdate" resultFormat="e4x"
    method="POST" result="listTasks()"/>
  <mx:HTTPService id="svcTasksDestroy" resultFormat="e4x"
    method="POST" result="listTasks()"/>
  <mx:XMLListCollection id="tasksXLC"
    source="{XMLList(svcTasksList.lastResult.children())}"/>
  <mx:Panel title="Simple Todo" width="100%"
  height="100%">
    <mx:HBox width="100%" paddingLeft="5"
    paddingRight="5"
      paddingTop="5">
      <mx:Label text="New Task"/>
      <mx:TextInput id="newTaskTI" width="100%"
        enter="createTask()"/>
      <mx:Button label="Create" click="createTask()"/>
    </mx:HBox>
    <mx:List id="taskList" width="100%" height="100%"
      editable="true" labelField="name"
      dataProvider="{tasksXLC}"
      itemEditEnd="updateSelectedTask(event)"/>
    <mx:ControlBar width="100%" horizontalAlign="center">
      <mx:Button label="Delete" width="100%" height="30"
        enabled="{taskList.selectedItem != null}"
        click="deleteTask(XML(taskList.selectedItem))"/>
    </mx:ControlBar>
  </mx:Panel>
</mx:Application>
```

This is a complete Flex application in 67 lines of code! Compile and run the application by clicking the green "play" button: you will see the screen shown in Figure 3.

A Quick Highlight Tour of this Code:

1. We use a vertical layout to make the components flow vertically. Other choices are horizontal (for horizontal flow) and absolute (which we saw before). The backgroundGradientColors specify the start and end of the gradient fill for the background.

2. We define a HTTPService svcTasksList, which does a GET to /tasks.xml (thus triggering the index action of the TasksController) and specifies a resultFormat of e4x so the result of the service can be handled with the new E4X XML API. We then take the lastResult of this service, which is an XML document, get its children (which is an XMLList of the tasks), and make this be the source of an XMLList-Collection called tasksXLC. We do this with a binding to the source attribute. Similarly, we define svcTasksCreate, svcTasksUpdate and svcTasksDestroy to be used for the other CRUD operations.

3. We can pass data to Rails via data bindings using curly braces { } inside XML (as shown in svcTaskCreate) or by sending form parameters that Rails would be expecting (as shown in the updateSelectedTask method).

4. For svcTasksUpdate and svcTasksDestroy we are not setting the url property statically, but instead dynamically setting it to include the id of the task we are updating or destroying.

5 A hack: you can't send HTTP PUT or DELETE from the Flash player in a web browser, so we need to fake it. Luckily, since you can't send PUT or DELETE from HTML in a web browser either, Rails already has a hack in place—we just need to know how to use it. Rails will look for a _method parameter in its params hash and, if there is one, use it instead of the actual HTTP method. So, if we do a form POST with a _method of PUT, Rails will pretend we sent an HTTP PUT. (If you're thinking that it's ironic that at the core of a "cleaner" architecture is a giant hack, well, you're not alone.) This _method can be added to a params Object **(params['_method'] = "PUT";)** or to an anonymous object **(svcTasksDestroy.send({_method: "DELETE"});)** If you're new to Flex, {} can be used for both anonymous object creation and for data binding. Think of an anonymous object like a hash in Ruby.

RESOURCES

Organization	URL	Information
Manning Publications	http://www.manning.com/armstrong	Book: *Flexible Rails*—provides a code-focused, iterative tutorial introduction to using Flex with Rails.
	http://www.manning.com/armstrong2	Book: *Enterprise Flexible Rails*—provides the definitive tutorial introduction to the Ruboss Framework.
Ruboss Technology Corporation	http://www.ruboss.com	The Ruboss Framework: The Open Source Framework Which Puts Flex On Rails. Ruboss taking the Flex + Rails combination further, making it even easier and more accessible to the Enterprise.

ABOUT THE AUTHOR

Peter Armstrong

Peter Armstrong is the CEO and Co-Founder of Ruboss Technology Corporation (http://ruboss.com), a self-funded RIA startup in Vancouver, BC, Canada. Peter is also the co-creator of the open source Ruboss Framework, the RESTful way to develop Adobe Flex and AIR applications that easily int egrate with Ruby on Rails. Peter is the author of *Flexible Rails* and *Enterprise Flexible Rails*. Peter has been developing rich client applications for over 7 years. He has worked with Flex full-time since July 2004, including being a key part of the Dorado Product Engineering team that won the 2006 Adobe MAX Award for RIA/Web Development (http://my.adobe.acrobat.com/doradomaxpresentation/). He is the organizer of The Vancouver Ruby/Rails Meetup group and is a frequent conference speaker on using Flex and Rails together.

Blog
http://peterarmstrong.com/

RECOMMENDED BOOK

Flexible Rails is a book about how to use Ruby on Rails and Adobe Flex to build next-generation rich Internet applications (RIAs). The book takes you to the leading edge of RIA development, presenting examples in Flex 3 and Rails 2.

BUY NOW
books.dzone.com/books/flexible-rails

Want More? **Download Now.** Subscribe at refcardz.com

Upcoming Refcardz:

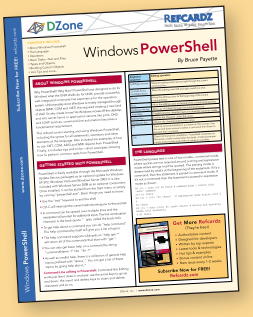
- C#
- GlassFish Application Server
- RSS and Atom
- Apache Struts 2
- MS Silverlight 2
- NetBeans IDE 6 Java Editor
- Groovy

Available:

Published June 2008
- jQuerySelectors
- Design Patterns

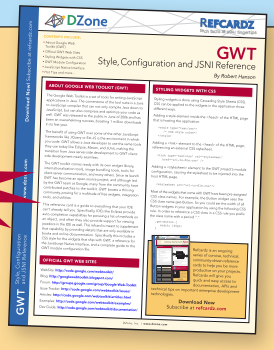
Published May 2008
- Dependency Injection in EJB 3

Published April 2008
- Spring Configuration
- Getting Started with Eclipse
- Getting Started with Ajax



FREE

Windows PowerShell
Published May 2008



GWT Style, Configuration and JSNI Reference
Published April 2008

DZone

DZone communities deliver over 3.5 million pages per month to more than 1.5 million software developers, architects and designers. DZone offers something for every developer, including news, tutorials, blogs, cheatsheets, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-08-0
ISBN-10: 1-934238-08-2
50795
9 781934 238080
$7.95