Thesis Paper

GAIN KNOWLEDGE OF THE PROGRAMMING LANGUAGE - “RUBYONRAILS”

Thesis Supervisor: **Matin Saad Abdullah (Senior Lecturer)**

Department of Computer Science and Engineering

Semester: Spring, Year: 2007

Submitted By:

Rozina Sultana
Student’s ID: 02101015

Md.Fazle Munim
Student’s ID: 04241004

BRAC University, 66 Mohakhali • Dhaka 1212 • Phone: 988 1265 • Fax: 881 0383 •
http://www.bracuniversity.net

DECLARATION

In accordance with the requirements of the degree of Bachelor of Computer Science in the division of Computer Science and Engineering, I present the following thesis entitled “Gain knowledge of the Programming Language - RubyOnRails”. This work was performed under the supervision of Matin Saad Abdullah (Senior Lecturer) and my group member was Md. Fazle munim.

I hereby declare that the work submitted in this thesis is our own and based on the results found by ourselves. Materials of work found by other researcher and the helps by the other people are mentioned on references. This thesis, neither in whole nor in part, has been previously submitted for any degree.

Student	**Supervisor**	**Chairperson**
		Department of Computer Science & Engineering

ACKNOWLEDGMENT

Our heartiest thank goes to our thesis supervisor who always encouraged us to do such a thesis. He helped us in all aspects when we got stuck, and he pushed us forward always. He gave us all kinds of support through out our thesis period. He provided us all kind resources we needed for our thesis. We got all kind of inspiration and courage to do thesis from him.

Our specials thank goes to Dr. Mumit Khan (Associate Professor, BRAC University), Gina Blaber (Vice President, Oreilly Conferences), Patrick Dirden (Vice President, Oreilly Conferences) and PuneRuby Groups (RubyOnRails Community in India) for assist me many time with resources and guidance.

ABSTRACT

This is a learning basis thesis work on RubyOnRails. In this thesis I mainly concentrate on how the language works and it's usability to the users. I also tried to discover the hidden facts of the language and its computability over other framework and usability in different environments. I also uncover a clear idea about RubyOnRails in compare and construct with other language. With that it is also supportive to present why RubyOnRail work well than the traditional languages.

TABLE OF CONTENTS

No	Title		Page
1.0	**Introduction**		1
2.0	**Essentials and Basic Introduction**		2
	Ruby		
2.1		What is Ruby	2
2.2		History of Ruby	2
	Rails framework		
2.3		What is Rails	2
	RubyOnRails (RoR)		
2.4		What is Ruby on Rails	3
3.0	**Basic Structures And Architectures**		5
3.1		Software Architecture of Ruby on Rails	5
3.2		A framework user's–eye view of application development	6
3.3		MVC framework concept	7
3.4		Analyzing Rails' implementation of MVC	8
3.5		Overview of how Rails implements the MVC framework design	8
3.6		Schematic view of Ruby and the Rails framework	9
3.7		Tracing the lifecycle of a Rails run	9
4.0	**Component/Services Of RoR**		11
4.1		Component/Service Details	11
4.2		The functional requirements of the system	12
4.3		Rails Application Structure	12
4.4		Directory Structure Description	14
4.5		Rails Statistics:	18
5.0	**Features and Qualities**		19
5.1		Features of Ruby	19
5.2		Fundamentals of Ruby	21
5.3		Key Quality Attributes of Rails	22

5.4		How each quality goal achieved in the system	25
6.0	**User's Compatibility and Usability**		26
6.1		What makes Ruby so popular?	26
6.2		Rails aims to take the burden off developers	26
6.3		Ten Reasons Ruby Is Cool	27
6.4		Why Rails	36
6.5		Rails three major goals	37
7.0	**Compare and Contrasts with Other Languages**		38
7.1		Ruby versus Other Languages	38
7.2		Contrast ruby and other language	38
7.3		RoR Vs PHP	41
7.4		Why Ruby on Rails is better than PHP	41
7.5		Criticizes on RoR and its reply	41
7.6		Comparison based on Developers choice	43
7.7		Ruby on Rails and J2EE: Is there room for both	45
7.8		Rails and a typical J2EE Web stack	45
7.9		Comparison of Rails and J2EE stacks	45
7.10		PHP vs Java vs Ruby:	46
8.0	**Other Frameworks Computability**		47
	Ajax		
8.1		Ajax on Rails	47
8.2		Traditional Web App vs. an Ajax App	47
8.3		How Rails Implements Ajax	47
	Flex Builder 2		
8.4		Flex Builder 2	51
8.5		How Flex Implements on RoR	51
9.0	**Performance and Scalability**		61
10.0	**Limitations of our Work:**		62
11.0	**Future Development Areas**		62
12.0	**References:**		63
13.0	**Contact:**		64

1.0 - Introduction:

This paper contain the report of a learning base thesis on the Programming Language "RubyOnRails". Where main objective of the report is, discover the specifics and secret facts of Rubyonrails, its compatibility over other Frameworks and the comparison with other programming language.

In thesis I mainly concentrate on how the language works and its usability to the users. I also tried to discover the hidden facts of the language and its computability over other framework and usability in different environments. I also find out a clear idea about RubyOnRails in compare and construct with other language. With that it is also supportive to present why RubyOnRail work well than the other traditional languages.

In the thesis work I develop many different small solutions with different frame work and environment to show how RubyOnRails works and its user friendly environment on different system. The thesis works not only indicate the future development of solution with ruby on rails but also indicate some lankness of RubyOnRails which should be developed soon. It also cover the details of different Framework and supportive system what we need to do work with RubyOnRails.

2.0 - Essentials and Basic Introduction:

Ruby:

2.1 - What is Ruby?

Ruby is an exciting new, pure, object oriented programming language. While few people in the West have heard of Ruby yet, it has taken off like wildfire in Japan, already overtaking the Python language in popularity. Ruby is the interpreted scripting language for quick and easy object-oriented programming. It has many features to process text files and to do system management tasks (as in Perl). It is simple, straight-forward, extensible, and portable.

2.2 - History of Ruby:

Yukihiro Matsumoto of Japan is the founder of Ruby. Ruby was created in 1993. However, it was only after 1995 that it became popular. The updated news for Ruby lovers is that Ruby has become more popular than Python in Japan. Until recently, Ruby's adoption outside Japan was hampered by the lack of documentation in English. There are not many books available on Ruby. This makes it extremely difficult for Ruby to penetrate the other parts of the world. However, the year 2002 may just be the beginning of the rise in the popularity of Ruby because a lot of books in English will be released in 2002.

NOTE: To find the name Yukihiro Matsumoto on the Ruby mailing list at www.ruby-lang.org. Matsumoto is also known as Matz in the Ruby community.

Rails framework:

2.3 - What is Rails:

A framework is a program, set of programs, and/or code library that writes most of your application for you. When you use a framework, your job is to write the parts of the application that make it do the specific things you want The term framework comes from the field of building construction, where it refers to a partially built house or building. Once a house reaches the framework stage, much of the work of building is done—but the house looks exactly like any other house in the same style at the same stage. It's only after the framework is in place that the builders and designers start to do things that make the house distinct from other houses.

Unlike scaffolding, which gets removed once the house is built, the framework is part of the house. That's true in the case of Ruby on Rails, too. When you run your application, the Rails framework—the code installed in the various Rails directories on your computer—is part of it. You didn't write that code, but it's still part of your application; it still gets executed when your application runs.

A computer application framework like Rails and a house framework are different in one important respect: The computer framework is reusable. Install Rails once, and it serves as the framework for any number of applications. What it provides, it keeps providing; you never have to write the parts of your application that are pre-written as part of Rails. The difference between what you can do with Rails and what you would have to do if you wrote the equivalent of a Rails application from scratch is considerable. If you're developing a shopping cart site with Rails, you have to decide things like whether shipping charges will be shown before checkout, or whether to slap up links to products similar to those in the customer's cart.

But you don't have to design a translator that automatically maps database table names to Ruby method names, or write a comprehensive library of helper routines that automate the generation of HTML form elements, or engineer a system that layers automatic method calls in a particular order based on a simple list. These tasks (and many more) have been programmed already, and they're available to every Rails application. The Rails framework exists to be used, and it's designed for use. The best way to understand both the "what" and the "why" of its design, and its relation to the language in which it's written, is to first grasp what you're supposed to do when you use it.

RubyOnRails (RoR):

2.4 - What is Ruby on Rails?

Tools to make a web designer's life a bit easier - First there was Ruby, and now there is Ruby on Rails.

Ruby is an open-source web-programming language that was developed in Japan in 1995. Ruby resides on your web server and operates similarly to other programming languages, such as PHP, ASP and Perl.

"Rails" is a separate pre-built framework to help automate common Ruby commands. It was developed in 2004 by a Danish programmer, David Heinemeirer Hansen. It, too, is open-source and it makes programming in Ruby much easier and quicker.
Again nowadays, Ruby is not only a web programming language. Ruby is an all-purpose, object-oriented scripting language. It has many included libraries, including CGI and other web technology implementations.

A rail is not designed to automate Ruby commands. A rail is a full-stack, Model-View-Controller (MVC) web framework, designed using the Ruby programming language. Because of Ruby's dynamic nature, Rails provides many exciting features for web programmers, such as easy database access and methods to utilize some of the newest technologies on the web.

For example, to write programming code to create, update and delete files from a database can take much time and effort. Rails, however, has a pre-built framework for quickly programming this common task in Ruby, and by using it a programmer can accomplish in minutes what would sometimes take hours to do. As such, Ruby on Rails is gaining in popularity as a simple, easy and free programming option to create websites.

3.0 - Basic Structures And Architectures

3.1 - Software Architecture of Ruby on Rails:

Many web-based software systems contain a number of common types of components and functionality. Calling application code based on HTTP posts, performing database transactions, rendering HTML responses, and putting all of the system's functionality into a scaleable, extensible architecture are all common concerns in these applications. Because of the prevalence of web applications in today's world, these problems are faced by many developers worldwide on a daily basis. Clearly, when facing such a task, a prudent software engineer would seek a common framework for web application development that could reduce the need to reinvent existing solutions, create a common base architecture for a number of different applications, and provide a ready-to-go solution for as many of the requirements as possible.

This solution to the problem is not a new one. Frameworks like Microsoft's .NET or Sun's Java 2 Enterprise Edition (J2EE) are widely used to satisfy the needs of developers creating large scale service oriented architectures for web deployment. These systems have been tried and tested many times and are well known for the robust solutions they can provide. Unfortunately, with such a robust architecture seems to come with an inherently large amount of overhead in terms of configuration and setup. In addition, there tends to be a large learning curve to understand the architecture and all of its variants. In particular, J2EE has many different technologies that can be glued together within it, including Struts, Hibernate, and spring.

Enter Ruby on Rails. Ruby on Rails is an open-source web application framework designed to make the job of the developer easier by providing everything one needs to create web-based, database-driven applications easily while focusing on the idea, not the technology behind implementing the idea. It is designed to be a marriage of the "quick-n-dirty" camp of PHP web developers and the "slow-n-clean" methodologies of the old enterprise architecture crowd. Many proponents of the framework will be quick to tout quotes saying that it can make developing web systems ten times faster than performing similar tasks with competing, more complex frameworks like J2EE.

Rails was developed while its creators were building a real life system with it. This allowed them to test out their ideas as they were putting them in to production, along with allowing them to drive the development of the framework based on the actual needs of a real project.

The creator of Ruby on Rails, David Heinemeier Hansson of 37signals, will tell you that some of the framework's power comes from the fact that it follows the mantra of "convention over configuration." This means that instead of allowing an ultimate amount of flexibility in what a developer can do with a system, there are limits imposed. While this may sound like a bad thing, it actually allows for a much easier time in creating the

types of solutions that it is specifically designed for. In David's words, "If you are happy to work along the golden path that I've embedded in Rails, you gain an immense reward in terms of productivity that allows you to do more, sooner, and better at the application level." Basically, because assumptions are made on the part of the framework about how the developer will be doing things, the developer doesn't have to tell the framework how those things will be done. This means less time spent writing configuration files and more time working on the idea of the product.

A high-level view of the Rails architecture makes it look very much like other web application frameworks. Its architecture is based on MVC, designed to separate concerns and reduce the coupling between disparate functionality. Part of Rails' power comes from the fact that this architecture is connected through an auto-generated code base structure. The developer can set up a project and then make changes to the model directly and observe the results without a lot of work to set up every part of getting that data propagated throughout the rest of the system.

The entire benefit of using Ruby on Rails is in many of the small things it does to benefit the developer in ways that other similar frameworks do not. These little things collectively work together to create a development experience which overall seems much more agile and less cumbersome than the process involved in working with other large and robust web development architectures.

3.2 - A framework user's–eye view of application development:

When you set out to write a Rails application—leaving aside configuration and other housekeeping chores—you have to perform three primary tasks:

1. Describe and model your application's domain. The domain is the universe of your application. The domain may be music store, university, dating service, address book, or hardware inventory. Whatever it is, you have to figure out what's in it—what entities exist in this universe—and how the items in it relate to each other. The domain description you come up with will guide the design of your database (which you'll need to create and initialize using the administrative tools provided by the database system) as well as some of the particulars of the Rails application.

2. Specify what can happen in this domain. The domain model is static; it's just things. Now you have to get dynamic. Addresses can be added to an address book. Musical scores can be purchased from music stores. Users can log in to a dating service. Students can register for classes at a university. You need to identify all the possible scenarios or actions that the elements of your domain can participate in.

3. Choose and design the publicly available views of the domain. At this point, you can start thinking in Web-browser terms. Once you've decided that your domain has students, and that they can register for classes, you can envision a welcome page, a registration page, and a confirmation page. Customers shopping for shoes may have access to a style

selector, a shopping cart, and a checkout page. Each of these pages, or views, shows the user how things stand at a certain point along the way in one of your domain's scenarios. You have to decide which views will exist.

3.3 - MVC framework concept:

MVC is the family of frameworks to which Rails belongs, and getting to know about the family traits will help you understand Rails.
The MVC principle divides the work of an application into three separate but closely cooperative subsystems. Although the correct term is MVC, for the sake of matching the framework with the three tasks listed in section 2.1.1, we'll flip it temporarily to MCV (arguably a more sensible order anyway). Model, controller, and view, in the general case of any framework of this type, can be described as follows:

■ Model—The parts of the application that define the entities that play a role in the universe of the application (books, hammers, shopping carts, students, and so on)

■ Controller—The facility within the application that directs traffic, on the one hand querying the models for specific data, and on the other hand organizing that data (searching, sorting, massaging it) into a form that fits the needs of a given view

■ View—A presentation of data in a particular format, triggered by a controller's decision to present the data.

Three things happen in an MVC application: You get information; you store and manipulate that information; and you present that information. On its own, that's not remarkable; most computer programs perform operations on data and give you the results. The MVC principle, however, isn't just a description of what happens to the data. It's also the governing principle behind how you, the developer, work on a program.

When you're writing program code to handle one of these areas or layers of your application (the models, the controller actions, the views), you are only writing code for that layer. If you wake up one day and decide to write all the entity modeling code for an address-book application, all you have to do is make decisions about how you think the address-book universe should be broken down into entities. You don't have to worry about how many fields you'll have to fill in on the screen to add a new entry, or whether to use a Confirm button when you delete someone, or anything else practical or visual. All you have to do is model the domain of the address book. After you've done that, you can start thinking about what you want to be able to do, and what kinds of data presentations you want access to (one person at a time, everyone who lives in a particular state, all the G's or B's or T's grouped together, and so on).

This clear-headed division of labor—your labor, as well as the application's makes the MVC approach attractive. You'll get a lot of mileage out of sticking to this three-part worldview when it comes to Rails. Whether you're getting a handle on Rails' theoretical

underpinnings, bearing down on the details of writing a real life Rails application (we'll do both in this chapter), or navigating the directory structure of your application, you'll find that you're always in this three-part structure: a universe populated with entities that are manipulated and controlled through actions that culminate in publicly available views.

3.4 - Analyzing Rails' implementation of MVC:

The MVC concept is all about dividing the work of programming and the functioning of a program into three layers: model, view, and controller. In accordance with its MVC foundations, Rails is made up largely of three separate programming libraries—separate in the sense that each has its own name and you can, if you need to, use them separately from each other.

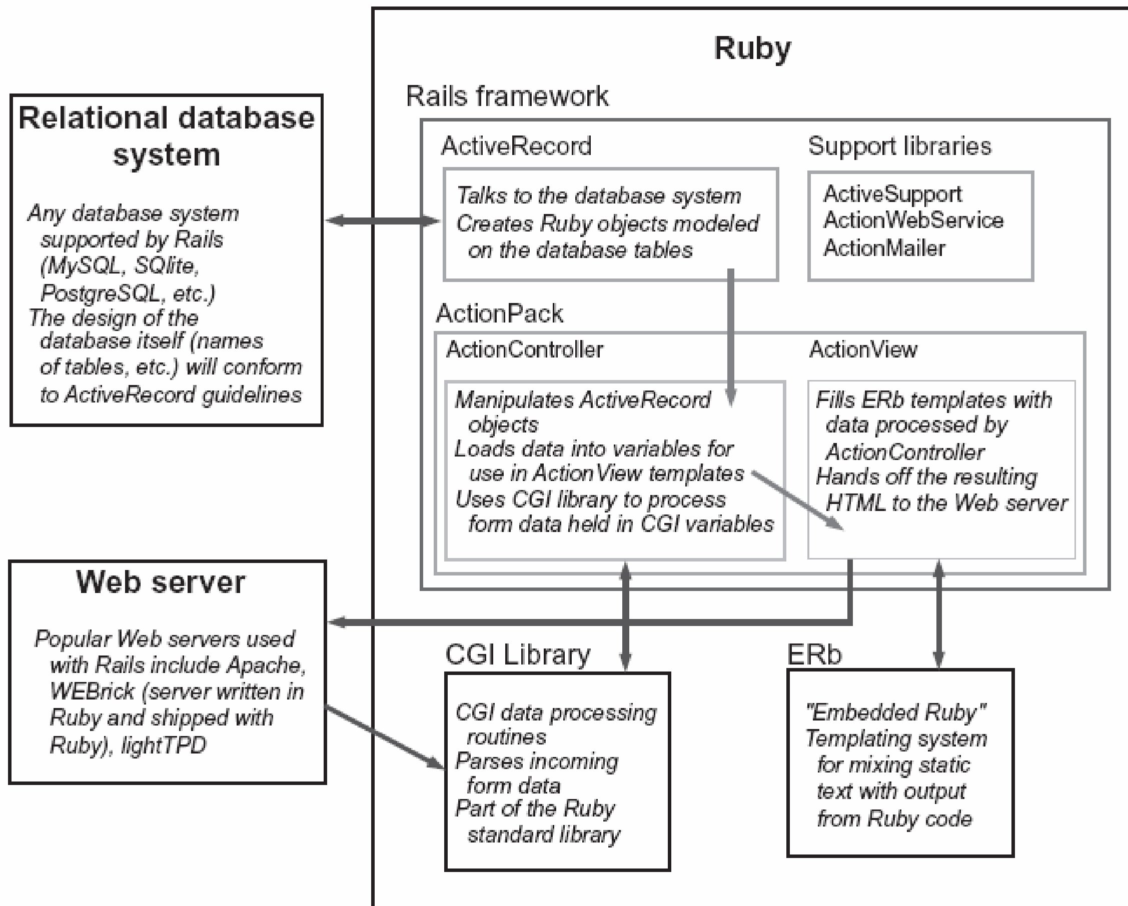
There are three libraries forming the bulk of the Rails framework. You can see these three libraries installed on your computer. They usually reside in the gems area of a Ruby installation.
Assuming a standard, default installation, you can
Find them like this:

```
$ cd /usr/local/lib/ruby/gems/1.8/gems
$ ls
```

3.5 - Overview of how Rails implements the MVC framework design:

MVC phase	Rails Sub Library	Purpose
Model	*Active Record*	Provides an interface and binding between the tables in a relational database and the Ruby program code that manipulates database records. Ruby method names are automatically generated from the field names of data-base tables, and so on.
View	*Action View*	An Embedded Ruby (ERb) based system for defining pre-sentation templates for data presentation. Every Web connection to a Rails application results in the displaying of a view.
Controller	*Action Controller*	A data broker sitting between Active Record (the database interface) and Action View (the presentation engine). Action Controller provides facilities for manipulating and organizing data from the database and/or from Web form input, which it then hands off to Action View for template insertion and display.

3.6 - Schematic view of Ruby and the Rails framework:

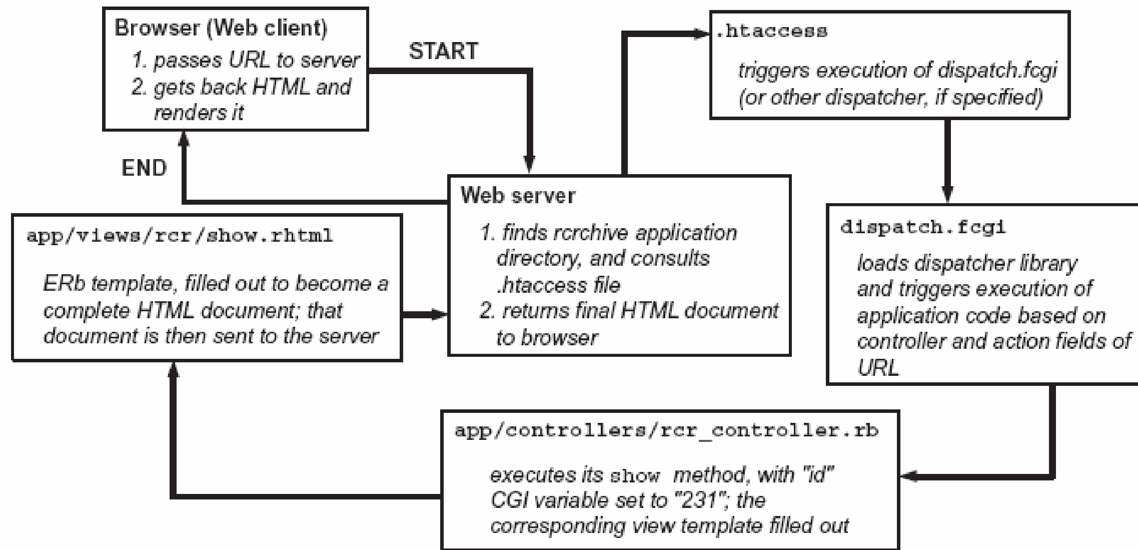


Source: www.rubyonrails.com

3.7 - Tracing the lifecycle of a Rails run

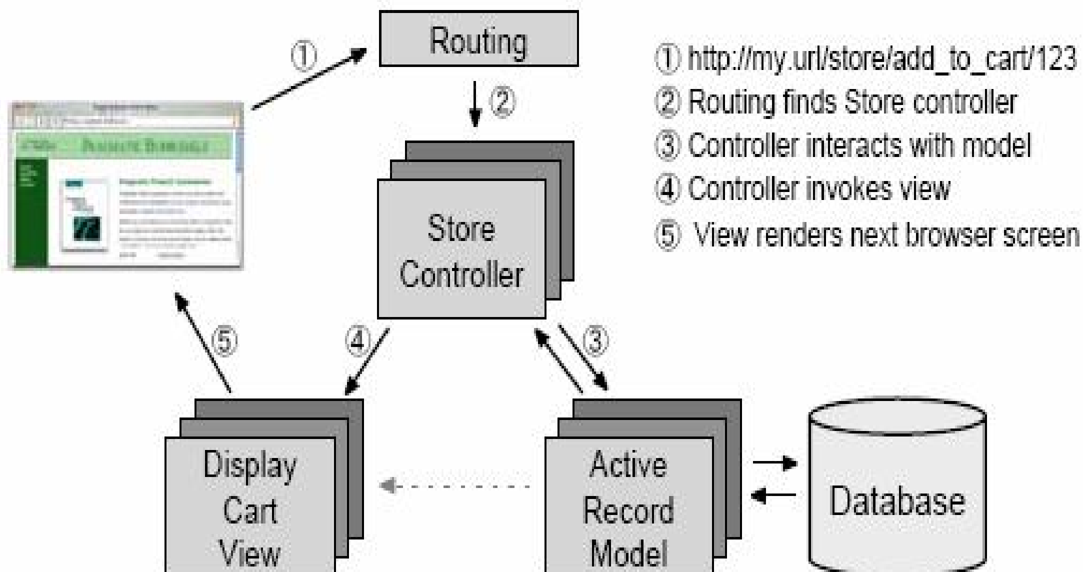
To round off this annotated tour of how Rails works, we'll look in detail at what happens when a request comes in from a Web client to a Rails application. The Web server and several auxiliary scripts and programs automatically made available to the Rails application. Although we're using WEBrick for the working example, we'll examine the basics of what's involved with setting up Apache to serve a Rails application. This process is more complicated which is why you aren't doing it in the working example, and why it contains useful lessons about how the whole request-handling process operates. The process of listening to and responding to a request coming in to a Rails application can be broken into several stages: the Web server talking to the dispatcher (which is a Ruby program); the dispatcher awakening the appropriate controller and asking that controller (which is also a Ruby program) to perform a particular action; the performance of that action; and the filling out of the view, based on the calculations and data manipulations carried out in the controller action. We'll look at each of these stages in turn.

Rails sequence for http://www.rcrchive.net/rcr/show/231 (controller "rcr", action "show", id"231")



Source: **www.rubyinside.com**

Flow of steps involved in Rails' typical handling of an incoming request

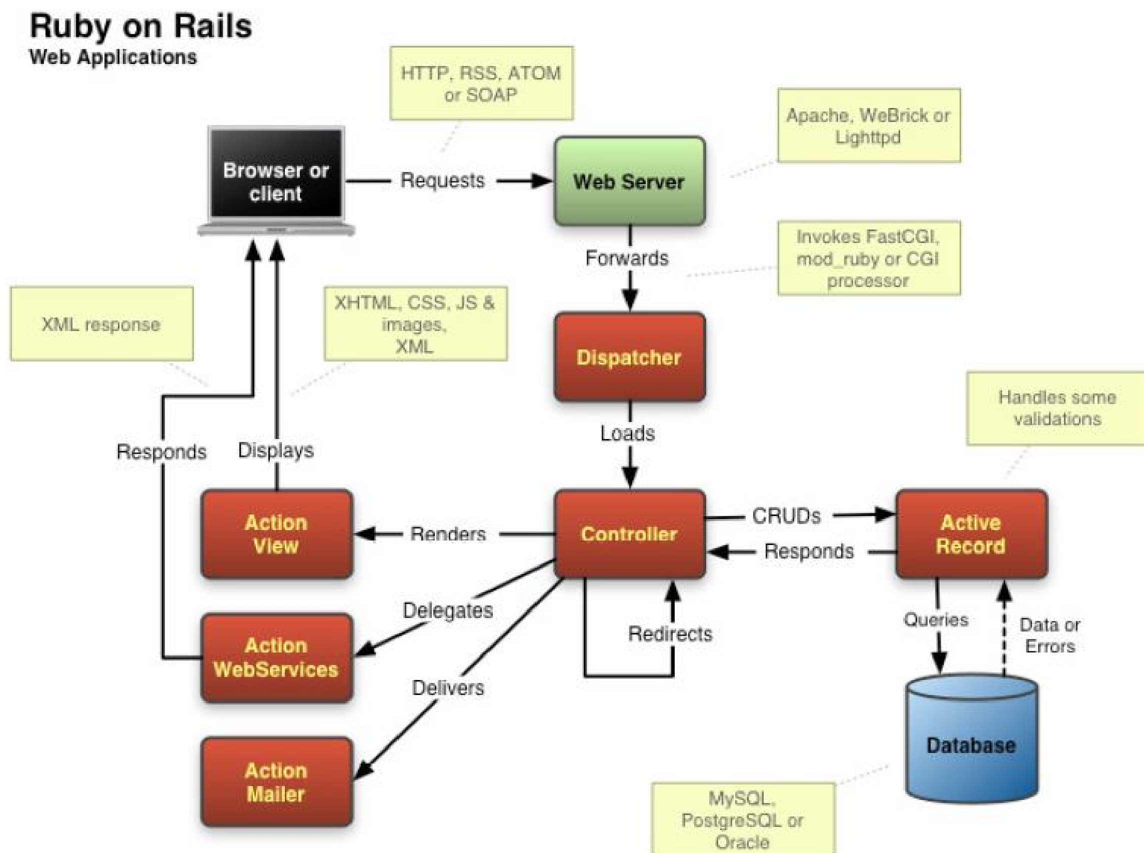


Source: Rubyonrails.com

4.0 - Component/Services Of RoR

4.1 - Component/Service Details:

Rails Architectural Approach The key to understanding the Rails framework is realizing how it utilizes the Model-View-Controller pattern. What sets this stack apart from others is the way all three facets of the pattern are integrated into Rails out of the box. The goal of the framework is to allow developers to get up and running quickly by providing "what most of the people need most of the time." This attitude can be found throughout the Rails architecture as conventions trade flexibility in the infrastructure for flexibility in the application. The process view in Figure 2 shows how Rails supports the Model-View-Controller pattern in both organization and design. Controllers are the cornerstones of a Rails application. They are responsible for interacting with Active Record, redirecting to other controller actions and rendering out different views back to the client.



Process view of Rails workflow

Source: **www.econsultant.com/web-developer/ruby-rails-tutorials/**

4.2 - The functional requirements of the system:

- The web server is responsible for accepting incoming requests from clients and determining how to process it.

- The front controller accepts HTTP requests, parses the URL, and forwards processing of the request to an appropriate action. This process is referred to as routing, determining what should be done with the request through a mapping of request to a particular controller or action.

- The page controller is the logical center of the application. It coordinates the interaction between the user, the views, and the model. This includes:
  - Routing external requests to internal actions
  - Managing caching
  - Managing sessions

- The template view is responsible for determining how the information from the controller is presented to the end-user via the browser. It generates dynamic content that is usually based on a backend store.

- The model is responsible for maintaining the state of the application. Models represent the domain object of a software system, containing the data and behavior unique to that object.

- The mailer provides the ability to send and receive e-mail.

- The web service provider is responsible for exposing web service APIs and handling SOAP and XML-RPC protocol requests.

- A plug-in allows developers to extend or modify the core framework.

4.3 - Rails Application Structure

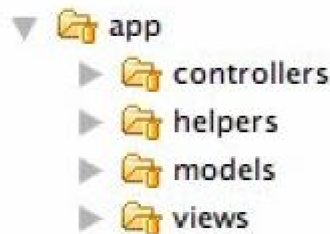
Rails applications have a defined structure and organization. The framework often favors convention over configuration to achieve the end of solving most of what people need most of the time. When a new Rails project is created a dozen folders appear in the application directory.

Folder	Description
app	Holds application specific code. This includes models, views, Controllers, apis and helpers.
components	Components are self-contained packages of controllers and views. These are normally utilized when there is a need to reuse a view that has associated reusable behavior.
config	Configuration files for Rails environment, URL routes and database.
db	Database schema and migrations.
doc	Holds application documentation generated from RDoc.
lib	Application specific libraries.
log	Development and production server logs.
public	This folder is made available to the web server. It contains the dispatch scripts along with static content such as images, java scripts, style sheets and default html files.

script	Helper scripts for development server, automation, generation and plug in management.
test	Unit, functional and integration tests along with test fixtures.
tmp	Temporary directory for development server.
vendor	Contains plugins and external libraries the application depends on.

4.4 - Directory Structure Description

The app folder is where the Rails application code will live. Figure 5 shows the explicit structure of the app folder. It is clear from the organization where the elements of MVC should go. The helpers folder contains associated helper classes for controllers and views. Helpers are used to define reusable methods that can be shared among actions within a controller or the entire application itself.



“app” folder structure

Models Rails interacts with models through the Active Record component. Active Record models are based on the Active Record pattern defined in Martin Fowler’s “Patterns of Enterprise Application Architecture.” Active Record defines a model structure that encapsulates a row in the database and domain logic on that data. This allows for logic such as validation, associations and other custom behavior to exist in the model class itself. The Active Record component requires only the database connection information to be configured in order to work. There is no need for explicit object to database mapping. Active Record elicits field information directly from the database itself at startup time. It is possible to have a fully functional model that just extends from the ActiveRecord::Base class.

```
class Post < ActiveRecord::Base
end
```

ActiveRecord seeks to simplify the object relational mapping process by following a few conventions. Table names in the database are pluralized versions of the objects they represent. For example, a “posts” table would represent a collection of “Post” model objects. Associations must be defined in the models themselves and those relationships follow specified naming schemes. Most of the conventions imposed by ActiveRecord can be easily overridden but they are there to help solve the issues most people have when setting up an ORM layer. The ActiveRecord component is database agnostic. It is an abstract layer that connects to a database through an adapter interface. Currently, ActiveRecord supports the following 9 databases: DB2, Firebird, MySQL, OpenBase, Oracle, PostgreSQL, SQLServer, SQLite and Sybase. This abstraction also provides the ability to define an agnostic ActiveRecord schema written in Ruby. Rails offers something that builds upon this idea of agnostic structure called Migrations. They offer a database independent way to create and step up/down versions of the schema. In addition to ActiveRecord being the gateway to application models it is also capable of defining and managing database structure in an abstract way.

Views Views are the rendered response to a client request. In Rails, there are several types of views that address different client needs:

View	Description
RHTML	This is the default view for Rails applications. RHTML renders HTML based content back to a client web browser. RHTML also contains Ruby expressions that allow for programming within the view. This includes things like loops and calls to helper methods much in the same way other frameworks embeds expressions.
RJS	RJS templates are Ruby based javascript views. This type of view is used for asynchronous javascript calls (AJAX) made to a Rails app. RJS templates allow the view to execute javascript effects and manipulate the page where the AJAX call originated.
RXML	RXML is Ruby generated XML code. It uses builder templates to easier construct an XML document to return to the client. Rails automatically returns RXML views as the XML content type.

All of the views in Rails are routed through a controller. A controller action is responsible for passing information to a corresponding view. Any variables created in the instance of the Controller action will be made available to the view as well as application and controller specific helper functions. The code below shows how these variables are used

in a view that loops over a collection of Post model objects and renders out some attributes. In this case the @posts variable would have been defined in the corresponding controller action.

```
<h1>Posts</h1>
<% for post in @posts %>
<h2><%=post.title%></h2>
<p><%=post.body%></p>
<% end %>
```

Rails also has support for reusable and template views. Partials are snippets of views that can be reused throughout an application. Layouts are template views that can wrap other views and be utilized to theme or brand a web application. Layouts can be applied on an application, controller or action basis.

Controllers Controllers are the directors of a Rails application. They take a dispatched request and decide what to do with it. Controllers are responsible for interacting with the model, choosing which view to render and any redirects that take place. Rails controllers are a hybrid of the Front and Page Controller patterns. Instead of each potential action being placed in its own class, common actions are grouped together into one controller class. The action is no longer a standalone class but simply a method within a controller. The snippet of code below defines a Rails controller that has add, edit and delete actions.

```
class PostsController < ActionController::Base
def add
end
def edit
end
def delete
end
end
```

The default routing format to invoke one of these actions is: controller/action/id. The id parameter is part of the default mapping but it is not required. For instance, the default route for the add action above would be “/posts/add”. The route to edit a Post with an id of 2 would be “/posts/edit/2”. This routing scheme in Rails is highly configurable beyond the default mapping and can produce very complex URL structures. Once a controller action has been executed a view must be rendered back to the client. By default Rails will render the corresponding view file with the same name as the action. This file can end in rhtml, rxml or rjs extensions and will be located in the views folder. The example controller and view below will simulate the lifecycle of a request for showing a Post.

```
class PostsController < ActionController::Base
def show
@post = Post.find(params[:id ])
end
```

end

The request will be routed to the show action. It will then interact with ActiveRecord and store the Post model it found for the specified id in the @post instance variable. After the action has executed, the view is discovered in app/views/posts folder. The action is named “show” so in this case Rails will look for a show file ending in either rhtml, rjs or rxml. In this case the goal is to render the show view of the Post back to the browser so the show.rhtml will be used.

The controller then passes the @post variable to the view.

```
<h1><%=@post.title %></h1>
<p><%=@post.body %></p>
posted by <%=@post.author%>
```

The Ruby expressions are then evaluated, the post information is inserted and the view is rendered back to the client.

Tests One of the issues that web frameworks are commonly stricken by is setting up environments for their components to be tested in. The Rails framework is inherently testable because its strict MVC structure and low coupling allows components to be tested independently. As a result, testing facilities were built in to Rails without much trouble. Unit, functional and integration tests allow models and controllers to be tested separately or together. Plugins can be unit tested as well. Mocks, fixtures and helpers further enhance the test environment in Rails.

Unit tests are primarily for models, and are also used to test mailers and plugins. Rails unit tests extend the Test::Unit::TestCase class, which is part of the Ruby Test::Unit library. Test::Unit is a general unit testing framework for Ruby, similar to JUnit for Java. Each model has a corresponding test case, and each test case is comprised of several test methods. The test methods will exercise some feature of the model and assert the success of its functionality.

Functional tests are primarily for controllers, and are also used to test web services. Rails functional tests also extend Test::Unit::TestCase, in a similar fashion as unit tests. The structure of the test cases is similar as well, with a test case corresponding to each controller containing several test methods. In the setup method that is invoked before each test method, Rails creates an instance of the controller under test and mock request and response objects. With these, a browser request can be simulated. The request object is configured in the test method, an action is invoked on the controller, and the response object is examined to determine the success of the operation.

Integration tests are used to simulate the execution of the entire application. The test case structure appears similar to functional tests, but the execution context of the test is different. In functional tests, the mock request and response objects provide the entire

context and the actions are executed directly. In integration tests, however, the application's routing logic is invoked. A mock request is sent in from the top, the URL is parsed, routing is handled, and then the action is invoked.

Common across all tests are the concepts of mocks, fixtures and helpers. Mocks are stub objects that can be substituted for real system components, in order to isolate functionality. For example, in a functional test, a mock model could be used so that no database connection will occur and only the controller logic is tested. Fixtures are preconfigured data sets for use in test cases. Fixture data is defined in a series of YML files and is loaded into the test database when test cases are executed. If the development database was used for testing, data could change over the course of several test cases and cause others to improperly fail. However, with a predefined set of fixture data, the test database can be cleared and reloaded before each test run to ensure a consistent environment. Finally, helpers allow common functionality to be reused across test cases. Often-invoked method sequences like user authentication can be extracted to a helper and reused cleanly.

Extending Rails Rails plugins are used to modify and extend the core framework. They can be found in the vendor/plugins folder of a Rails application. Plugins provide a way to define reusable code that can be shared among many applications. The idea behind this extension architecture is for it to be a place to develop or use functionality that works for some of the people some of the time. This allows Rails to maintain its focus on what most people need but still have an outlet for niche implementation.

4.5 - Rails Statistics:

Generated Jan 24, 2005:

Name	Lines	LOC	Classes	Methods	M/C	LOC/M
ActionMailer	7711	5458	60	545	9	8
ActionPack	7431	4214	106	604	5	4
ActiveRecord	9830	6252	131	799	6	5
ActiveSuport	1191	790	17	104	6	5
Railties	1349	879	15	77	5	9
Total	27512	17593	329	2129	6	6

Note: The above is not fully accurate. It excludes tests, but includes bundled libraries.

5.0 - Features and Qualities:

5.1 - Features of Ruby:

Ruby is an open-source, general-purpose, interpreted, and powerful server-side scripting language. Ruby also is freely available on the Web, but it is subject to a license. You can obtain the license from the URL www.ruby-lang.org/en/license.txt. With Ruby, programming becomes very easy. One of the major advantages is that Ruby provides a simple interpreter, unlike other programming languages. Using other programming language compilers, a programmer spends maximum effort in trying to get over the complexities of the compiler instead of concentrating on the actual coding part. This becomes frustrating at times for the programmer. With Ruby, you need not worry about all these issues; you can concentrate specifically on coding. The advantage is Ruby is an untyped language. With untyped languages, you need not bother to define everything before you start executing the code. However, one thing you need to remember is that Ruby is an interpreted language. Interpreted languages are slower when compared to compiled languages. Let us discuss some of the salient features of Ruby.

Easy:
Ruby has a clean and easy syntax that allows a new developer to learn Ruby very quickly and easily. It will require a lesser effort for people who have some programming knowledge. The syntax of Ruby is similar to that of many programming languages such as C++ and Perl. Therefore, it becomes very easy for programmers to learn Ruby and start writing programs. Ruby is a perfect object-oriented language, unlike some languages, such as Python, which only supports the concept of object orientation. In fact, Ruby is a simplification of these languages, and it does not require any extra effort to learn an unfamiliar concept, syntax, or keyword. With Ruby, you will be surprised at the amount of code you can churn out in a day. The reason is that after you learn the basic syntax, which is easy compared with other languages, you will not have many errors in your code. In addition, the Ruby interpreter is fantastic. There are no major hassles in using it. Because of all these factors, the time you spend in debugging code is minimal. Fortunately, in Ruby you do not have to go through the pain of putting semicolons at the end of each and every statement. All these features make Ruby a very good and simple language to work with.

Scalable:
UNIX shell scripting languages are fairly easy and can handle simple tasks very easily and efficiently. However, when you add more features to a script, the script becomes very large, complicated, and slow. You are unable to reuse your code, and even small projects require huge scripts. Ruby provides a better structure for large programs. You can write modules in Ruby and then reuse those modules across different code. Ruby also provides any built-in modules to help you in system management tasks, networking, socket programming, and graphic user interface (GUI) programming.

Object-Oriented:
As stated earlier, Ruby is a true object-oriented programming language. All the features of Ruby are implemented as objects. Ruby shows all the characteristics of an object-oriented language. Ruby shows multiple inheritances indirectly like Java. Java implements multiple inheritances by using interfaces, whereas Ruby implements the same by using mixins. Extensible There are many libraries that provide functionality that may be useful to have in your programs. It does not make sense to rewrite all these libraries in Ruby. In addition, there are occasions when you may need performance that is better than the performance of an interpreted language such as Ruby. It will be useful, therefore, if you could write the time-critical code in an efficient language and simply call it from your Ruby code. Ruby helps
you to take care of both these situations. Using Ruby, you can easily write extensions in C/C++ that hook seamlessly into Ruby's environment. It would seem that they are simply other pieces of Ruby code. You can create Ruby extensions in C/C++ using dynamic or static binding. You can even go the other way around and embed a complete Ruby interpreter into a C program, allowing you to use its scripting facilities rather than having to write your own purpose-built engine. The fact is that with Ruby extensions the sky is the limit!

Rich Core Library:
Many development modules are built into Ruby. A programmer can make use of these modules directly. In addition to modules that work on all platforms, the library has modules that are specific to a particular platform or environment.
Ruby built-in modules perform all types of usual tasks, such as HTTP, FTP, POP, SMTP, and many other services. Using the rich core library, you can write applications for downloading a Web page, connecting to a database, developing a GUI, and so on. Web Scripting Support and Data Handling Ruby can be used for developing Internet and intranet applications. You can even write a Web server using Ruby. You can write Common Gateway Interface (CGI) scripts using Ruby. You can even embed Ruby programs into Hypertext Markup Language (HTML). You also can write high-end Extensible Markup Language (XML) applications using Ruby.

Object Distribution:
Using distributed Ruby, you can create a server object and expose that object. Next, you can write a client program and access the server object. This becomes similar to Java's Remote Method Invocation (RMI). All this can be done very easily in Ruby.
Databases you can use the built-in objects from the various libraries that are available to make Ruby talk to a database. Using Ruby, you can easily connect to DB2, MySQL, Oracle, and Sybase. ODBC drivers are being written to connect Ruby to popular databases.

GUI Programming:
There are several GUIs, such as Tcl/Tk, GTK, and OpenGL. You can download the extensions for these GUIs from the Ruby Application Archive (RAA). The Ruby Application Archive is a Web page that acts as a repository for many Ruby applications.

Exception Handling you execute a program, and suddenly, an unknown error pops up. The program will end abruptly without knowing what to do and exit. Exceptional cases such as these are termed exceptions. To handle such exceptions, you need to add exception-handling code in your main code. With Ruby, exception handling becomes clean and simple. Using exception handling in Ruby, the programmer needs to make less effort to debug an error.

Portable:
Ruby can be installed in Windows and POSIX environments. Code written under Windows can be run under Linux and vice versa unless you are not trying to access features specific to that operating system. Therefore, when you say a code is portable, it means lesser expenditure and wider distribution.

Freeware:
Ruby is a freeware and can be redistributed freely in the source form subject to the license we discussed earlier. Programmers and users are allowed to use Ruby's source code in any desired way. You can download the Ruby source code, modify the code, and even distribute the code. Ruby is also free for commercial use. You can make applications in Ruby and upload it to the RAA for other users to access. In the same way, you can download many of the applications created by different users from this archive page. cases such as these are termed exceptions. To handle such exceptions, you need to add exception-handling code in your main code. With Ruby, exception handling becomes clean and simple. Using exception handling in Ruby, the programmer needs to make less effort to debug an error.

5.2 - Fundamentals of Ruby

- Ruby has simple syntax, partially inspired by Eiffel and Ada.

- Ruby has exception handling features, like Java or Python, to make it easy to handle errors.

- Ruby's operators are syntax sugar for the methods. You can redefine them easily.

- Ruby is a complete, full, pure object oriented language: OOL. This means all data in Ruby is an object, in the sense of Smalltalk: no exceptions. Example: In Ruby, the number 1 is an instance of class Fixnum.

- Ruby's OO is carefully designed to be both complete and open for improvements. Example: Ruby has the ability to add methods to a class, or even to an instance during runtime. So, if needed, an instance of one class *can* behave differently from other instances of the same class.

- Ruby features single inheritance only, *on purpose*. But Ruby knows the concept of modules (called Categories in Objective-C). Modules are collections of

methods. Every class can import a module and so gets all its methods for free. Some of us think that this is a much clearer way than multiple inheritance, which is complex, and not used very often compared with single inheritance (don't count C++ here, as it has often no other choice due to strong type checking!).

- Ruby features true closures. Not just unnamed function, but with present variable bindings.

- Ruby features blocks in its syntax (code surrounded by '{' ... '}' or 'do' ... 'end'). These blocks can be passed to methods, or converted into closures.

- Ruby features a true mark-and-sweep garbage collector. It works with all Ruby objects. You don't have to care about maintaining reference counts in extension libraries.

- Writing C extensions in Ruby is easier than in Perl or Python, due partly to the garbage collector, and partly to the fine extension API. SWIG interface is also available.

- Integers in Ruby can (and should) be used without counting their internal representation. There *are* small integers (instances of class Fixnum) and large integers (Bignum), but you need not worry over which one is used currently. If a value is small enough, an integer is a Fixnum, otherwise it is a Bignum. Conversion occurs automatically.
- Ruby needs no variable declarations. It uses simple naming conventions to denote the scope of variables. Examples: simple 'var' = local variable, '@var' = instance variable, '$var' = global variable. So it is also not necessary to use a tiresome 'self.' prepended to every instance member.

- Ruby can load extension libraries dynamically if an OS allows.
- Ruby features OS independent threading. Thus, for all platforms on which Ruby runs, you also have multithreading, regardless of if the OS supports it or not, even on MS-DOS! ;-)

- Ruby is highly portable: it is developed mostly on Linux, but works on many types of UNIX, DOS, Windows 95/98/Me/NT/2000/XP, MacOS, BeOS, OS/2, etc.

5.3 - Key Quality Attributes of Rails

Quality Attribute	**Requirement**
Simplicity	Allow developers to focus more on the domain-specific functionality of a software application by catering default configurations to the majority (80%) of the population, rather than trying to satisfy the entire development community. Create a

	framework that is easy to understand and quick to learn. This is considered the most important attribute since the system was designed to make it easier to write web applications. In order to do this, Rails needed to remove the need for large and complex XML configurations. There are many techniques that developers commonly use throughout their configurations. The goal is to take those ideas and make them the default, so that configurations are not needed. In addition, the structure of directories and method names allows developers to quickly learn the system. An example of this quality attribute can be seen in the object-relation mapping of models to tables. In Rails, the convention is for the model to be a singular noun and the name of the table in the database to be the plural of that noun (e.g. Job model and jobs table). As a result, developers do not have to explicitly state this relationship in a configuration file. The understandability of the system is visible through many of the method names in each library, such as through dynamic attribute-based finders in the persistence layer. For example, developers can find users in the database through a method called User.find_by_first_name_and_last_name().
Reusability	Adhere to the "Don't Repeat Yourself" (DRY) philosophy. Every piece of knowledge in the system should be expressed in just one place. Allow developers to easily reuse components in a webpage. This was considered an important attribute because of the inherent problem of maintainability when information and knowledge is specified in more than one place. It is inevitable that this information will begin to conflict at some point. DRY is important for software to remain flexible and maintainable. This is visible in the system through Rails' use of layouts, components, and partials. Each of these elements is aimed at abstracting out parts of a view such that they can be reused by multiple actions in multiple controllers.
Extensibility	Allow incorporation of new features into the framework without affecting the stable codebase, so that units of code can be fixed or updated on their own release schedule. Although not as important as simplicity and reusability, developers will often find the need to modify parts of the core architecture or add new features to existing classes in the architecture without having to subclass those components. Plugins are important in building an extensible framework by allowing plugins and the core framework to be developed independently. This can be seen through several community contributions on, such as the Calculations plugin which adds the ability to use calculation queries (from any model) to the database in the underlying persistence layer. Although the plugin adds functionality to an existing class, it does not specifically modify that class.

Testability	Application developers should be able to write unit, functional, and integration tests with minimal additional effort required to support database population and interacting with test data. Manually testing a web application can become a massive effort after more than a few features have been implemented. Test-first development is widely considered a valuable process in agile software development. As such, making testing easy was considered a relatively important design choice in the framework. Testing is baked right into every project from the start. Every time a new model or controller is generated, a model/functional test is simultaneously created that is associated with that component. In addition, Rails automatically creates test fixtures for populating the database with data prior to the execution of every test, making it easy to setup every test.
Productivity	Provide developers with tools for automatically generating the essential components required to get started on the system. The core idea of Rails (and Ruby) is to address the needs of developers, increasing their productivity with clear and consistent frameworks and tools. The goal of increasing productivity is clear throughout all of the quality attributes previously mentioned. However, this attribute is not ranked as high as others because it is more important for support of the framework rather than as part of the actual framework. These productivity tools are visible in Rails' use of generators, specifically scaffolds. Scaffolds automatically generate the model, controller, and views commonly used to manipulate an object in the database, such as: • Creating Editing Viewing Destroying Scaffolds allow developers to quickly get started with the basic elements needed for the common web application.
Modifiability	Existing components in the system should be able to easily facilitate the incorporation of changes. The system should allow components to have a high-level of cohesion and low-level of coupling such that changes can be easily made. As mentioned in the Reusability quality attribute, the DRY principle states that everything should be expressed in a single place; there should be minimal, if any, duplication of code. The creation of components that are either independent or reused throughout the code inherently increases the modifiability of the system. This quality attributed is ranked last as it is something that arises from Reusability, rather than something that was explicitly designed into the system. The use of layouts is a prime example of how the DRY principle creates a high level of modifiability. Layouts allow developers to create a template that will be used by a collection of views. Any changes to the layout are immediately visible in all views that use that layout. This design facilitates making changes less work on the developer.

5.4 - How each quality goal achieved in the system:

Goal	How Achieved	Tactics Used
Simplicity	Assume most common defaults for configurations; language constructs are succinct and expressive: “less is more”	− Abstract common services − Limit possible options − Information hiding
Reusability	Careful assignment of module responsibilities; strict separation of concerns	− Restrict communication paths − Separate the interface from the rest of the application
Extensibility	Separate extensions to core framework from actual core implementation	− Maintain existing interface
Testability	Use of fixtures to setup and tear down data; use of test harnesses to drive the actions of the test	− Specialize access routes/interfaces
Productivity	Auto-generation of models; existing templates and conventions; full stack of integrated components	− Generalize modules
Modifiability	All of the above contributed to the maintenance of a Rails application	− Semantic coherence − Use an intermediary − Component replacement

6.0 - User's Compatibility and Usability

6.1 - What makes Ruby so popular?

Ruby weaves the best features of the best programming languages into a seamless, concise whole.

Powerful:
Ruby combines the pure object-oriented power of the classic OO language Smalltalk with the expressiveness and convenience of a scripting language such as Perl. Ruby programs are compact, yet very readable and maintainable; you can get a lot done in a few lines, without being cryptic.

Simple:
The syntax and semantics are intuitive and very clean. There aren't any "special cases" you have to remember. For instance, integers, classes, and nil are all objects, just like everything else. Once you learn the basics, it's easy to guess how to do new things and guess correctly.

Transparent:
Ruby frees you from the drudgery of spoon-feeding the compiler. More than any other language we've worked with, Ruby stays out of your way, so you can concentrate on solving the problem at hand.

Available:
Ruby is open source and freely available for both development and deployment. Unlike some other new languages, Ruby does not restrict you to a single platform or vendor. You can run Ruby under Unix or Linux, Microsoft Windows, or specialized systems such as BeOS and others.

6.2 - Rails aims to take the burden off developers:

David Heinemeier Hansson, in creating Rails, developed a framework that did not set out to meet the needs of every possible enterprise application, but rather the most conventional. In particular, Rails aims to take the burden off developers through three core ideas:

1. Convention over configuration. Tailor towards the 80 percent of the population that configures applications in the same way and make that the default. Rails make it possible to create web applications without having to write a single line of XML configuration.

2. Instant feedback. Rails makes it possible to make changes to the system and immediately gain feedback. Simple changes, in response to customer collaboration, can be made with little effort on the developer's part; the changes are visible instant

3. Full-stack framework. Rather than gluing together extensions from multiple vendors, Rails provides all of the components commonly needed by web-based software systems.

6.3 - Ten Reasons Ruby Is Cool

1. Code Does What It Looks Like It Should Do

One of the guiding precepts of the development of Ruby is the Principle of Least Surprise. This means, in a nutshell, that if you read or write code in a certain way, it should have a pretty decent chance at doing what it looks like it should. It is often very easy to picture in your mind what you want a program to do; it is often very difficult to write the code to get it to do exactly that. Since developing the necessary logic and algorithms can be hard enough on its own, the programmer shouldn't have to worry about the language getting in his or her way.

Put another way, we expect the syntax and constructs of a language to do what we think they should do, without needing to look them up every time. Consider the following line of Ruby code:

```
puts "leon".reverse.capitalize
```

If you read it aloud, you can envision what it sounds like it should do. The cool thing is, it probably does exactly that. I can't be accountable for the imagination of everyone reading this article, but I would expect most folks who read the above line would think it takes the string "leon" (in reality, a string object that has the methods "reverse" and "capitalize"), reverses the order of the letters, then capitalizes the first character. Ruby code looks a lot like English when written... so even if you aren't familiar with the guts of the language, odds are that you could read a more complex program and figure out what it is doing:

```
staff_list = ["joe", "steve", "bob"]
staff_list.sort!
staff_list.each do |first_name|
  puts first_name.capitalize
end
```

If you expect to see the output as something like this, then you are on your way to programming in Ruby:

```
Bob
Joe
Steve
```

2. Variable Scope Determined By Name

Most languages that I have seen have syntax that includes "Public/Private" or "Global/Local" for declaring variable scope, as well as notation for defining constants. In Ruby, variable scope is implicitly defined by how you name your variables:
- locals start with lowercase
- globals start with $
- instance variables start with @
- class variables start with @@
- constants start with uppercase

This allows you to look at any variable in any code and immediately recognize its scope (once you memorize the above list). In practice, I have often seen variables start with a lowercase letter (intMyInteger -or- my_integer) and constants begin with a capital letter or are in all caps (DATABASE_PATH) -- so this is in line with what I expect to see. When I use global variables in VB, I often use all lowercase and just try to remember which are global and which aren't (hey, I said I wasn't a professional)... which often leads me to read through a functions "dim" statements, or use the IDE's "find" feature. I have every confidence that scoping variables by name will save me some headaches as I continue to explore Ruby.

3. Code Blocks

Code blocks, or "anonymous methods" in the Ruby parlance, are just that -- chunks of code that are used in conjunction with some sort of data set to perform operations over the entire data set. You can think of this as executing a loop and calling a function for each value in the loop (pseudo-code follows):

```
function doSomething(value) {
  print "Now processing " & value;
}

for (i=1, i<10, i++) {
  doSomething(i);
}
```

The same can be done in Ruby using a code block, without the need to define a separate function. A code block can be enclosed in either "do/end" or "{ }" braces (most examples I've seen use the "do/end" syntax, though both work the same way), and also contains the paramater you are passing in vertical pipes:

```
1.upto(10) do |i|
  puts "Now processing #{i}"
```

```
end
```

This accomplishes the same thing, using half as much code. "upto" is an iterator method, which I will be covering in more detail later. A code block is itself an object (of the class Proc; everything in Ruby is an object) that can be operated on. In this code sample, the "upto" method receives the code block between "do" and "end" as a parameter, and incorporates that block into itself during execution. The variable "i" is how the receiver (1, then 2, then 3...) is passed to the block of code for processing. The code is executed using the current value of i, until the end statement is reached. The end statement returns program control from the code block back to the "upto" method, which then passes the next value to the block (or exits the code block if the end condition is reached).

Note that there is nothing that prevents you from declaring a function (properly a "method") and passing values to it the same way that was done in the pseudo-code above. And in fact, another neat thing about methods is that the value of the last expression evaluated by the method becomes the return value... which, again, can save you some typing.

4. Intuitive Iterator Methods

The idea of an iterator method was new to me, as most of my object-oriented training comes from VB6 (which is not fully OO, as I understand). I am familiar with the notion of a method -- a sort of built-in function that comes with a class of objects -- but an iterator method took me a bit to get my brain around.

Simply put, an iterator method is a method that lets you iterate through elements of some kind of set of data. If you refer back to the example of sorting and capitalizing names in an array, you can see the "each" method in action:

```
my_array.each do |i|
  puts i
end
```

Translated into English, this becomes "take each element of my_array, in turn, and print its value to the screen". This type of functionality is a time saver, since you don't need to know anything about the array other than that it exists (whereas normally you would need to reference the lower and upper bounds, either explicitly or by attributes such as .Lbound and .Ubound).

These iterator methods are "intuitive", because often times looking at the method name gives a good idea about how it will work; and again, given the principle of least surprise, your initial expectation will often be correct:

```
my_directory.each do ...     #executes for each entry in my_directory
my_file.each_line do ...
my_hash.each_key do ...
ObjectSpace.each_object do ...     #executes for each object in the current process
```

5. Semantic Logic In Methods Is Visually Clear

Related to the first item, this practice also helps make Ruby code very clear to read. While not necessarily intuitive at first, once you get used to the following conventions, you will never forget them.

An exclamation point at the end of a method indicates that the method will modify the original object in some way. For example, there are methods called "chop" and "chop!" (available to most classes):

So this code:

```
my_string = "12345"
new_string = my_string.chop
puts "String chopped."
puts "my_string: #{my_string}"
puts "new_string: #{new_string}"
```

When run, will look like this:

```
String chopped.
my_string: 12345
new_string: 1234
```

However, this code:

```
my_string = "98765"
puts "my_string: #{my_string}"
my_string.chop!
puts "String chopped."
puts "my_string: #{my_string}"
```

Will give this as a result:

```
my_string: 98765
String chopped.
my_string: 9876
```

The ! at the end of the method name always serves as a reminder that the receiver might be modified in some way.

Similarly, a question mark at the end of a method name indicates that the object is being queried in some way. Methods ending in a question mark will generally return a True or False, which can be incorporated into your program flow control structures. Example:

```
my_number = 999474
puts my_number.to_s.include?("94")
```

... will evaluate as True.

There is also some tomfoolery going on here, in that the variable my_number is originally an object of the class Fixnum -- or in other words, a number (more on numbers and Fixnum later). By invoking the to_s method, the number is then "typecast" (not properly, since Ruby does not use strong typing... but the idea is that it treats my_number as a different type than it is currently assumed to be) as a string, so that we can use the include? method on to see if a particular substring is contained in the full string. The Fixnum class does not have an include? method, but we can achieve the same effect by using its to_s method.

When writing your own methods in ruby, it is considered good practice to adhere to these same conventions of using ! or ? if your method modifies or queries the receiver.

A final "sensible" method naming convention involves the use of the [] symbols. In many languages, a statement like my_array[3] would indicate that something is being done to the third element in an array named my_array. Many classes in Ruby have a method named "[]", which in essence allows operations to be performed on an object as if it were an array of some sort. There is a similar method, named "[]=" that allows assignment via the same type of element reference or range. For example (items are zero-indexed):

```
my_string = "This is a test."
puts "#{my_string[3..5]}"
puts "#{my_string[2]}"
my_string[10..13] = "Ruby"
puts my_string
```

Gives this as a result:

```
s i            # (elements 3, 4, and 5 in an array of characters)
105            # (ASCII decimal code for 'i', element number 2)
This is a Ruby.    # (elements 10 through 13 reassigned using '=' operator)
```

6. Numbers Are Objects

In many programming languages, even primarily OO-focused ones, numbers are given special treatment. They aren't really objects to speak of... they're just sort of there for you to use (they are called "primitive"), and the compiler/interpreter knows what to do with them. This is not the case in Ruby. In Ruby, any given number is really an object (an instance of the class Fixnum or Bignum, which are subclasses of Integer). Since classes have methods defined for them, you can do some very slick and unusual-looking things with numbers.

3.times do print "Foo! " end (Print works similarly to puts, but does not include a newline character at the end.)

Will give, as you might expect: Foo! Foo! Foo!

Something important to note, though, is that the "times" method is zero-indexed. So the following code:

```
3.times do |zork| puts "#{zork} " end
```

Will return: 0 1 2

If you have a need to iterate through a particular range of numbers, and (like me) you tend to get tripped up when trying to remember if something is indexed with 0 or 1, you can use the upto and downto methods. These methods provide functionality equivalent to a for loop, but without the extra syntax (although ruby does include a for loop construct, for completeness):

```
21.upto(25) do |n|
  print n
  print ", " unless n == 25
end
```

Again, without being familiar with the method definitions, it is pretty easy to take a look at the code and figure out what you think it should do. (Downto works the same as upto, but in reverse.) And you would probably be right:

```
21, 22, 23, 24, 25
```

If you need to iterate by step, there is also a step method that works the same way, but takes an extra argument (the step value).

The eql? method compares numbers on two levels at once; it checks to see if they have the same type and the same value. So consider the following two lines of code:

```
puts 1 == 1.0
puts 1.eql? 1.0
```

The first line of code resolves to "true" (one equals one), while the second line to "false" (integer is not the same as float). This looks odd and can be a bit counter-intuitive at first... although it can be a nice little "bonus" check to throw in if you frequently use both integers and floats in your code. And I do not believe this violates the principle of least surprise, since the "==" comparison operator does just what you think it should.

One other nice numeric feature of Ruby is that it will automatically create a Bignum object when performing an operation on a Fixnum object would cause it to overflow. You as a programmer don't need to worry about this detail, as it is handled transparently by the interpreter. The Bignum class has no upper size limit (well, other than the amount of memory available to Ruby), so it is literally impossible to overflow a numeric variable in Ruby.

7. Negated Control Structures
This was something brand new to me as far as programming goes. I'm sure there are other languages that contain a similar syntax, but Ruby was the first one that showed it to me.

The If construct has a negated version, called Unless. It works similarly to using a "not" keyword in the expression you are evaluating, or reversing the less than/greater than comparison:

```
unless first_name.length > 6 then      #equivalent to "if first_name.length <= 6"
  puts "You have a short first name."
else
  puts "You have a long first name."
end
```

The While construct has a similar negation ("until", can be compared to "while not"). Used as regular control structures as above, they are not necessarily better or worse than using normal if/while statements (though they do add to the readability of the code, in my opinion). Where they come in quite handy is when you use them as modifiers to other statements. In a more Ruby-esque fashion, I will share a method named "is_prime?" I wrote for the Fixnum class of object. Take a look (it incorporates some of the other features I've discussed so far):

```
class Fixnum
  def is_prime?
    return nil unless self >= 2
      return true if self == 2
      2.upto(self - 1) do |i|
        @is_prime = true
        @is_prime = false unless self.modulo(i) > 0
        break if @is_prime == false
      end
    @is_prime
  end
end
```

8. Native Support For Threading
Multithreading is something that I have not yet ventured much into as a programmer. The notion itself often brings up war stories of week-long debugging sessions, arcane incantations of having to have everything "just right", and the fear of a dreaded race condition that can nicely lock up an app.

What I have found, somewhat to my surprise, is that Ruby's native support for multithreading makes it seem not quite so bad. In fact, looking at it within the context of such a user-friendly language, I get a sense for how useful multithreading can be. The added bonus of native support means that multithreaded apps will run the same way on

whichever platform you choose... which means no switching libraries behind the scenes.

In short, creating a new thread is as simple as calling the "new" method on a thread object. You pass the method a block of code, and whatever is included in that block of code is executed in a new thread. The convention used in the Pickaxe Book ("Programming Ruby: The Pragmatic Programmer's Guide", available via the "Ruby Documentation" link in the Resources section of this article) is to create an array object to manage the threads collectively, which seems to make good sense to me. So, here is a simple example of a (useless) program that uses threading:

```
#create our thread array
threads = []

5.times do |i|
  #create 5 threads into the array
  threads[i] = Thread.new do
    #wait a random amount of time, then print a message
    sleep(rand(10))
    puts "I am thread number #{i}"
  end
end

#let each thread finish before ending the program
threads.each {|t| t.join}
```

The comments in the code should give you the general idea of what is going on. 5 new threads are created; each thread waits a random amount of time between 1 and 9 seconds (the rand method returns an integer greater than zero and less than 10, in this case) then prints a message. I ran this program a few times to test output, here is one of the results I got:

```
I am thread number 4
I am thread number 2
I am thread number 0
I am thread number 1
I am thread number 3
```

There are a number of structures and functions that Ruby provides as a way to manage thread operation, and I can see where things can get very hairy very quickly when thinking about things like variable scope (are they procedure local or thread local?), accessing shared resources (the dreaded race condition), exception handling (what do you do if one thread raises an exception?) and the like. The Pickaxe book does a decent job of presenting Ruby's way of grappling with these kinds of issues, though perhaps not in enough detail for anyone other than a novice. I, being a novice, have got enough to go on for now, and am curious to see how I can implement threading in any new Ruby code I manage to come up with.

9. Ruby On Rails
I have to confess, I don't know a whole lot about Ruby on Rails. What I *do* know, however, is that Ruby on Rails is a web development framework that uses Ruby and is designed to take a lot of the "mechanics" of web apps and do them automatically, so the developer can concentrate on the content and functionality of the app overall. This isn't a very easy concept to explain in general, but there is a video clip on the Ruby on Rails website that shows it in action. The programmer goes from nothing to a functional blog with a MySQL backend inside about 15 minutes.

In short, the Rails application gives you a sort of template that you can add code to. You can do very general things and Rails will automatically fill in the specifics for you. This lets you point the application at a database and reference the appropriate tables and fields in your code, and Rails will build the appropriate queries behind the scenes. This is an up-and-coming player in the web application field, but has caught on very rapidly and -- much like Ruby overall -- those who use it, swear by it.

The little I have seen of Ruby on Rails is enough to impress me, but it is also worth noting that knowledge of Ruby does not automatically make you a Ruby on Rails expert. It takes some effort to familiarize yourself with the particulars and terminology of the Rails framework, and to figure out what to type where in order to get it to do what you want. The video clip I referred to previously punctuates this point -- while informative, you are led through a whirlwind tour of Ruby on Rails with terms being thrown around left and right. It is an interesting video to watch, and does a fine job of illustrating the power of Ruby on Rails... but also shows that you have to know what you're doing. I have deliberately not done anything with Ruby on Rails yet, as I am hoping to spend more time familiarizing myself with the Ruby language first. However, it does leave the door open for another "bigger" project for someone looking to take their Ruby experience further.

10. Experts Take The Time To Help
Something that I have found in my brief foray into the world of Ruby is that, in general, people who program in Ruby do so because they like to. There are languages available that are more or less functionally equivalent and more widely available "out of the box" (e.g. Python and Perl); sometimes it can be a necessary "evil" to use one of these languages as a part of something larger that you are trying to accomplish (as an example, a part of my job consists of maintaining Visual Basic 6 code... whether I like to or not). As Ruby is a less-widespread language, it does not seem as though there are many critical processes that are dependent upon it for success.

6.4 - Why Rails

1. *Time is Money*
   While 10X productivity increases might only apply to specific projects, I haven't worked on a web application that could be developed faster in a framework other than Rails. Other knowledgeable people agree. With Rails, there's basically no configuration within an application. All of the needed components are available when you start your project. There's also zero turn-around time to view changes in code (Ruby is a scripted language, which means you don't need to recompile code to view changes in your web browser). I've found that "zero turnaround time" is one the best ways to encourage developers to write more maintainable code. With J2EE, developers might avoid refactoring bad code as the time to recompile and restart the web server serves as a major deterrent. With Ruby, it takes seconds to view the changes.
2. *Rails' earliest detractors have become it's biggest supporters*
   This may be the greatest sign - an acknowledgement by many of the leaders in the Java community that Rails has really gotten to the root of the web development problem. Many people who have a considerable investment in Java are endorsing Rails. In the words of Levar Burton of "Reading Rainbow" fame, don't take my word for it…
3. *Smaller Teams = Better Projects*
   Small teams can accomplish a lot with Rails - and there are few arguments against small teams in development projects. Instead of delegating work to a development team, I can personally lead and develop our applications (while doing it profitably), and I can afford to hire a few of the best instead of a lot of the rest.
4. *The Rails community is the cream of the crop*
   Developers who use Rails do it because they recognized there must be a better way - I'd argue they have a much better sense of the pieces needed to complete a successful project than a "heads down" coder. "Big Picture" coders need less management attention, which again leads to more successful small teams.
5. *…and it's easy to switch if you don't have Rails experience*
   While the number of Ruby developers pales in comparison to other languages, it is an extremely easy language to pickup. I have few reservations about hiring a developer without Ruby experience as they don't need to know a lot of details about the framework - they just need to be good coders.
6. *When usability matters (and when doesn't it?)*
   Many people associate Rails with great-looking applications. Because Rails has fantastic AJAX support - implementing AJAX is becoming trivially easy - the framework works great with small teams that have the ability to work on the interface design and the backend functionality. Usability is more important than ever - people have less and less patience for tedious applications today, and Rails gives developers the tools to make applications easier for its users.
7. *Higher Quality*
   We're not perfect, but the amount of broken code that's been rolled out in our Rails projects is far less than the number of bugs I've experienced in applications with other frameworks. This is largely because it is far easier to implement unit

testing in Rails as compared to other frameworks. Unit Testing is a way of automating tests - developers write scripts to test out parts of an application, and can then run these at anytime in the future. Just added a new feature and not sure if it will break other parts of the application? Simply run the tests and see if they all pass.

8. *But can it scale?*
   One of the major items of concern raised by those in the Java community is whether Rails can scale. The bottom line: there's no reason why it can't do it better (and cheaper) than Java. However, there hasn't been an eBay-like application written in Rails to prove this theory (but there haven't been a lot of new eBay-like applications as a whole either). Scaling is never simple, but there's no reason why it would be any harder in Rails.

6.5 - Rails three major goals:

- Simplicity: Developing applications should be easier, since many data-driven sites share a common set of parameters. For example, instead of focusing on writing code that will connect your application to a database, Rails handles all of that for you, thus allowing you to spend that time working with your actual application logic.
- Logical: Rails follows the Model-View-Controller (MVC) architecture for application development, which allows for a logical separation of your application logic, business rules, and user views. By keeping these things segregated, it not only makes initial development easy, it also makes updates and maintenance to your system quicker.
- Happiness: Since developing using Ruby on Rails strives to be simple and logical, it will in turn lead to a happy (and more productive) developer.

7.0 - Compare and Contrasts with Other Languages:

7.1 - Ruby versus Other Languages:

Languages can be divided into two types, compiled languages and scripting languages. Applications created using compiled languages are faster than those created using scripting languages. With compiled languages, you can easily access the operating system features, whereas with scripting languages you cannot. However, nowadays the distinction between compiled and scripting languages is not the same. Scripting languages are as fast as compiled languages. In addition, you can access the operating system features by using scripting languages such as Perl, Python, and Ruby. Ruby can be compared with other programming languages easily because of its resemblance to many programming languages, such as C, Perl, Python, Java, Smalltalk, and the shell scripts of UNIX. You actually can find a lot of syntax common between Ruby and C. Ruby can be compared with Perl and Python because both Perl and Python are scripting languages. What you can do in Ruby you also can do in Perl and Python. The only difference is that Ruby is much more flexible than Perl and Python. Creating applications in Ruby is much easier than creating applications in Perl and Python. Ruby can be compared with Smalltalk because both are true object-oriented programming languages. Ruby is similar to Java only in terms of showing multiple inheritance. Both Ruby and Java implement multiple inheritances indirectly, unlike C++, which shows multiple inheritances directly.

7.2 - Contrast ruby and other language:

Let's compare other languages with ruby. We take up Perl, Python, Tcl, Java, Eiffel and C++.

It is based on our own judgment and biased view, however, it does **not** intend to spread fibs. Please tell me mistakes, correction or supplementation for this page.

Perl

Ruby owes a lot of ideas Perl. Now, let's compare Perl with Ruby:

- Complicated OOP
- Easy to make code cyptogram
- Cannot represent *real* data structure (refferences have made better)

Python

Python deals with problem areas as similar as Ruby does. Now, let's compare Python with Ruby:

- incomplete OOP
  - Some data (numerical, list, dictionary, string, etc.) are not genuine object.
  - Extension modules in C don't realize genuine objects.
  - Tiresome self. are required when each access to instance variable
- Functions are first class object (Of cource, we know it is not weak)
- del, len, etc. are not Object-Oriented. (There are __len__, etc. but ...)
- Ugly method calling for superclass
- No class method (module functions alternate it)
- No way to automatically convert between long integer and small integer.
- Because there is no genuine GC,
  - Memory leak occur often
  - Writing extension modules is bothersome (some betterments in 1.5)
- Tuple and list has overlap functionally (not serious)
- Function objects are accepted as arguments, but iterators are more elegant.
- After all, declarations are featured (global)
- Slower than ruby (may be improved?)

Tcl

Tcl is well-known as a language for GUI (It is not true!). Now, let's compare Tcl with Ruby:

- No OOP feature in Tcl.
- Yet slow.
- After all, declarations are featured (global)
- Only strings are data type, so hard to handle references or objects. (objects are internally introduced from Tcl8.0)
- For Tk, other languages than Tcl get to hundle.

Java

Java is the language in the news one way or another. Now, let's compare Java with Ruby:

- Compiler
- Basic data types are not objects.
- Interfaces are sharable but no way to share implementations
- Typed variable
- Variables are typed though there are arrays only for Genericity.

Eiffel

An veteran would know the OOPL Eiffel. Now, let's compare Eiffel with Ruby:

- Too much difference in target

C++

C++ is famous OOPL in many places. Now, let's compare C++ with Ruby:

- Basically C++ and Ruby is Both Different type of languages

	Ruby	SmallTalk-80	C++	Java	Python	Perl5
Typing	dynamic	dynamic	static	static	dynamic	mostly dynamic
Runtime access to method names	yes	yes	no	yes	yes	yes
Runtime access to class names	yes	yes	no	yes	yes	yes
Runtime access to instance variable names	yes	yes	no	yes	yes	yes
Forwarding	yes	yes	no	no (?)	yes	yes
Metaclasses	yes	yes	no	yes	yes	no
Inheritance	mix-in	single	multiple	single	multiple	multiple
Access to super method	super	super	Superclass::method	super	(Python 2.1 syntax): super(MyClass,self).methodName	SUPER::methodName
root class	Object	Object (can have multiple)	none	Object	none	"UNIVERSAL" package
Receiver name	self	self	this	this	self	$_[0]
Private Data	yes	yes	yes	yes	yes	no
Private methods	yes	no	yes	yes	yes	no
Class Variables	yes	yes	yes	yes	yes	yes
Templates	not needed(1)	not needed	yes	no	not needed	not needed
Garbage Collection	yes	yes	no	yes	yes	reference counting

7.3 - RoR Vs PHP:

PHP is till the main compotator of the ruby. But the progress and development of the RoR framework is really alarming for the PHP.

7.4 - Why Ruby on Rails is better than PHP?

1 - Rails can handle search-friendly URLs. No more ?source=@@@HRJamie.

2 - Rails is the hot Web 2.0 technology. (Our office is a block from Adaptive Path, so we can just about see the glow from their aura.)

3 - Developers tell me things like "Rails is better for Web-based apps" and "You can collaborate easier in Rails" or "[something about dynamic libraries that is beyond me]."

7.5 - Criticizes on RoR and its reply:

1) it's a buzz language

As a consultant, I've come across many clients that only want to use ruby on rails simply because they've heard about it in the latest issue of their favorite tech magazine. Just because a language is the latest buzz doesn't mean it is more powerful a better choice for the job.

Reply: You're right that most-buzzed tool isn't always the best for the job, but sometimes it really is. Having a buzz isn't in and of itself a reason to avoid something - a tool should be evaluated on its suitability for the job rather than dismissed out of hand.

2) The language is young

It was released to the public in July of 2004 (source: http://en.wikipedia.org/wiki/Ruby_on_rails#History). Many other languages have a better history and track record.

Reply: Ruby on Rails is young. It is also not a language - it's a framework. Ruby is the language that underlies the framework, and was released in 1995. PHP was first released in 1994 (I know of no frameworks that attracted anything like the numbers Rails has prior to the debut of Rails; since then a number of 'em have sprouted up - including Cake, Symfony, and Zend's own framework (which is still at 0.1.4 as of this comment). Java was released in 1996 (the the JDK 1.0, at least), with Hibernate coming in 2001 and Spring in 2003. Again, however, youth alone is not necessarily a problem.

3) Poor support for IIS

Even though I personally use apache+linux or freebsd, there are still many corporations and sites running IIS. Although there are docs explaining how to configure it properly, the difficulty and lack of information leads me to believe that it is aimed at a *nix crowd.

Reply: IIS support is problematic, as is hosting more broadly. More and more hosting providers are adding Ruby and Rails support, however, so this will only improve over time. This would be a valid reason to avoid using Rails on a project - if you can't or don't want to change hosting providers.

4) creates dirty code

Here is an simple example of ruby code (it takes the factorial of a number):

```
def fact(n)
  if n == 0
    1
  else
    n * fact(n-1)
  end
end
```

This may work for small scripts and apps, but large applications with thousands of lines of code and hundreds of source files will be very difficult to manage with this type of syntax.

Reply: I'm not sure why the given example is dirty. Here's the same code in PHP:

```
function fact($n) {
if ($n == 0) {
return 1;
} else {
return ($n * fact($n - 1));
}
}
```

Certainly it could be cleaned up - eliminating multiple exit points from the function, for instance - but the PHP example suffers from the exact same problems. And of course, there's always another way to perform such a simple calculation. Perhaps this is 'cleaner'?

```
def fact(n)
f = 1
(1..n).each do |i|
f *= i
end
end
```

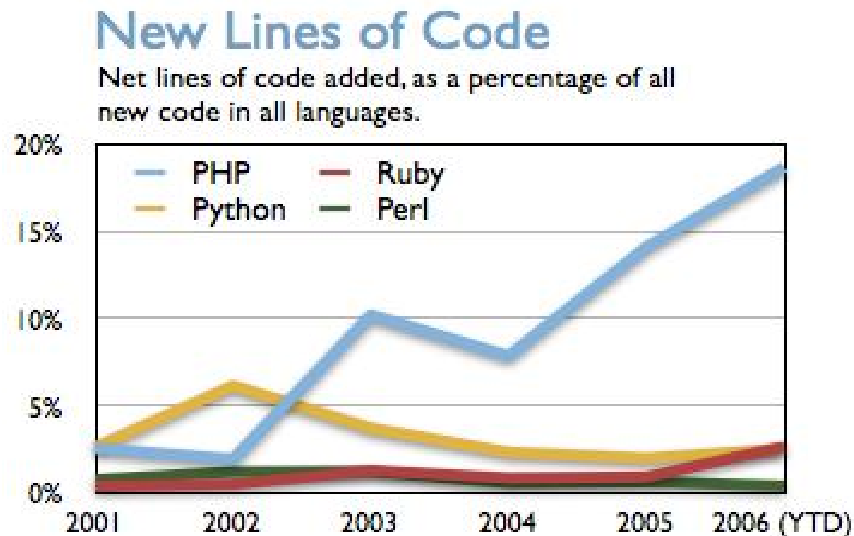
5) scalability

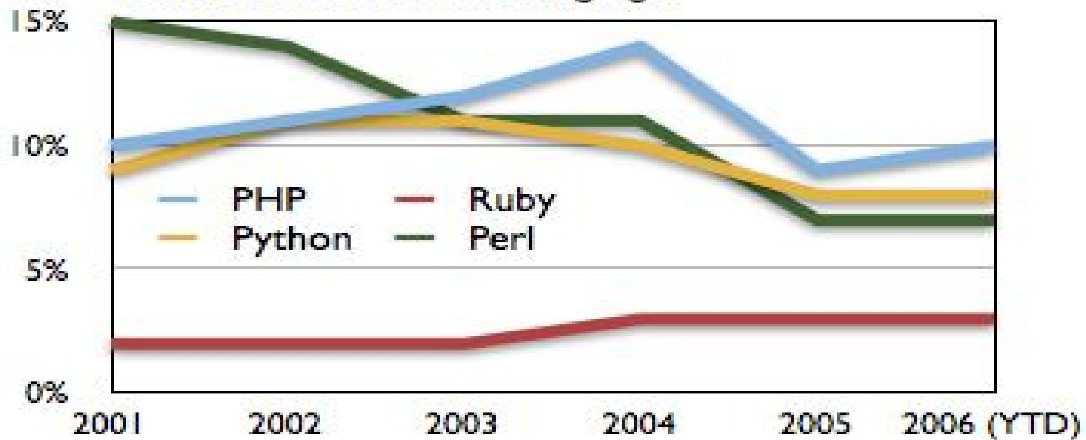
I'm not saying ruby does not scale, it's more of a question: Does it scale? this one also ties in with #2. I have not come across any large, feature rich, and popular website or service using ruby on rails as its main powerhouse. Until I can be confident that it is indeed possible, I am going to focus my attenion on other languages.

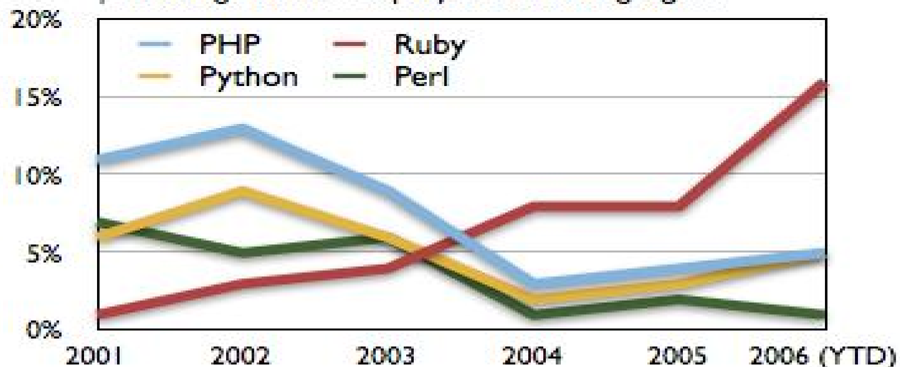
Reply: I'm not sure what your metric for a "large, feature rich, and popular website or service" is. 37signals' apps have supported over 500,000 users - but the might not be classified as 'feature-rich', since the 37s philosophy is one of simplicity. A List Apart uses RoR and is 3343 on the Alexa traffic ratings; Penny Arcade uses it and is 1497 (for comparison: Yahoo is #1, the Washington Post is 182, comics.com is 1523, and Ford is 3920). One of the underlying philosophies of RoR is 'share nothing', so a properly-coded Rails app's scalability is determined more by the supporting software (e.g., the database and web server) than by Ruby or Rails itself. There's a ton of discussion about this on the web at large for anyone who's interested

7.6 – Comparison based on Developers choice:

"If PHP Eats Rails for Breakfast the Rails Eats PHP on Dinner:"







Source: **www.ohloh.net/wiki/articles/php_eats_rails**

Curiously, it seems that the number of new PHP projects has declined. Perhaps this implies that much of the new PHP code is being added to established projects. If so, one might explain the rise in new code as a consequence of maturing code bases: as developers gain long experience within the framework of an older application, their output increases. Alternatively, perhaps the small number of new projects in fact contains huge amounts of code which has been repurposed [a nice word for "cut-and-pasted"] from older applications. This is just speculation, and I invite discussion.

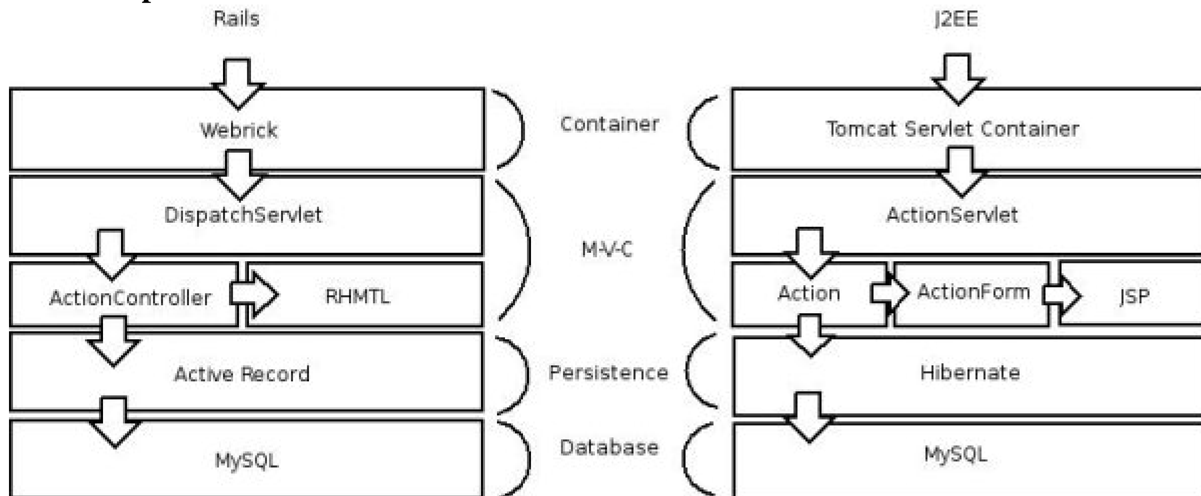
7.7 - Ruby on Rails and J2EE: Is there room for both?

Ruby on Rails is a Web application framework that aims to provide an easy path to application development. In fact, the framework's proponents claim that Ruby on Rails developers can be up to ten times more productive than they would be when using traditional J2EE frameworks. (Read the article titled "Rolling with Ruby on Rails" for more on this claim. While this statement has been the source of considerable debate in the Rails and J2EE communities, little has actually been said about how Rails and J2EE architectures compare. This article will contrast the Rails framework against a typical J2EE implementation using common open source tools that are regularly found in enterprise applications.

7.8 - Rails and a typical J2EE Web stack

Figure 1 compares the Rails stack to a typical J2EE Web stack comprised of the Tomcat servlet container, the Struts Web application framework, and the Hibernate persistence framework.

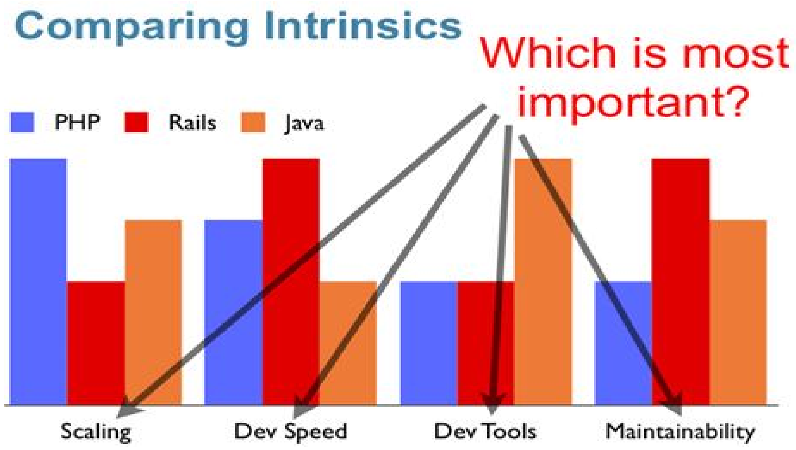
7.9 - **Comparison of Rails and J2EE stacks**



Source: **www.radrails.org**

As you can see, the fundamental difference between the Rails stack and the components that make up a common J2EE-based Web application is small. Both have a container in which the application code will execute; an MVC framework that helps to separate the application's model, view, and control; and a mechanism to persist data.

7.10 - PHP vs Java vs Ruby:



Source: **www.radrails.org**

8.0 - Other Frameworks Computability

8.1 - Ajax on Rails:

In a few short months, Ajax has moved from an obscure and rarely used technology to the hottest thing since sliced bread. This article introduces the incredibly easy-to-use Ajax support that is part of the Ruby on Rails web application framework. This is not a step-by-step tutorial, and I assume that you know a little bit about how to organize and construct a Rails web application. If you need a quick refresher, check out

Just in case you've been stranded on a faraway island for most of the year, here's the history of Ajax in 60 seconds or less.

8.2 - Traditional Web App vs. an Ajax App:

Let me distill the essence of an Ajax web application by examining a use case: inserting a new item into a list.

A typical user interface displays the current list on a web page followed by an input field in which the user can type the text of a new item. When the user clicks on a Create New Item button, the app actually creates and inserts the new item into the list.

At this point, a traditional web application sends the value of the input field to the server. The server then acts upon the data (usually by updating a database) and responds by sending back a new web page that displays an updated list that now contains the new item. This uses a lot of bandwidth, because most of the new page is exactly the same as the old one. The performance of this web app degrades as the list gets longer.

In contrast, an Ajax web application sends the input field to the server in the background and updates the affected portion of the current web page in place. This dramatically increases the responsiveness of the user interface and makes it feel much more like a desktop application.

You can see this for yourself. Below are links to two different weblogs, one that uses Ajax to post comments and another that does not. Try posting some comments to each one:

8.3 - How Rails Implements Ajax

Rails has a simple, consistent model for how it implements Ajax operations.

Once the browser has rendered and displayed the initial web page, different user actions cause it to display a new web page (like any traditional web app) or trigger an Ajax operation:

1. A trigger action occurs. This could be the user clicking on a button or link, the user making changes to the data on a form or in a field, or just a periodic trigger (based on a timer).
2. Data associated with the trigger (a field or an entire form) is sent asynchronously to an action handler on the server via XMLHttpRequest.
3. The server-side action handler takes some action (that's why it is an *action handler*) based on the data, and returns an HTML fragment as its response.
4. The client-side JavaScript (created automatically by Rails) receives the HTML fragment and uses it to update a specified part of the current page's HTML, often the content of a <div> tag.

An Ajax request to the server can also return any arbitrary data, but I'll talk only about HTML fragments. The real beauty is how easy Rails makes it to implement all of this in your web application.

Using link_to_remote

Rails has several helper methods for implementing Ajax in your view's templates. One of the simplest yet very versatile methods is link_to_remote(). Consider a simple web page that asks for the time and has a link on which the user clicks to obtain the current time. The app uses Ajax via link_to_remote() to retrieve the time and display it on the web page.

My view template (*index.rhtml*) looks like:

```
<html>
  <head>
    <title>Ajax Demo</title>
    <%= javascript_include_tag "prototype" %>
  </head>
  <body>
    <h1>What time is it?</h1>
    <div id="time_div">
      I don't have the time, but
      <%= link_to_remote( "click here",
                    :update => "time_div",
                    :url =>{ :action => :say_when }) %>
      and I will look it up.
    </div>
  </body>
</html>
```

There are two helper methods of interest in the above template, marked in bold. javascript_include_tag() includes the Prototype JavaScript library. All of the Rails Ajax features use this JavaScript library, which the Rails distribution helpfully includes.

The link_to_remote() call here uses its simplest form, with three parameters:

1. The text to display for the link--in this case, "click here".
2. The id of the DOM element containing content to replace with the results of executing the action--in this case, time_div.
3. The URL of the server-side action to call--in this case, an action called say_when.

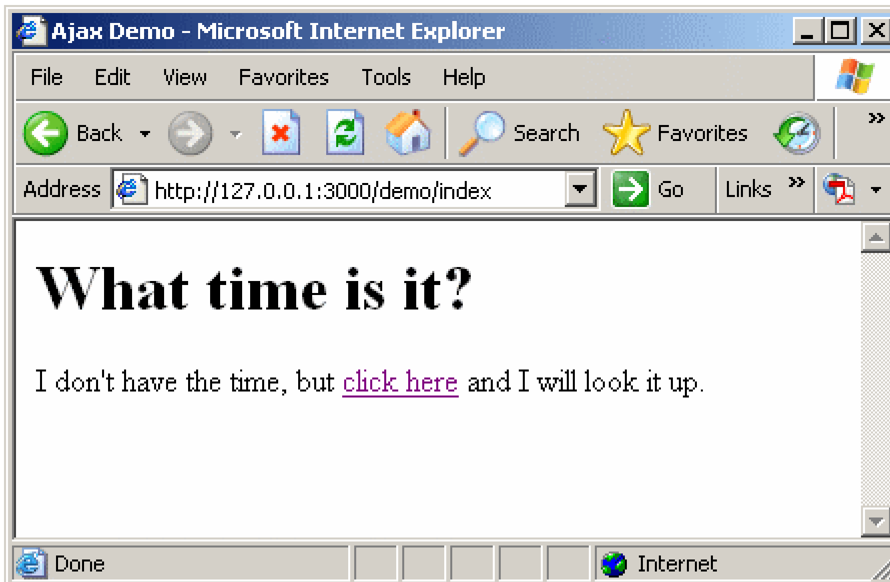


Figure 1. Before clicking on the link

My controller class looks like:

```
class DemoController < ApplicationController
  def index
  end

  def say_when
    render_text "<p>The time is <b>" + DateTime.now.to_s + "</b></p>"
  end
end
```

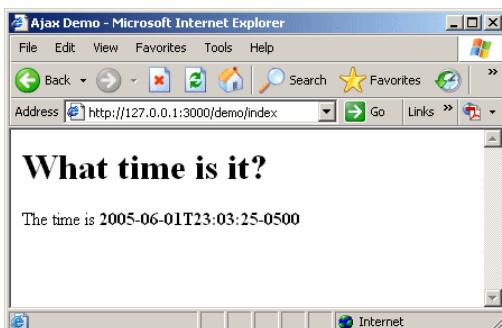


Figure 2. After clicking on the link

The index action handler doesn't do anything except letting Rails recognize that there is an index action and causing the rendering of the *index.rhtml* template. The say_when action handler constructs an HTML fragment that contains the current date and time. Figures 1 and 2 show how the index page appears both before and after clicking on the "click here" link.

When the user clicks on the "click here" link, the browser constructs an XMLHttpRequest, sending it to the server with a URL that will invoke the say_when action handler, which returns an HTML response fragment containing the current time. The client-side JavaScript receives this response and uses it to replace the contents of the <div> with an id of time_div.

It's also possible to insert the response, instead of replacing the existing content:

```
<%= link_to_remote( "click here",
            :update => "time_div",
            :url => { :action => :say_when },
            :position => "after" ) %>
```

I added an optional parameter **:position => "after"**, which tells Rails to insert the returned HTML fragment after the target element (time_div). The position parameter can accept the values before, after, top, and bottom. top and bottom insert inside of the target element, while before and after insert outside of the target element.

In any case, now that I added the position parameter, my "click here" link won't disappear, so I can click on it repeatedly and watch the addition of new time reports.

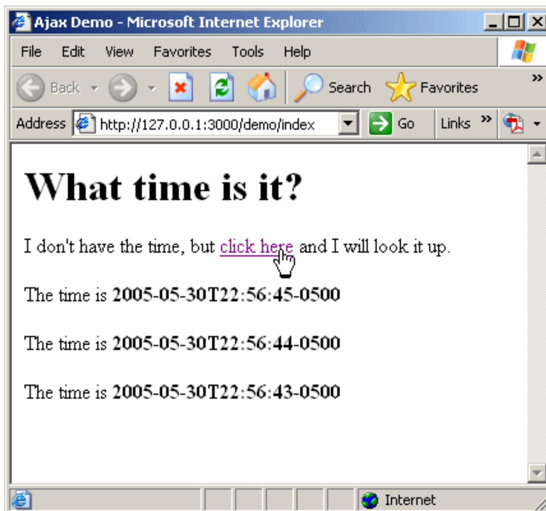


Figure 3. The position option inserts new content

8.4 - Flex Builder 2:

When using Flex you have several options to choose from for back-end server software. So why might you want to choose Rails? Ruby on Rails, like Flex, is a well thought out, elegantly simple framework. As you will see, Rails uses code generation and metaprogramming to make it incredibly easy to integrate with a database using almost no SQL code. Furthermore, when you use Rails, you also get to use Ruby, a programming language that is both extremely powerful and easy to use. Using Flex and Ruby on Rails, you will be able to get more done with less code.

Flex + Rails + Ruby = RIA Nirvana.

8.5 - How Flex Impliments on RoR:

To get started, you'll create the user interface for displaying the data. Open Flex Builder and go to File > New > Flex Application to create a new Project called flex_issuetracker. Once the project has been created, open flex_issuetracker.mxml and add the following code:

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

            [Bindable]
            private var statusArray:Array = ["Opened", "Assigned",
"Closed"];

            [Bindable]
            private var priorityArray:Array = ["Blocker", "Critical",
"Major", "Minor", "Trivial"];

        ]]>
    </mx:Script>
    <mx:VDividedBox x="0" y="0" height="100%" width="100%">
        <mx:Panel width="100%" height="376" layout="absolute"
title="Create/Update Bugs">
            <mx:Form x="10" y="10" width="930" height="280">
                <mx:FormItem label="Reported by">
                    <mx:TextInput width="220" id="reportedby"
text="{bugs_dg.selectedItem.reportedby}"/>
                </mx:FormItem>
                <mx:FormItem label="Assigned to">
                    <mx:TextInput width="220" id="assignedto"
text="{bugs_dg.selectedItem.assignedto}"/>
                </mx:FormItem>
                <mx:FormItem label="Description">
                    <mx:TextArea width="336" height="111"
id="description" text="{bugs_dg.selectedItem.description}"/>
```

```
                </mx:FormItem>
                <mx:FormItem label="Status" width="287">
                    <mx:ComboBox width="199" id="status"
selectedIndex="{statusArray.indexOf(bugs_dg.selectedItem.status)}">
                        <mx:dataProvider>
                            {statusArray}
                        </mx:dataProvider>
                    </mx:ComboBox>h
                </mx:FormItem>
                <mx:FormItem label="Priority">
                    <mx:ComboBox width="199" id="priority"
selectedIndex="{priorityArray.indexOf(bugs_dg.selectedItem.priority)}">
                        <mx:dataProvider>
                            {priorityArray}
                        </mx:dataProvider>
                    </mx:ComboBox>
                </mx:FormItem>
            </mx:Form>
            <mx:ControlBar horizontalAlign="right">
                <mx:Button label="Clear" />
                <mx:Button label="Update" />
                <mx:Button label="Create" />
            </mx:ControlBar>
        </mx:Panel>
        <mx:Panel width="100%" height="444" layout="absolute"
title="Bugs">
            <mx:DataGrid x="0" y="0" width="100%" height="100%"
id="bugs_dg" >
                <mx:columns>
                    <mx:DataGridColumn headerText="Reported by"
dataField="reportedby"/>
                    <mx:DataGridColumn headerText="Assigned to"
dataField="assignedto"/>
                    <mx:DataGridColumn headerText="Description"
dataField="description"/>
                    <mx:DataGridColumn headerText="Status"
dataField="status"/>
                    <mx:DataGridColumn headerText="Priority"
dataField="priority"/>
                </mx:columns>
            </mx:DataGrid>
            <mx:ControlBar horizontalAlign="right">
                <mx:Button label="Delete" />
            </mx:ControlBar>
        </mx:Panel>
    </mx:VDividedBox>
</mx:Application>
```

The code in this MXML file does several things:

1. It creates a basic form for submitting new bugs and editing existing bugs.

   Note: The values in the FormItems are updated automatically using databinding based on the values of the selected item in the datagrid discussed in the following

sections. The form is wrapped inside of a Panel component. Using a Panel component has two benefits:

a. It generally produces a nicer looking interface and provides a label for you to describe the content/purpose of the panel.
b. You can attach a ControlBar to your panel so that you can add additional controls that work with the content. The ControlBar at the bottom of this panel has three buttons. One that creates new bugs, another for updating an existing bug, and lastly a button for clearing the form. Currently these buttons do not do anything. They are just placeholders for now, at least until you set up your Rails back end for the application.

```
<mx:Panel width="100%" height="376" layout="absolute"
title="Create/Update Bugs">
    <mx:Form x="10" y="10" width="930" height="280">
        <mx:FormItem label="Reported by">
            <mx:TextInput width="220" id="reportedby"
text="{bugs_dg.selectedItem.reportedby}"/>
        </mx:FormItem>
        <mx:FormItem label="Assigned to">
            <mx:TextInput width="220" id="assignedto"
text="{bugs_dg.selectedItem.assignedto}"/>
        </mx:FormItem>
        <mx:FormItem label="Description">
            <mx:TextArea width="336" height="111"
id="description" text="{bugs_dg.selectedItem.description}"/>
        </mx:FormItem>
        <mx:FormItem label="Status" width="287">
            <mx:ComboBox width="199" id="status"
selectedIndex="{statusArray.indexOf(bugs_dg.selectedItem.status)}
">
                <mx:dataProvider>
                    {statusArray}
                </mx:dataProvider>
            </mx:ComboBox>h
        </mx:FormItem>
        <mx:FormItem label="Priority">
            <mx:ComboBox width="199" id="priority"
selectedIndex="{priorityArray.indexOf(bugs_dg.selectedItem.priori
ty)}">
                <mx:dataProvider>
                    {priorityArray}
                </mx:dataProvider>
            </mx:ComboBox>
        </mx:FormItem>
    </mx:Form>
    <mx:ControlBar horizontalAlign="right">
        <mx:Button label="Clear" />
        <mx:Button label="Update" />
        <mx:Button label="Create" />
    </mx:ControlBar>
</mx:Panel>
```

2. This code creates a `DataGrid` component that both displays information about existing bugs and allows you to select a bug record to update or delete. The app uses the `dataField` parameter in the columns to specify the name of the attribute in the `dataprovider` that your application will use to populate this column. You do not have any data yet, but it is important to keep in mind that the column names in your table should match the values specified here. The `DataGrid` component is also wrapped in a panel with a control bar. This control bar has one button that will be used for deleting the bug that is currently selected in the `DataGrid` component.

```
3.  <mx:Panel width="100%" height="444" layout="absolute"
    title="Bugs">
4.      <mx:DataGrid x="0" y="0" width="100%" height="100%"
    id="bugs_dg" >
5.          <mx:columns>
6.              <mx:DataGridColumn headerText="Reported by"
    dataField="reportedby"/>
7.              <mx:DataGridColumn headerText="Assigned to"
    dataField="assignedto"/>
8.              <mx:DataGridColumn headerText="Description"
    dataField="description"/>
9.              <mx:DataGridColumn headerText="Status"
    dataField="status"/>
10.             <mx:DataGridColumn headerText="Priority"
    dataField="priority"/>
11.         </mx:columns>
12.     </mx:DataGrid>
13.     <mx:ControlBar horizontalAlign="right">
14.         <mx:Button label="Delete" />
15.     </mx:ControlBar>
16. </mx:Panel>
```

17. The panels are wrapped inside of a `VDividedBox` component, which allows you to dynamically resize the area that is allotted to the form or the DataGrid component using a divider that the user can drag.
18. The code declares two arrays that are used as data providers to populate the `ComboBox` components that are part of the form

```
19. [Bindable]
20. private var statusArray:Array = ["Opened", "Assigned",
    "Closed"];
21.
22. [Bindable]
23. private var priorityArray:Array = ["Blocker", "Critical",
    "Major", "Minor", "Trivial"];
```

Setting up your Rails application

Now that you've coded the interface, the next step is to set up your Rails application. First, you need to create a new database. This tutorial uses MySQL, but Rails is capable of supporting many other databases, too. Open your MySQL Command Line Client and enter the following line:

```
CREATE flex_issue_tracker_development
```

That is all the SQL you are going to need to write. Rails will take care of the rest.

To create your Rails application, open up a command-line window and change the directory to the location that you would like to use for your application. I generally use C:\Dev. Once you're in your desired location on your server, enter the following line:

```
rails issue_tracker
```

This command generates all of the folders and files that you will need for your Rails application. Part of the power of Rails is that it promotes consistency across projects by ensuring that all Rails applications have the same structure.

Next, create the model for this application. In Rails, the model is the code that wraps a table in your database. To create your model, enter the following line:

```
cd issue_tracker
```

To change the directory to your Rails app.

Next, enter the following line:

```
ruby script/generate model bug
```

Finally, you'll create the controller. The controller contains the logic for interacting between the view (in this case Flex) and the model. To create your controller enter the following line:

```
ruby script/generate controller bugs
```

Now, inside your Rails application folder, go to app > controllers. You will see the bugs_controller.rb file that you just generated. Open this file and add the following code:

```
def create
  @bug = Bug.new(params[:bug])
  @bug.save
  render :xml => @bug.to_xml
end

def list
  @bugs = Bug.find :all
  render :xml => @bugs.to_xml
end

def update
  @bug = Bug.find(params[:id])
  @bug.update_attributes(params[:bug])
  render :xml => @bug.to_xml
end

def delete
```

```
  @bug = Bug.find(params[:id])
  @bug.destroy
  render :xml => @bug.to_xml
end
```

These methods implement basic CRUD functionality, where the `list` method corresponds to read. The following is a brief explanation of the key parts of this code:

1. `params`: This variable is a hash that is automatically populated with the values that you pass an argument to the Rails method.
2. `Bug`: This refers to the ActiveRecord that you created when you generated your model. Note that it is a convention in Rails for the table name to be the plural of the model's name. Using this convention, when Rails encounters a model called "Bug" it looks for a table called "bugs" in the database.
3. `find`: The `find` method searches the records in a database for the specified table. `Bug.find :all` returns all of the records for the bugs table. `Bug.find(params[:id])` returns the first record in the bugs table whose id matches the value contained in the `params` hash. Please note that it is also a convention in Rails for each table to have column named id that is the primary key.
4. `new`: This creates a new ActiveRecord instance.
5. `save`: This saves the record to the database.
6. `update_attributes`: This updates the record using the values provided in the hash.
7. `destroy`: This deletes a record from the database.
8. `to_xml`: This builds XML document from the model.
9. Lastly, it is important to note that all methods in Ruby return a value. When you do not use `return`, Ruby methods return the value of the last line inside the method.

Now you must specify to your Rails application how to access the database that you just created. To do this, go to your Rails application folder (e.g., C:\Dev\issue_tracker) and open up the config folder. Locate the database.yml file and open it. Edit this file so that it reads as follows:

```
development:
 adapter: mysql
        database: flex_issue_tracker_development
        username: <your username here>
        password: <your password here>
        host: localhost
```

Be sure to fill in your own username and password. For this tutorial, you are only going to be running Rails in development mode, so you do not need to worry about the production and test sections of the file.

Now that your Rails app can access your database, your next step is to add a table to your database for storing your bugs. The easiest way to do this is to use Rails migrations.

Migrations virtually eliminate the need for using SQL statements to set up your database. Using migrations also has many other benefits, such as being database agnostic; a discussion of the benefits is beyond the scope of this tutorial.

To implement the migration for your app, go to db > migrate. You will see a file called 001_create_bugs.rb that was automatically created when you created your model. Edit this file to look like the following:

```
class CreateBugs < ActiveRecord::Migration
  def self.up
    create_table :bugs do |t|
      t.column :reportedby, :string
           t.column :assignedto, :string
          t.column :description, :text
      t.column :status, :string
           t.column :priority, :string
    end
  end

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

The up method creates a table called bugs with six columns:

- id (automatically generated primary key)
- reportedby
- assignedto
- description
- status
- priority

Note that the column names correspond to the values in the `dataField` attributes in the `DataGrid` column that you created in your Flex interface. The down method removes the table.

To run the migration, go to the command line and enter:

```
rake migrate
```

Now, if you go back to your MySQL Command Line Client and type:

```
USE flex_issue_tracker_development
DESCRIBE bugs;
```

You will see the following in the command-line window:

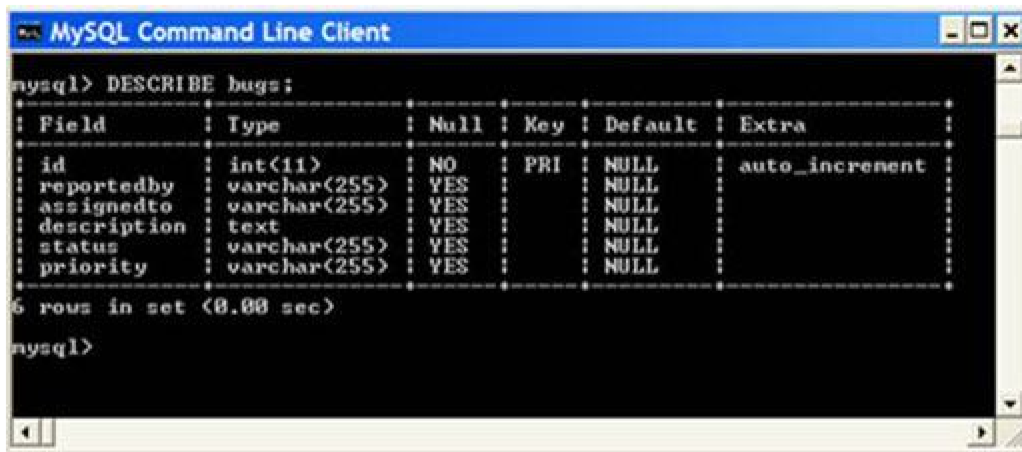


Figure 1. The command-line window

Finally, set up your Rails app to boot up the server. Ruby comes with its own handy server called WEBrick. To start WEBrick go to the command line and enter:

```
ruby script/server
```

Note that the server defaults to running on port 3000. This will be important in the next section.

That's it. Your Rails app is all ready. If you open up an Internet browser and type `http://localhost:3000` you will see the Rails welcome page.

Integration

Now that you've got your Flex interface, your Rails app, and your database all set up, the next step is to get Flex talking with Rails. Fortunately, this is really easy. You are going to be using the HTTPService to call the `create`, `list`, `update`, and `delete` methods that you added to the bugs_controller.rb file in your Rails application. Add the following code to your Flex application:

```
<mx:HTTPService id="listBugs" url="http://localhost:3000/bugs/list"
useProxy="false" method="GET"/>
        <mx:HTTPService id="updateBug"
url="http://localhost:3000/bugs/update" useProxy="false" method="POST"
result="listBugs.send()"/>
        <mx:HTTPService id="deleteBug"
url="http://localhost:3000/bugs/delete" useProxy="false" method="POST"
result="listBugs.send()"/>
        <mx:HTTPService id="createBug"
url="http://localhost:3000/bugs/create" useProxy="false" method="POST"
result="listBugs.send()"
   contentType="application/xml">
        <mx:request xmlns="">
            <bug>
                <reportedby>{reportedby.text}</reportedby>
```

```
            <assignedto>{assignedto.text}</assignedto>
            <description>{description.text}</description>
            <status>{status.text}</status>
            <priority>{priority.text}</priority>
        </bug>
    </mx:request>
   </mx:HTTPService>
```

Each of the four HTTPServices that you just added calls one of the methods in your Rails application. Notice that the `url` attribute in each HTTPService follows the structure: `http://<server address>/<controller name>/<method name>`.

Note: When you are ready to deploy your application, change these URLs from absolute to relative (for instance, `url="http://localhost:3000/bugs/list"` becomes `url="bugs/list"`)and move the SWF files and HTML files that are generated by Flex into the public folder in your Rails app.

Next, edit the Clear, Update, and Create buttons that are under your form to look like the following:

```
<mx:Button label="Clear" click="clearForm()"/>
    <mx:Button label="Update" click="sendBugUpdate(); clearForm()"/>
    <mx:Button label="Create" click="createBug.send(); clearForm()"/>
```

Then, edit the Delete button under your `DataGrid` component to look like the following:

```
<mx:Button label="Delete"
click="deleteBug.send({id:bugs_dg.selectedItem.id}); "/>
```

Go to the opening tag of the `DataGrid` component and add a `dataProvider` that binds to the value of the last result returned by the `listbugs` HTTPService, as follows:

```
<mx:DataGrid x="0" y="0" width="100%" height="100%" id="bugs_dg"
dataProvider="{listBugs.lastResult.bugs.bug}">
```

Now go to the top of your application and add a `creationComplete` event that calls the `listBugs` HTTPService using the `send()` method.

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute" creationComplete="listBugs.send()">
```

Lastly, add the following two functions inside the `<mx:Script>` block, just below the two arrays:

```
private function clearForm():void
    {
        reportedby.text = ""
        assignedto.text = "";
        description.text = "";
        status.selectedIndex = 0;
```

```
        priority.selectedIndex = 0;
    }

     private function sendBugUpdate():void
    {
            var bugUpdate:Object = new Object();
            bugUpdate['id'] = bugs_dg.selectedItem.id;
            bugUpdate['bug[reportedby]'] = reportedby.text;
             bugUpdate['bug[assignedto]'] = assignedto.text;
             bugUpdate['bug[description]'] = description.text;
             bugUpdate['bug[status]'] = status.text;
             bugUpdate['bug[priority]'] = priority.text;

             updateBug.send(bugUpdate);
    }
```

That's it. With surprisingly little code, your issue tracker is ready to roll. You now have an interface and a back end that support basic CRUD operations. To test it out, click run inside Flex Builder. Once the app launches, fill in the form with some data for a new bug. Click the Create button and you will see a new entry appear in the datagrid. Add a few more bugs, and then close the application. Go back in to Flex Builder and click run again. This time, when the application launches, you will see the bugs that you already entered appear in the datagrid. You can now update these bugs, delete them, or create new bugs and your changes will automatically be persisted to your database.

9.0 - Performance and Scalability

- Rails apps are compatible with Unix and Windows environments
- Rails apps can be deployed on multiple web server configurations
  - WEBrick – a standalone Ruby web server, easy for development
  - FastCGI – running behind Apache or LightTPD
  - CGI
- Easy control of the deployment with Capistrano
  - A scripting tool to control the deployment of the application on multiple sites
- Easy control of schema migration
  - rake migrate VERSION=3

	Ease of Setup	Speed	Scalability
WEBrick	□□□□□	□□	□
Apache-CGI	□□□□	□	□□
Apache-FCGI	□□	□□□□	□□□□
LightTPD-FCGI	□	□□□□□	□□□□□

- Support many database back-ends
  - Oracle, MySQL, PosgreSQL, SQLite
- Built-in support for AJAX
  - Javascript
  - Document Object Model (DOM)
  - XMLHttpRequest
- Built-in functional and unit testing
  - Code tests while you develop
  - Test templates are automatically generated
- Rails comes with a complete toolbox of classes, helpers and utilities

10.0 – Limitations of our Work:

- Most of the books were in Japanese Language
- No Support with out online
- No Developing Partner in Bangladesh
- Bugs in Rails Framework
- Bugs in Gems Kit
- Installation Problem
- Errors in tutorials

11.0 - Future Development Areas

- Advance Tools to develop RubyOnRails
- User Interface Design
- Solve the Bugs
- Easy accessibility of Core Library
- Basic Tutorial Development rather than converting
- Solve the Bugs in Gems Kit
- Create a common relation with Ruby and Rail's Framework

12.0 - References:

Web Page:

www.rubyonrails.com
www.econsultant.com/web-developer/ruby-rails-tutorials/
www.radrails.org
www.oracle.com/technology/pub/articles/saternos-ror-faq.html
www.snapvine.com/code/ragi/
www.ohloh.net/wiki/articles/php_eats_rails
www.locusfoc.us/ram
www.rubyinside.com/category/ruby-on-rails/page/2/
www.rubyinside.com
www.flex.org

Books:

John.Wiley.and.Sons.Making.Use.of.Ruby.ebook-TLFeBOOK

Manning.Publications.Ruby.for.Rails.Ruby.Techniques.for.Rails.Developers

Pragmatic.Bookshelf.Rails.Recipes.Jun.2006

Programming Ruby

Ruby on Rails: Up and Running

Community:

wiki.rubyonrails.com

api.rubyonrails.com

Real Applications:

www.basecamphq.com

www.campfirenow.com

www.odeo.com

13.0 – Contact:

Md. Fazle Munim

Student Id: 04241004

BRAC University, Dhaka, Bangladesh.

Email: sayket@gmail.com

Phone: 880-152408633, 880-1727291545

Residence Address: 70/71, Boro Moghbazar Romna Dhaka-1217.

Rozina Sultana

Student Id:02101015

BRAC University, Dhaka, Bangladesh.

Email:rozina_sultana@yahoo.com

Phone :01715025652

Residence Address:797 Ibrahimpur ,Dhaka Cantonment Dhaka .