This paper was originally presented at the Southwest Fox conference
in Gilbert, Arizona in October, 2010. http://www.swfox.net

An Introduction to Ruby and Rails

Rick Borup
Information Technology Associates
701 Devonshire Dr, Suite 127
Champaign, IL 61820
Voice: (217) 359-0918
Fax: (217) 398-0413
Email: rborup@ita-software.com

Ruby is a powerful, object-oriented, open-source language with a clean and easily understandable syntax. Originally released to the public in 1995, wide-spread interest in the Ruby language really began to take off in 2003 with the release of the Ruby on Rails® framework. Also known as "RoR" or simply "Rails", Ruby on Rails is an open-source Web framework that makes it possible to quickly and easily create data-based Web applications. Together, the Ruby language and the Ruby on Rails framework can become an exciting and powerful addition to any developer's toolbox. This session introduces both the Ruby language and the Rails framework, focusing on their installation and use on a Windows® platform with a perspective geared toward experienced Visual FoxPro® developers.

Table of Contents

Introduction

This paper is an introduction to Ruby and to the Ruby on Rails framework. Ruby is the name of the programming language, while Ruby on Rails is the name of a framework for developing data-based websites using Ruby. The framework is frequently referred to as simply Rails, which is how I most often refer to it in this paper. In print, although not when spoken, the framework is also sometimes referred to as RoR.

When I use the phrase Ruby and Rails, as in the title of this paper, I'm referring to both the Ruby language and the Rails framework, but it's important to remember that they're two separate things. Please don't confuse Ruby on Rails (the framework) with Ruby and Rails, which is just my way of referring to the two together.

This paper is intended for developers who are interested in installing and running Ruby and Ruby on Rails on a Windows® computer. The majority of Ruby developers very likely work on Mac computers, but that's not a requirement. There is a large and probably growing number of developers working with Ruby and Rails on Windows machines, and there is a large body of knowledge available to assist you if you want to do so too.[1]

Finally, this paper is geared specifically toward developers with experience in Visual FoxPro®. At various times I'll point out similarities and differences between Ruby and Visual FoxPro. Experienced Visual FoxPro developers will recognize these similarities and differences even when they're not explicitly stated.

Installing Ruby and Rails on Windows

The Ruby language and the Ruby on Rails framework are both free. Installation is simply a matter of downloading the desired packages and installing them.

On Windows machines, you have two basic choices: you can install a bundled package called InstantRails, which includes both the Ruby language and the Rails framework (along with some other components you may or may not want), or you can use the RubyInstaller to install the Ruby language and then install the Rails framework and other required components separately.

Regardless of which way you choose to install Ruby and Rails, be aware that you do not automatically get an integrated development environment (IDE) like you're used to with Visual FoxPro or Visual Studio. There are IDEs for working with Ruby and Rails, but they're third-party products. An IDE can be helpful but is certainly not necessary. I'll mention a couple of the popular IDEs later on, but I don't use them in this session and while you're first learning Ruby and Rails I suggest you don't use them either. The command line and a good text editor are all you need.

[1] I attended the Ruby Midwest conference in Kansas City, MO this past summer. I was concerned about having the only Windows machine at the conference, but I needn't have worried: there were several others.

Installing Instant Rails

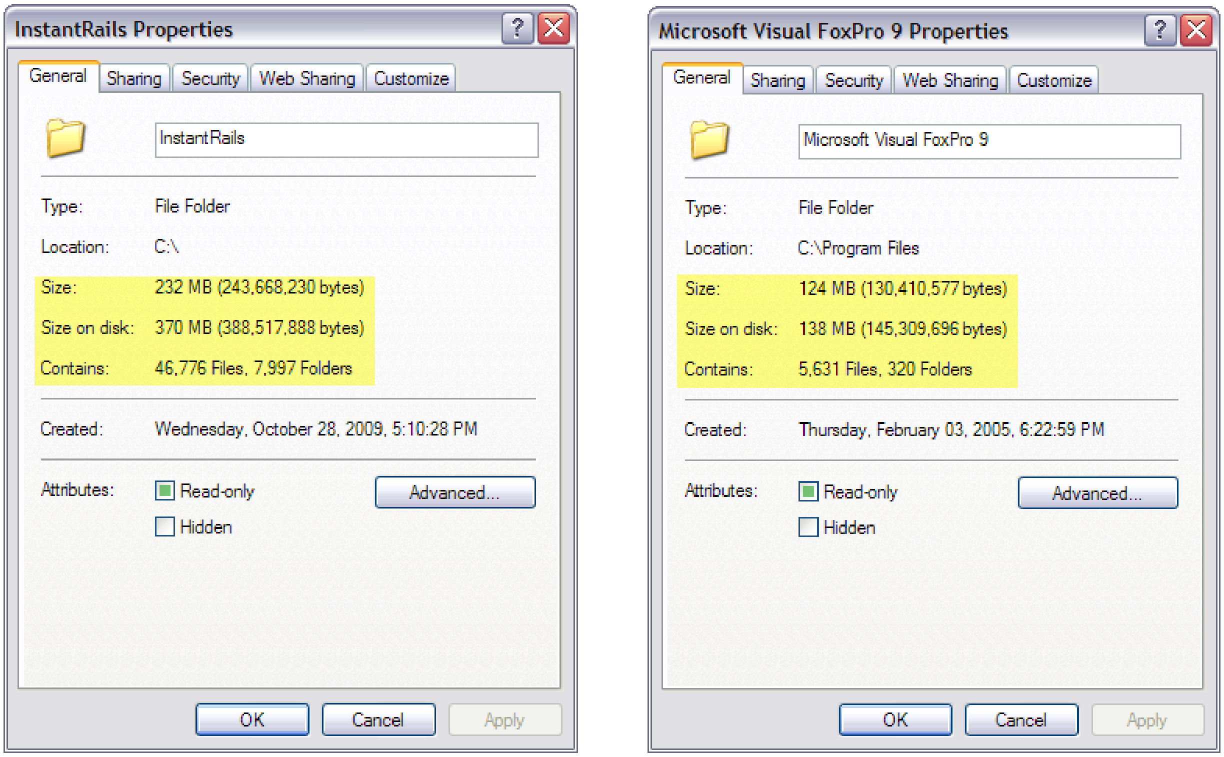
Instant Rails can be found at http://rubyforge.org/projects/instantrails/. As the description on that Web page tells you, "Instant Rails is a one-stop Rails runtime solution containing Ruby, Rails, Apache, and MySQL, all pre-configured and ready to run. No installer, you simply drop it into the directory of your choice and run it. It does not modify your system environment."

Instant Rails is a good choice for getting started with Ruby and Rails because you can download a single zip file, install it on any folder on your computer, play and learn, and uninstall (if desired) without making any modifications to your Windows system environment — kind of like the old days when you dropped a DOS app in a folder and were good to go.

At the time of this writing the current release of Instant Rails is v2.0. Although it's still a perfectly good choice for getting started and learning, you should be aware that this package dates from late 2007 and doesn't contain the latest versions of Ruby and other components. Instant Rails ships with Ruby 1.8.6; although this is not the latest version of the language, it is a stable release that's still widely used.

The Instant Rails download page is http://rubyforge.org/frs/?group_id=904. Locate the link for the InstantRails-2.0-win.zip file, download it, and extract the contents into a directory on your machine. C:\InstantRails is a good choice. I recommend you also read the release notes, which you can find as a separate link.

Regardless of whether you install InstantRails or use the RubyInstaller and install Rails separately, the installation comprises many more files than you're probably used to when working with tools like Visual FoxPro and Visual Studio. The Ruby and Rails footprint on your hard drive isn't really all that big but the components are very granular, so be prepared for a lot of files!



InstantRails Properties

InstantRails

Type: File Folder
Location: C:\
Size: 232 MB (243,668,230 bytes)
Size on disk: 370 MB (388,517,888 bytes)
Contains: 46,776 Files, 7,997 Folders
Created: Wednesday, October 28, 2009, 5:10:28 PM
Attributes: ☑ Read-only
□ Hidden

Microsoft Visual FoxPro 9 Properties

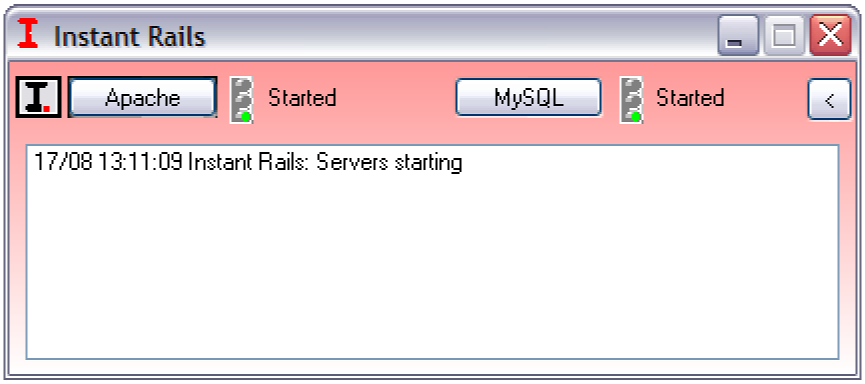
Microsoft Visual FoxPro 9

Type: File Folder
Location: C:\Program Files
Size: 124 MB (130,410,577 bytes)
Size on disk: 138 MB (145,309,696 bytes)
Contains: 5,631 Files, 320 Folders
Created: Thursday, February 03, 2005, 6:22:59 PM
Attributes: ☑ Read-only
□ Hidden

Figure 1: Ruby, Rails, and their related components comprise a large number of files compared to Visual FoxPro. The InstantRails footprint on your hard drive, although larger than VFP 9.0, is not nearly as large as Visual Studio. The RubyInstaller footprint is smaller than InstantRails, primarily because it does not include MySQL and Apache.

The Instant Rails executable file is InstantRails.exe. Before running it for the first time, take time to read the readme.txt file found in the root folder of your installation.

Running InstantRails.exe opens a small console window from which you can start and stop the Apache Web server and the MySQL database, and perform other functions.



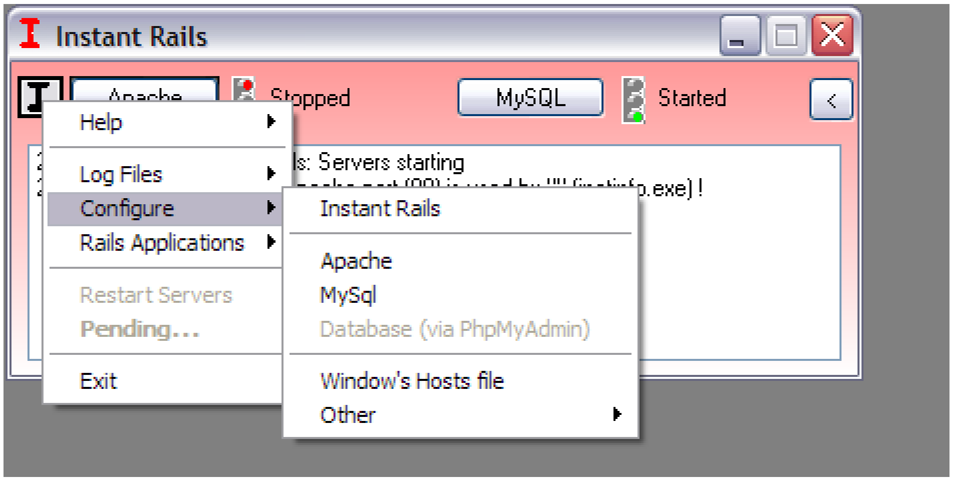
Apache Started
MySQL Started

```
17/08 13:11:09 Instant Rails: Servers starting
```

Figure 2: The Instant Rails console shows an activity log and enables you to start and stop Apache and MySQL,

If you have Microsoft Internet Information Service (IIS) installed and running on your machine, you'll probably want to stop the IIS service while running Apache. Otherwise you'll get a message telling you that Apache port 80 is being used by inetinfo.exe.

Clicking the InstantRails icon on the console opens a menu from which you can configure InstantRails, launch the Interactive Ruby command window, work with Rails applications you've developed, and more.



```
ls: Servers starting
.exe) !
```

Figure 3: Click the InstantRails icon in the console window to open a menu from which you can perform configuration and other tasks.

While you do need a Web server and a database in order to work with Ruby on Rails, you don't specifically need Apache and MySQL. These are the two you get if you install Instant Rails, but if you install the Ruby language and the Rails framework separately (as explained in the following section) you will not need to install Apache or MySQL. The default Web server for development work in Rails is now WEBrick or Mongrel, depending on which version you're working with, and the default database is now SQLite. Information on these tools is included later in this paper.

Installing with RubyInstaller

The alternative to InstantRails is to use the RubyInstaller to install the Ruby language and then install Rails and the other components separately. Although it's a bit more work, you end up with a somewhat more flexible installation, plus you don't need to install Apache and MySQL unless you really want them.

An older version of the Ruby Installer, formerly known as the OneClick Installer, can be downloaded from http://rubyforge.org/projects/rubyinstaller. The newer installer packages are available from http://rubyinstaller.org/. Whichever one you choose, your first decision will be which version of Ruby to install.

Ruby versions

There are two versions of the language in use today, Ruby 1.8 and Ruby 1.9. Within version 1.8 the most current release is 1.8.7, although 1.8.6 is still available. Ruby 1.9 is available as 1.9.1 and also the just-released 1.9.2. Ruby 1.9 includes some significant changes over 1.8; it's the latest and greatest, but in some cases code written for Ruby version 1.8 may behave differently in version 1.9.

So, which version of Ruby should you choose?

Although older, Ruby 1.8.6 and 1.8.7 are still the most widely used versions and therefore a good choice for starters. The current production release of Rails, version 2.3, recommends Ruby version 1.8.7 so that's another reason it's probably your best choice at this time.

Ruby 1.9 was released circa 2009 so it's not really all that new, but Rails 2.3 doesn't support it. Rails version 3, which is in beta/release candidate status at the time of this writing, is compatible with Ruby 1.9.2 but also with Ruby 1.8.7. If you're going to bypass Rails 2.3 and jump straight into Rails 3.0 then you probably want Ruby 1.9.2. Otherwise Ruby 1.8.7 is your best choice.

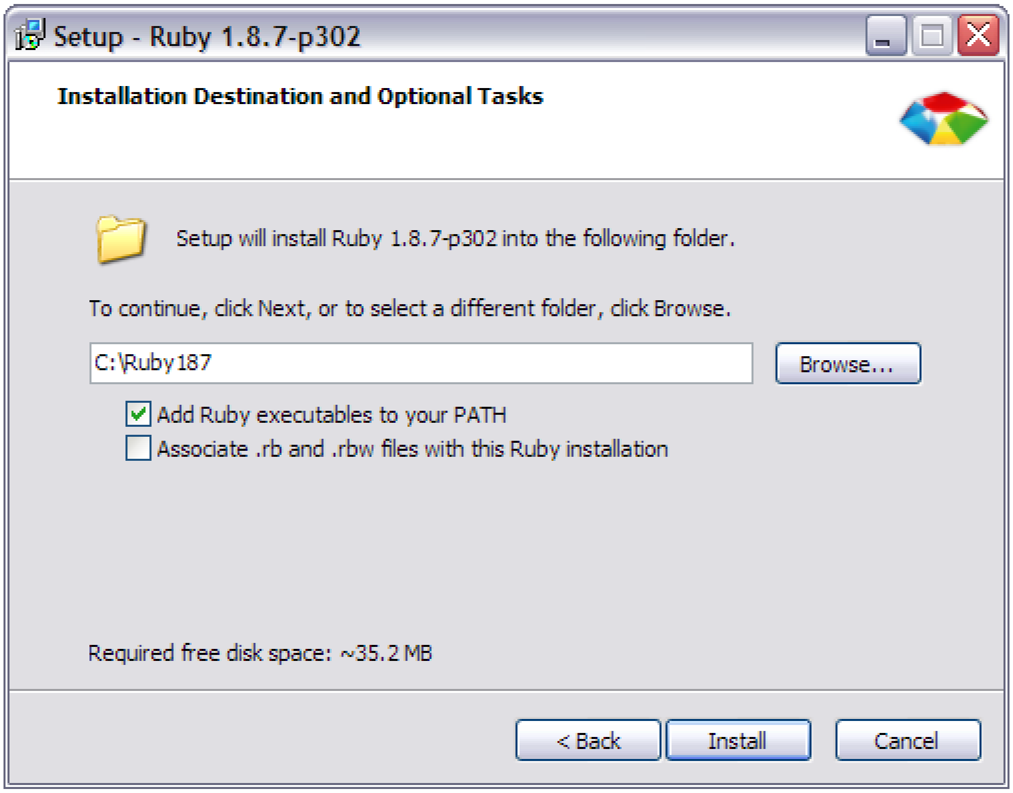
Installing Ruby 1.8.7

To install Ruby 1.8.7, go to the RubyInstaller website at www.rubyinstaller.org and download the Ruby 1.8.7-p302 package. This is an executable installer so all you need to do is run it to install Ruby.[2]

Running rubyinstaller_1.8.7-p302.exe launches the Ruby setup wizard.[3] The default installation folder is C:\Ruby187, which enables installation of this version of Ruby to coexist with other versions installed in different folders. I recommend accepting the default destination folder and also marking the check box to add it to your path. If this is the only version of Ruby on your machine you may also want to mark the second check box to associate .rb and .rbw files with this installation of Ruby, but this is not strictly necessary.

[2] I recommend the executable installer, but if you prefer a zip file you can download the 7-zip package instead of the exe. If you don't already have a zip utility that can handle the 7-zip format, go get the free 7-zip software from the link provided on the RubyInstaller download page.

[3] If you think the style of the setup wizard dialog in Figure 4 looks familiar, you're right: the RubyInstaller is built using Inno Setup.



Setup - Ruby 1.8.7-p302

Installation Destination and Optional Tasks

Setup will install Ruby 1.8.7-p302 into the following folder.

To continue, click Next, or to select a different folder, click Browse.

C:\Ruby187

☑ Add Ruby executables to your PATH

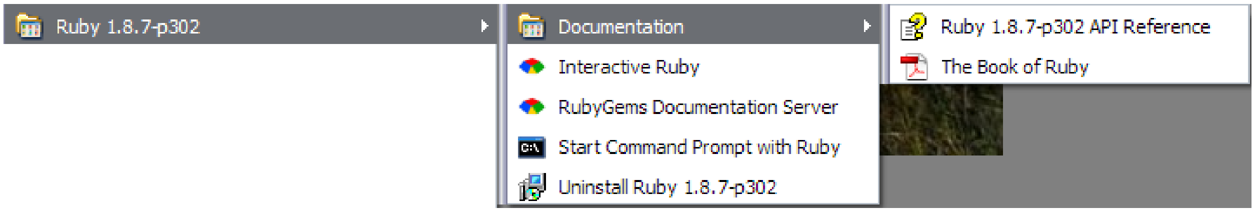
☐ Associate .rb and .rbw files with this Ruby installation

Required free disk space: ~35.2 MB

Figure 4: The default destination folder is C:\Ruby187, enabling Ruby v1.8.7 to coexist with other versions installed in different folders.

The RubyInstaller installs the Ruby language interpreter and the Ruby core library on your computer. If you marked the check box, the installer adds C:\Ruby187\bin to your system path, which enables you to run the Ruby executable (ruby.exe) from the command line without changing to the Ruby187 folder. On my Windows XP machine the installer placed this string at the beginning of the path; on my Vista machine it was placed after the %SystemRoot% paths. Either way, you may want to edit your system path and move the Ruby path string to wherever you want it to be based on your particular system configuration and preferences.

The installer also creates a Ruby187 item on your Start menu. This contains shortcuts to launch Interactive Ruby and to open a command window at the Ruby installation folder, among other things. You typically won't need to use these shortcuts to work with Ruby, although the shortcuts to the documentation are useful if you want to refer to it.



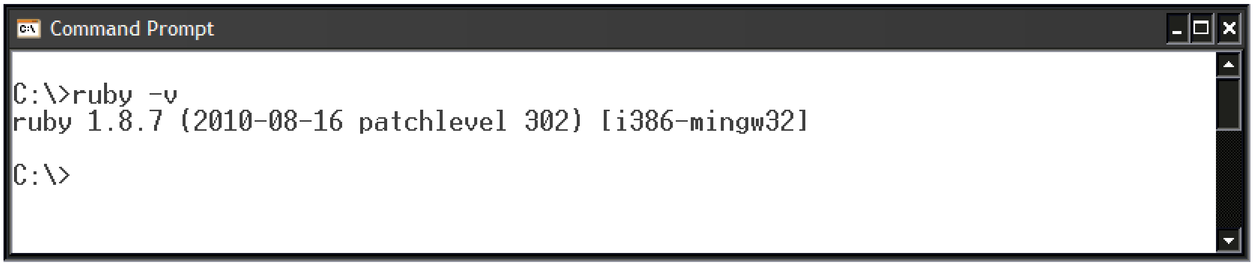
Ruby 1.8.7-p302
Documentation
Interactive Ruby
RubyGems Documentation Server
Start Command Prompt with Ruby
Uninstall Ruby 1.8.7-p302
Ruby 1.8.7-p302 API Reference
The Book of Ruby

Figure 5: The Start menu for Ruby 1.8.7 provides shortcuts to things you'll use when working with Ruby, including some documentation.

To ensure that your installation was successful, open a command prompt and type

```
C:\> ruby -v
```

This runs ruby.exe and instructs it to show its version number. You should see version 1.8.7 along with the patch level, as shown below.



```
C:\>ruby -v
ruby 1.8.7 (2010-08-16 patchlevel 302) [i386-mingw32]

C:\>
```

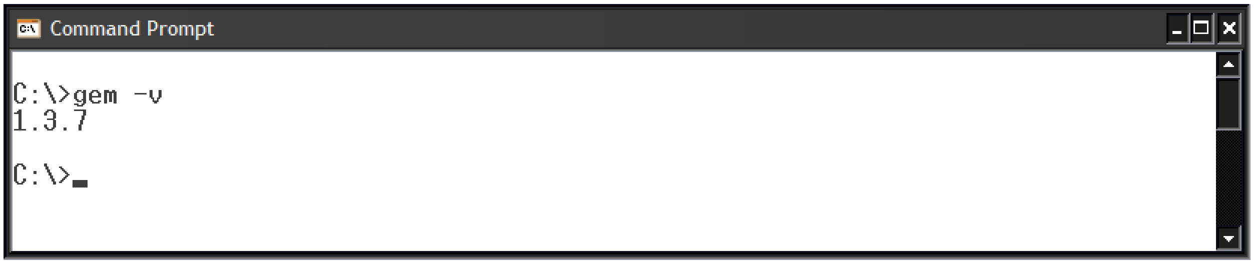
RubyGems

The Ruby 1.8.7 installer also installs RubyGems, a package manager for Ruby libraries that you'll use to install other components including Rails.

You can type

```
C:\> gem -v
```

at the command prompt to check the version of RubyGems that's currently installed.



```
C:\>gem -v
1.3.7

C:\>
```

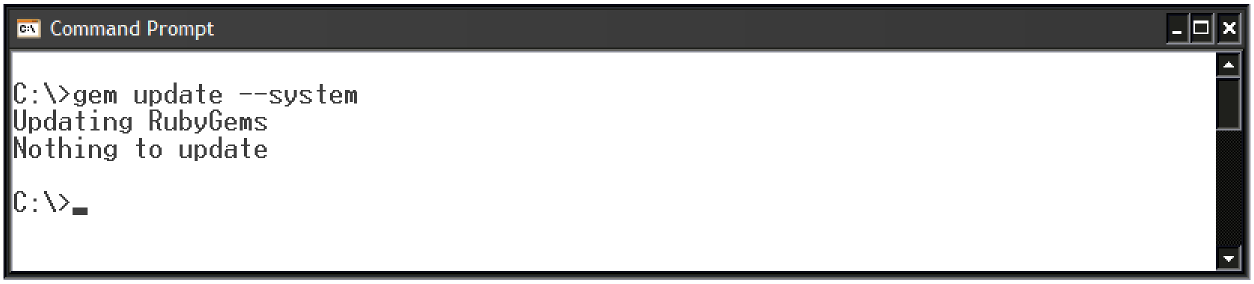
Figure 6: RubyGems version 1.3.7 is installed.

The version of RubyGems may not be the most recent one. To see if an update is available, type the following at the command prompt.

```
C:\> gem update --system
```

(Note the double hyphen in --system.)

RubyGems checks the gem server to see if a newer version is available. You'll need an Internet connection for this to work. If your version of RubyGems is current, you'll see a message to that effect.



```
C:\>gem update --system
Updating RubyGems
Nothing to update

C:\>_
```

Figure 7: RubyGems 1.3.7 is current so no update is required.

Some older versions of RubyGems have a bug that prevents automatic updates. If you get the Nothing to update message in response to 'gem update –system', you can do the following.[4]

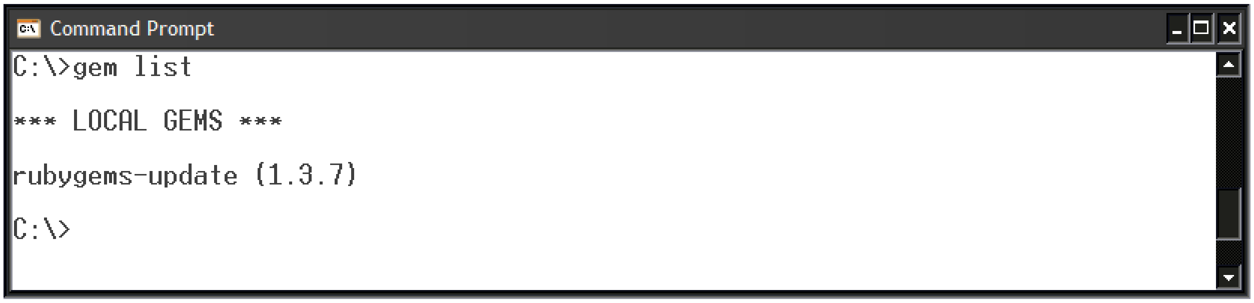
```
C:\> gem install rubygems-update
```

and then

```
C:\> update_rubygems
```

This will take a little time to run as your system downloads files from the gem server (http://rubygems.org) and installs them on your system. There is no visible indication that this is happening, other than that the command prompt has not returned in the command window. When the command prompt reappears, the update is done. Running the gem list command now shows that the rubygems-update gem is installed.

```
C:\> gem list
```

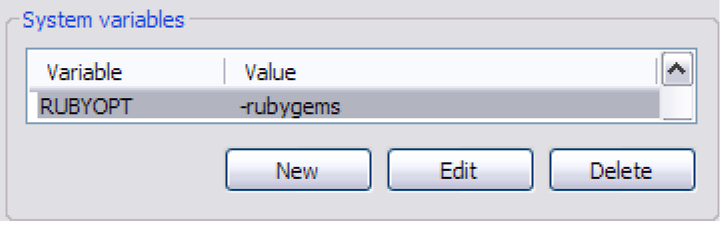


```
C:\>gem list

*** LOCAL GEMS ***

rubygems-update (1.3.7)

C:\>
```

[4] If gem –v shows you're already running RubyGems version 1.3.7 it probably isn't necessary to do this, but in my experience it doesn't hurt anything.

Figure 8: The gem list command now shows the rubygems-update version 1.3.7 is installed on the local machine.

To ensure that the Ruby interpreter loads the rubygems module whenever it starts up, add a RUBYOPT parameter to your system environment variables and give it a value of -rubygems. This value is then passed as a parameter to ruby.exe (the interpreter) whenever it's run. Note that this is only necessary in Ruby 1.8; it is not necessary in Ruby 1.9.



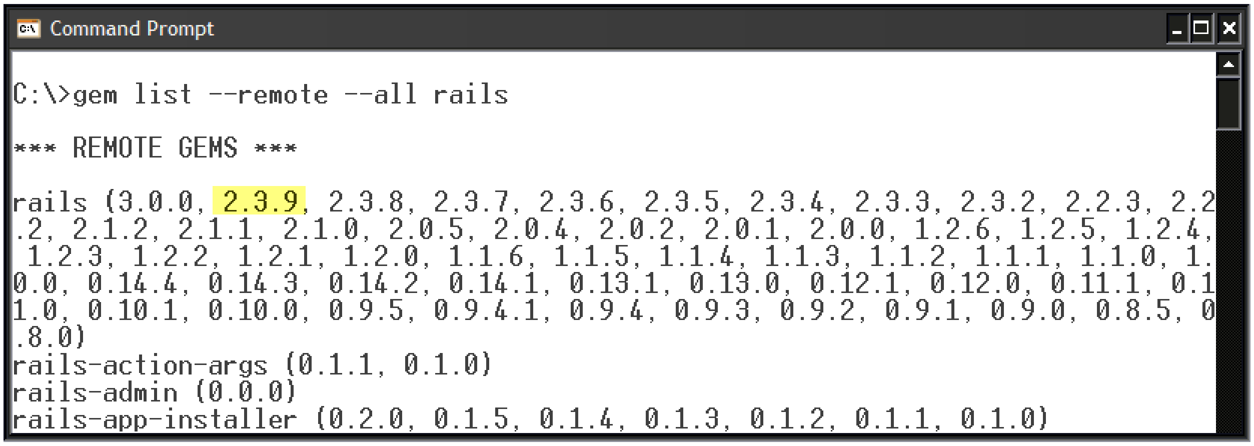
Variable	Value
RUBYOPT	-rubygems

Figure 9: Add a system variable named RUBYOPT with a value of -rubygems. The -rubygems parameter is automatically added to the ruby.exe command to ensure that Ruby loads the RubyGems module whenever it starts up.

You'll use the gem command again to install Rails and the other necessary components. To find out which gems are available on the remote server, run the gem list command with the --remote parameter followed by the generic name of the gems you’re interested in. For example, to find out what versions of Rails are available on the remote server, type

```
C:\> gem list --remote --all rails
```

This returns a long list of Rails-related gems. The one we're interested in is the latest version of Rails 2.0, which at the time of this writing was version 2.3.9.[5]



```
C:\>gem list --remote --all rails

*** REMOTE GEMS ***

rails (3.0.0, 2.3.9, 2.3.8, 2.3.7, 2.3.6, 2.3.5, 2.3.4, 2.3.3, 2.3.2, 2.2.3, 2.2
.2, 2.1.2, 2.1.1, 2.1.0, 2.0.5, 2.0.4, 2.0.2, 2.0.1, 2.0.0, 1.2.6, 1.2.5, 1.2.4,
 1.2.3, 1.2.2, 1.2.1, 1.2.0, 1.1.6, 1.1.5, 1.1.4, 1.1.3, 1.1.2, 1.1.1, 1.1.0, 1.
0.0, 0.14.4, 0.14.3, 0.14.2, 0.14.1, 0.13.1, 0.13.0, 0.12.1, 0.12.0, 0.11.1, 0.1
1.0, 0.10.1, 0.10.0, 0.9.5, 0.9.4.1, 0.9.4, 0.9.3, 0.9.2, 0.9.1, 0.9.0, 0.8.5, 0
.8.0)
rails-action-args (0.1.1, 0.1.0)
rails-admin (0.0.0)
rails-app-installer (0.2.0, 0.1.5, 0.1.4, 0.1.3, 0.1.2, 0.1.1, 0.1.0)
```

[5] You’ll notice that Rails 3.0.0 is now listed! Rails 3.0 is the very latest version of Rails and has some significant differences from Rails 2.0. Rails 3.0 was released in late August 2010, which was too late to be included in this paper. Anyway, learning Rails 2.0 will give you a huge jump start towards learning Rails 3.0 if that’s the direction you decide to go.

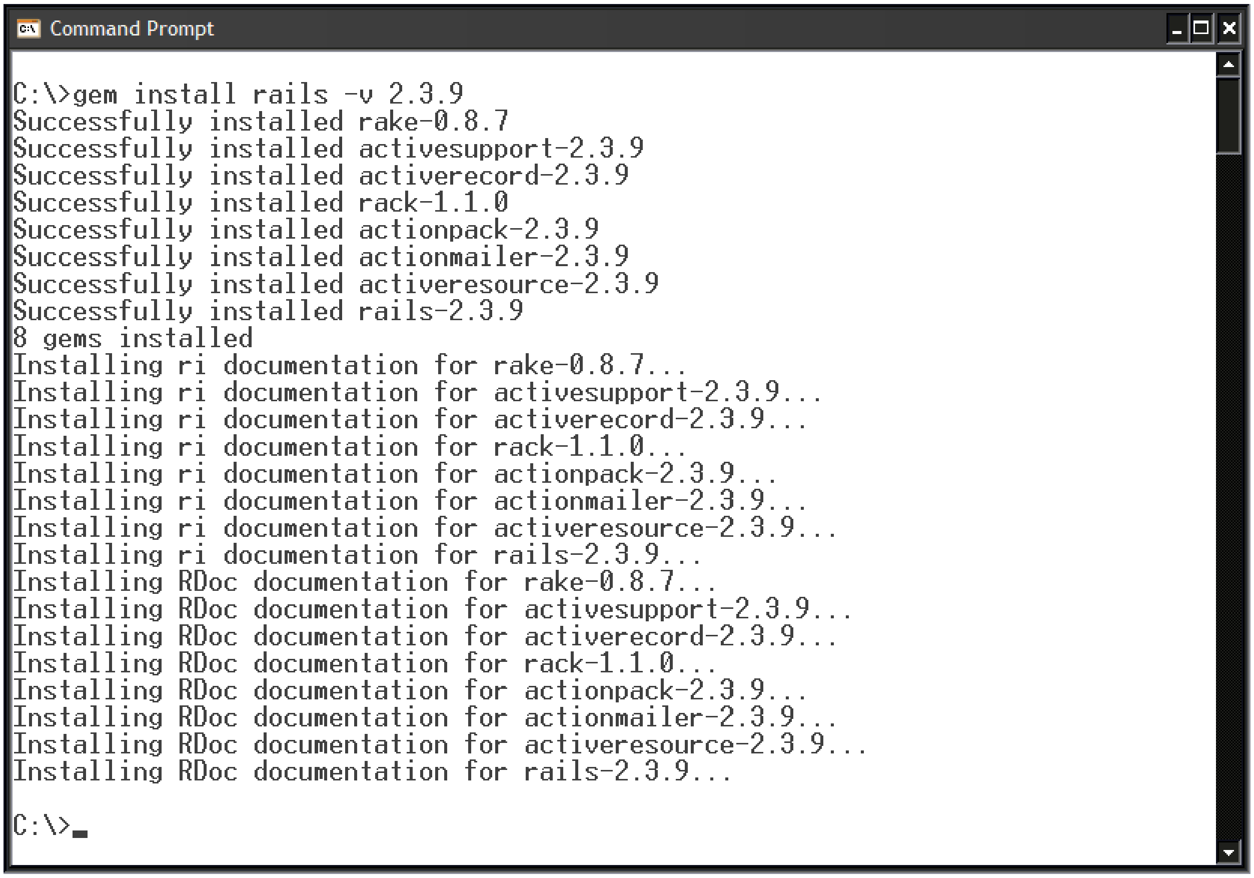
Figure 10: Use the gem list command with the --remote parameter to list gems on the remote server.

Installing Rails

You can install Rails by using gem to download and install the package from the remote gem server. The –v parameter is included because we want to install a specific version.

```
C:\> gem install rails –v 2.3.9
```

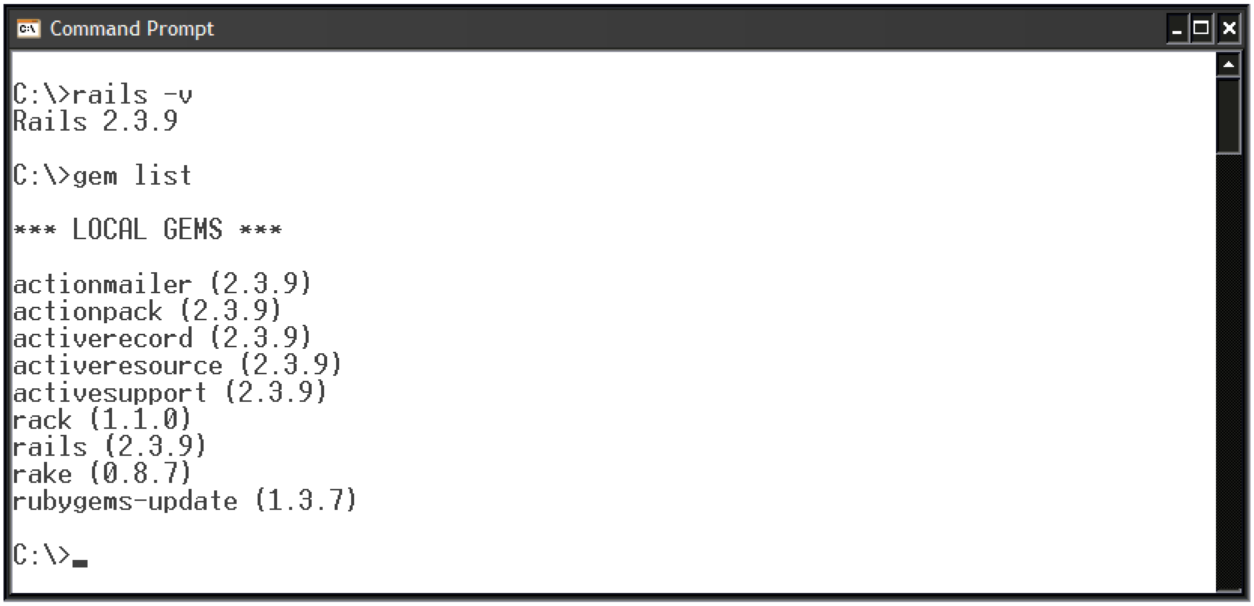
This will take a little while as the required files are downloaded from the gem server and installed on your machine. As before, there is no continuous visible progress indicator. Just be patient and wait for the results to begin to appear in the command window (see Figure 11). When the command prompt to returns, the installation is done.



```
C:\>gem install rails -v 2.3.9
Successfully installed rake-0.8.7
Successfully installed activesupport-2.3.9
Successfully installed activerecord-2.3.9
Successfully installed rack-1.1.0
Successfully installed actionpack-2.3.9
Successfully installed actionmailer-2.3.9
Successfully installed activeresource-2.3.9
Successfully installed rails-2.3.9
8 gems installed
Installing ri documentation for rake-0.8.7...
Installing ri documentation for activesupport-2.3.9...
Installing ri documentation for activerecord-2.3.9...
Installing ri documentation for rack-1.1.0...
Installing ri documentation for actionpack-2.3.9...
Installing ri documentation for actionmailer-2.3.9...
Installing ri documentation for activeresource-2.3.9...
Installing ri documentation for rails-2.3.9...
Installing RDoc documentation for rake-0.8.7...
Installing RDoc documentation for activesupport-2.3.9...
Installing RDoc documentation for activerecord-2.3.9...
Installing RDoc documentation for rack-1.1.0...
Installing RDoc documentation for actionpack-2.3.9...
Installing RDoc documentation for actionmailer-2.3.9...
Installing RDoc documentation for activeresource-2.3.9...
Installing RDoc documentation for rails-2.3.9...

C:\>_
```

Figure 11: Rails installs a number of gems and their associated documentation.

To confirm successful installation, run rails –v from the command line to query the installed version of Rails. Running gem list again now shows you the new gems that have been installed, including rake, which we'll meet again later in the section about Rails.

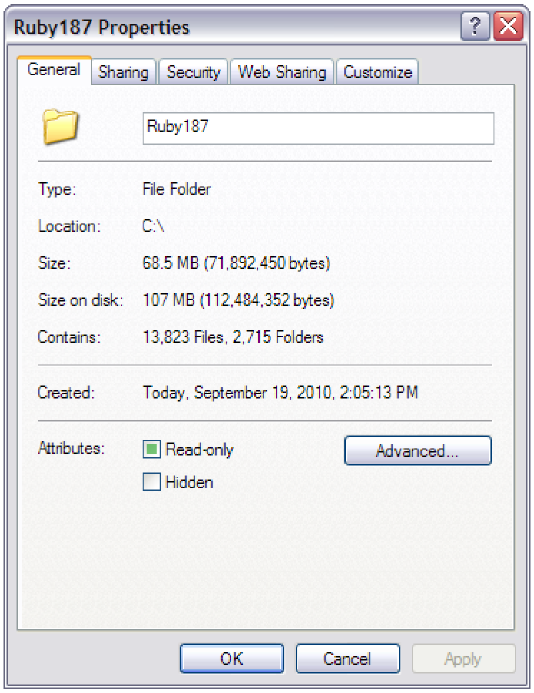


```
C:\>rails -v
Rails 2.3.9

C:\>gem list

*** LOCAL GEMS ***

actionmailer (2.3.9)
actionpack (2.3.9)
activerecord (2.3.9)
activeresource (2.3.9)
activesupport (2.3.9)
rack (1.1.0)
rails (2.3.9)
rake (0.8.7)
rubygems-update (1.3.7)

C:\>_
```

Figure 12: Rails version 2.3.9 is installed and the local gems are listed.

If you're wondering where all the files these gems installed are located, they're in C:\Ruby187\lib\ruby\gems\1.8 and its subfolders.

The RubyInstaller installation is not nearly as large as InstantRails. Figure 13 shows the Ruby187 folder's footprint on my hard drive after installing everything up to this point.



Ruby187

Type: File Folder

Location: C:\

Size: 68.5 MB (71,892,450 bytes)

Size on disk: 107 MB (112,484,352 bytes)

Contains: 13,823 Files, 2,715 Folders

Created: Today, September 19, 2010, 2:05:13 PM

Attributes: ▣ Read-only

□ Hidden

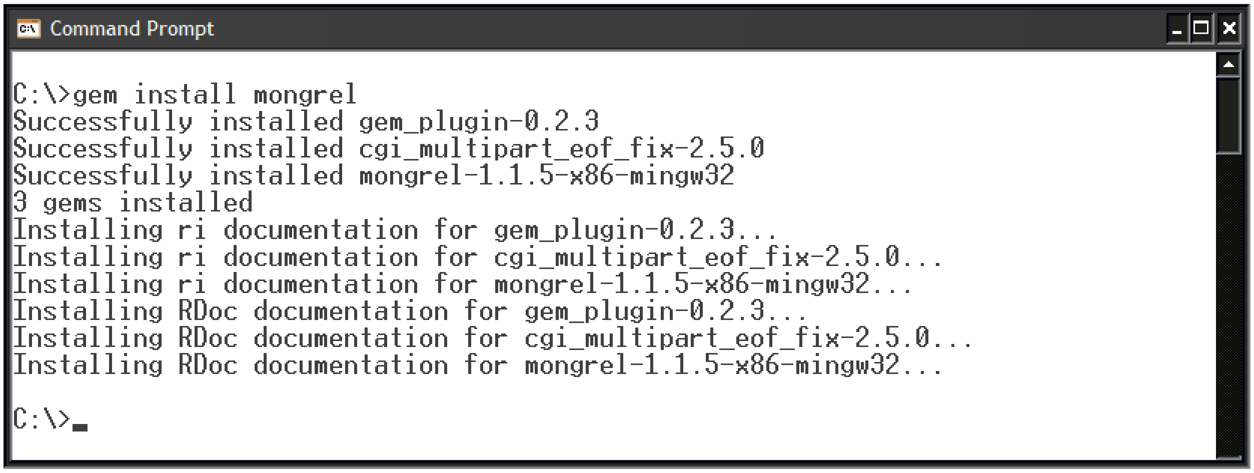
Figure 13: After installing Ruby 1.8.7 with the RubyInstaller and also installing Rails, the Ruby187 folder contains several thousand files but its footprint on disk is still a modest 107MB.

Installing Mongrel

Rails installs a small Web server called WEBrick. It's suitable for learning and for testing small Rails applications, but it's not very fast. Mongrel is a faster, better alternative that can be easily install using gem. Type

```
C:\> gem install mongrel
```

to download and install the latest version of Mongrel. As always, the results are displayed in the command window.



```
C:\>gem install mongrel
Successfully installed gem_plugin-0.2.3
Successfully installed cgi_multipart_eof_fix-2.5.0
Successfully installed mongrel-1.1.5-x86-mingw32
3 gems installed
Installing ri documentation for gem_plugin-0.2.3...
Installing ri documentation for cgi_multipart_eof_fix-2.5.0...
Installing ri documentation for mongrel-1.1.5-x86-mingw32...
Installing RDoc documentation for gem_plugin-0.2.3...
Installing RDoc documentation for cgi_multipart_eof_fix-2.5.0...
Installing RDoc documentation for mongrel-1.1.5-x86-mingw32...

C:\>_
```

Figure 14: Run gem install mongrel to install the Mongrel Web server and its dependencies.

Mongrel now supersedes WEBrick as the default Web server for Rails applications on your local machine.

Installing SQLite

Rails is a framework for developing data-based Web applications, so you're going to want a database to work with. MySQL used to be the standard for learning and testing, but these days it's been dropped in favor of SQLite.

Unlike the previous components we've been installing, SQLite is not packaged as a gem. However, it's very easy to install. Go to the SQLite download page at http://www.sqlite.org/download.html and locate the section entitled Precompiled Binaries for Windows. The file names incorporate the version number, which at the time of this writing is version 3.7.2. You'll want to download the current version, whatever that may be at the time you go get it.

Download the file containing the DLL for accessing SQLite databases. For version 3.7.2 the file name is sqlitedll-3_7_2.zip. Extract the DLL and place it in a location along your system path. Since you're going to be using SQLite with Rails, you can place the DLL in C:\Ruby187\bin.

You should also download the zip file described as "A command line program for accessing and modifying SQLite databases." For version 3.2.7 the file name is sqlite-3_7_2.zip. This zip file contains sqlite.exe, which you can also place in C:\Ruby187\bin.

Finally, you'll need to install the Ruby bindings so Rails knows how to access SQLite databases. This is done by installing the sqlite3-ruby gem.

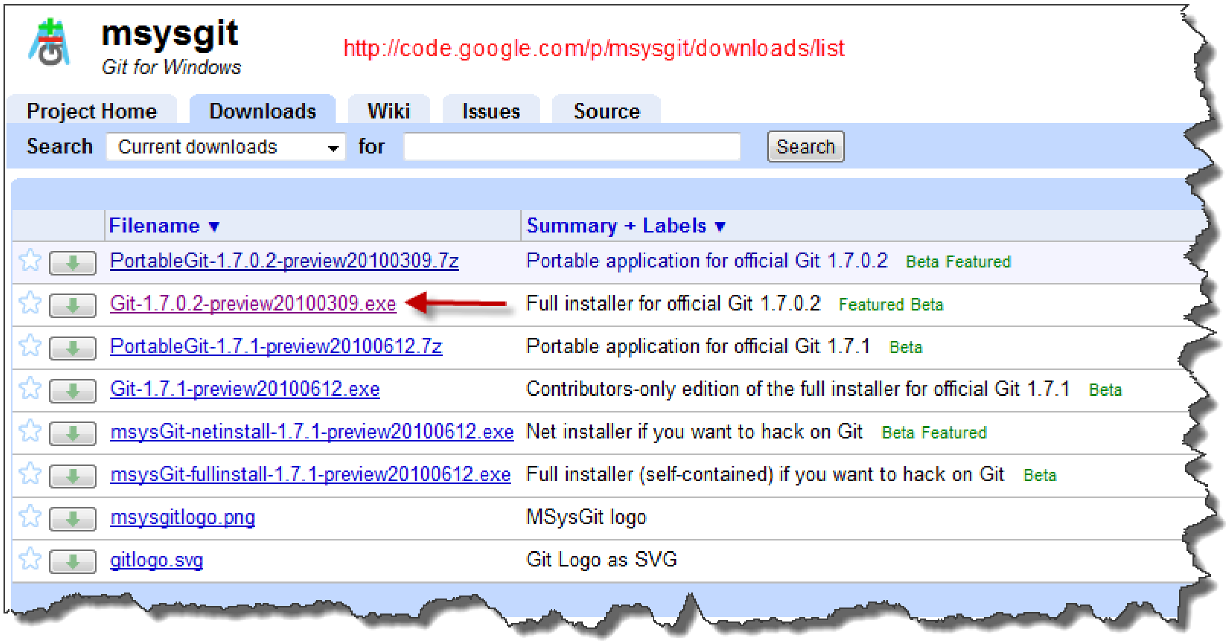
```
C:\> gem install sqlite3-ruby
```

Don't overlook the '3' in the name of the gem. You installed version 3 of SQLite, so you want the gem for SQLite3 as well.

Side Note: If you use Firefox for Web development, there's a terrific Add-on called SQLite Manager that provides a great visual interface to SQLite databases. I'd encourage you install it and give it a try.

Installing Git

Git is a distributed version control system. It's widely used in the Ruby and Rails communities as well as many others, especially in the world of open source. You can download files from Git repositories without actually installing Git on your computer, so installing a Git client on your computer is optional. However, if you do want to install it you can download the current version from the msysgit website at http://code.google.com/p/msysgit/downloads/list.



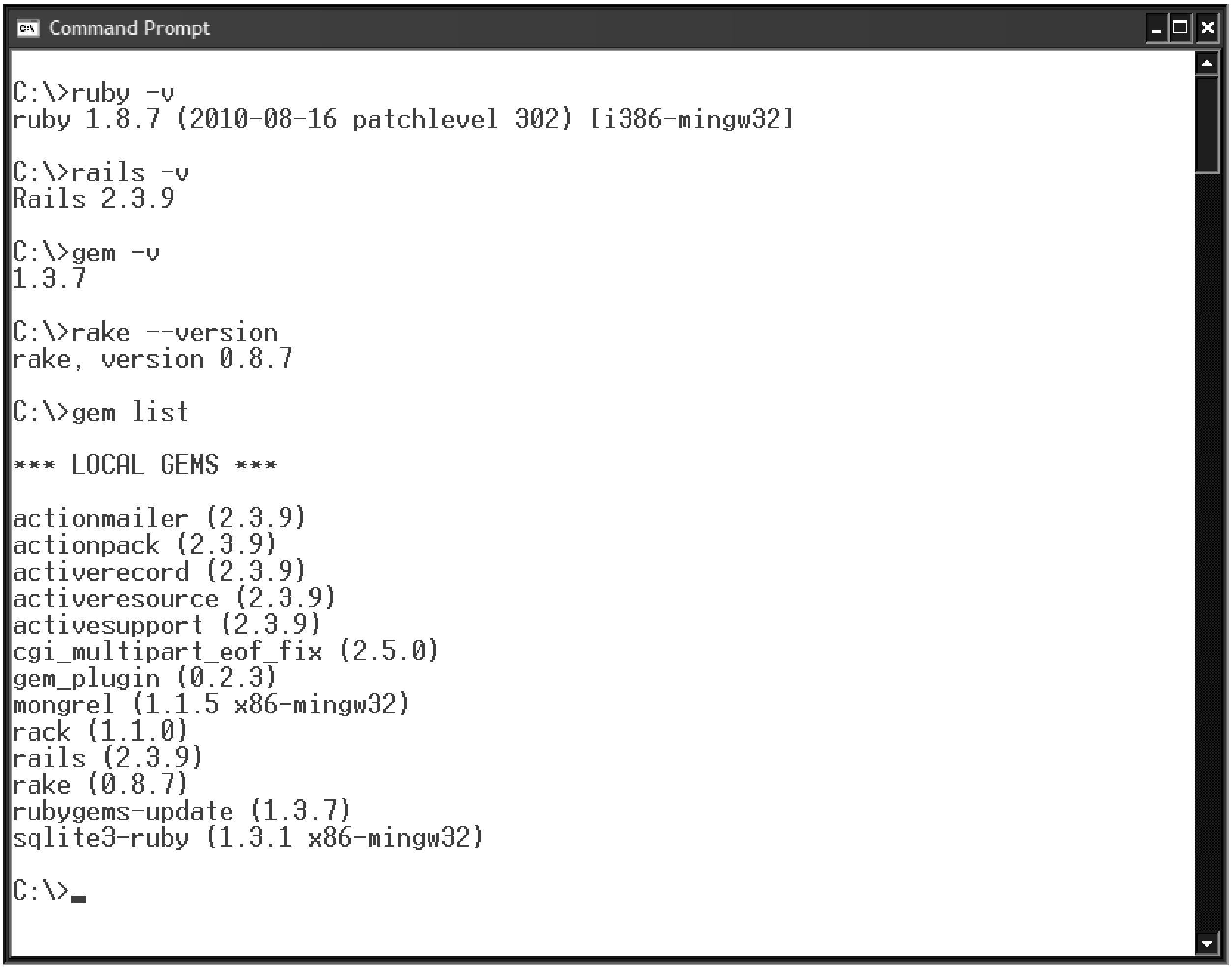
Filename ▾	Summary + Labels ▾
PortableGit-1.7.0.2-preview20100309.7z	Portable application for official Git 1.7.0.2 Beta Featured
Git-1.7.0.2-preview20100309.exe	Full installer for official Git 1.7.0.2 Featured Beta
PortableGit-1.7.1-preview20100612.7z	Portable application for official Git 1.7.1 Beta
Git-1.7.1-preview20100612.exe	Contributors-only edition of the full installer for official Git 1.7.1 Beta
msysGit-netinstall-1.7.1-preview20100612.exe	Net installer if you want to hack on Git Beta Featured
msysGit-fullinstall-1.7.1-preview20100612.exe	Full installer (self-contained) if you want to hack on Git Beta
msysgitlogo.png	MSysGit logo
gitlogo.svg	Git Logo as SVG

Figure 15: Download Git from the msysgit website.

Video instructions for Windows can be found at http://book.git-scm.com/6_git_on_windows.html. The presenter runs through things rather quickly for a novice to follow, but really about all you have to do is launch the installer and accept the defaults.

The finished installation

Once you've got everything installed you're all set to start working with the Ruby language and the Rails framework. If you're ever in doubt about what's installed or what versions you're running, remember that you can run most of the component executables with the –v or --version parameter, and you can run gem list to see which gems are installed. Figure 16 summarizes the installation on one of my machines.



```
C:\>ruby -v
ruby 1.8.7 (2010-08-16 patchlevel 302) [i386-mingw32]

C:\>rails -v
Rails 2.3.9

C:\>gem -v
1.3.7

C:\>rake --version
rake, version 0.8.7

C:\>gem list

*** LOCAL GEMS ***

actionmailer (2.3.9)
actionpack (2.3.9)
activerecord (2.3.9)
activeresource (2.3.9)
activesupport (2.3.9)
cgi_multipart_eof_fix (2.5.0)
gem_plugin (0.2.3)
mongrel (1.1.5 x86-mingw32)
rack (1.1.0)
rails (2.3.9)
rake (0.8.7)
rubygems-update (1.3.7)
sqlite3-ruby (1.3.1 x86-mingw32)

C:\>_
```

Figure 16: Most components accept the –v or --version parameter to tell you what version is installed. The gem list command shows the local gems on your computer.

Optional system configuration changes

There are a couple of changes you may want to make to your system configuration to make things more convenient when working with Ruby.

Customize the command window

When you're working with Ruby you're going to be spending a fair amount of time in the command window, especially when using the Interactive Ruby (IRB) console, so you might want to configure its size and appearance the way you want it.

To configure the command window, right-click on its title bar and open the Properties sheet. When "Raster Fonts" is selected, which it is by default, Windows gives you a choice of several window sizes and shapes. If none of these is satisfactory and you want more control over the font size, select the Lucida Console font instead of Raster Fonts and then select the desired font size.[6] This is especially useful when you want to use a larger than normal font. For example, I set it to a 24-pt font when making presentations.

When you save the changes you have a choice of making them permanent or only for the current instance of the command window.

Add Ruby to your path

If you installed InstantRails, no changes were made to your system environment. While this gives you a nice clean install, it also means that whenever you want to invoke Ruby you need to CD to C:\InstantRails\ruby\bin, which is where ruby.exe is located along with other files you'll want to be able to easily reference. Working with Ruby will be a lot more convenient for you if you add C:\InstantRails\ruby\bin to your system path; that way you'll have access to ruby.exe regardless of the current folder.

If you installed Ruby using the RubyInstaller, the installation folder was automatically added to your system path if you marked the check box during setup, as discussed above.

Easily clear the IRB console

When working with the Interactive Ruby command window, a.k.a. the IRB console, you'll probably encounter times when you want to clear the screen. Unfortunately, on a Windows machine you can't simply type cls like you can in the command window. You can clear the IRB console by entering system('cls'), but who wants to type all that?

[6] Thanks to the person who showed me this trick at a previous conference. I was seated way down the table from where you were so I don’t know who you are, but thank you.

The solution is to create a .irbrc file in your %USERPROFILE% folder and define a method to handle the cls command.[7] Note that Windows won't let you directly create a file whose name starts with a period, so first create it as foo.irbrc and then use the command window to rename it to just .irbrc.

```
C:\> REN foo.irbrc .irbrc
```

Once your .irbrc file is created, use Notepad or another text editor to enter the following:

```
def cls
  System('cls')
end
```

Now you can type just cls in the IRB console to clear the screen. If you'd prefer something other than cls, such as clear like we're used to using in Visual FoxPro, simply change the method definition line to from def cls to def clear. [8]

And now for something completely different

Before we get started learning about the Ruby language and the Rails framework, I want to encourage you to get ready for a mental paradigm shift. As an object-oriented language, many aspects of Ruby will be familiar to experienced Visual FoxPro developers, but many things are materially different. On top of that, you won't be developing desktop apps in Rails – this is all about developing applications for the Web.

This quote from p. 37 of Practices of an Agile Developer is apropos here:

> "It can help if you take care to transition completely to the new environment as much as possible… Write a completely different kind of application from the kind you usually write. Don't use your old language tools at all while you're transitioning. **It's easier to form new associations and new habits when you have less baggage from the old habits lying around.**"
> (emphasis mine)

It may not be possible to put Visual FoxPro or any of your other day to day development tools away while you learn Ruby and Rails—it certainly wasn't that way for me—but, insofar as possible, try to clear your mind of what you already know and prepare yourself to begin doing software development the Ruby and Rails way.

[7] The %USERPROFILE% folder is \Documents and Settings\<username> on Windows XP, or \Users\<username> on Windows Vista and Windows 7.

[8] Just as in Visual FoxPro or any other language, stay away from reserved words when naming things of your own. You can find a list of Ruby's reserved words at http://www.ruby-doc.org/docs/keywords/1.9/ and a list of Rails reserved words at http://wiki.rubyonrails.org/rails/pages/reservedwords.

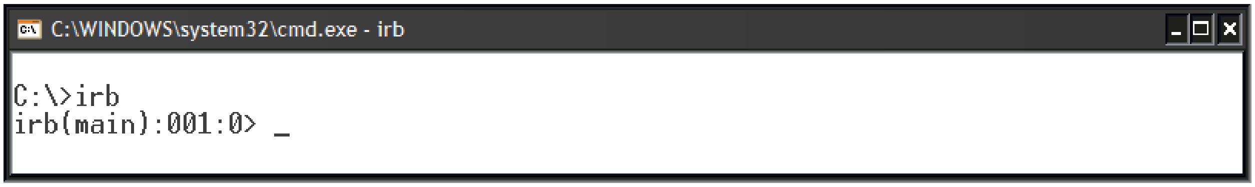
Working with Ruby on Windows

Interactive Ruby

The easiest way to get started with Ruby is to use the Interactive Ruby shell, called irb for short. It's automatically installed with Ruby so you don't need to do anything else to get it. To fire up Interactive Ruby, simply open a command prompt and type irb.

```
C:\> irb
```

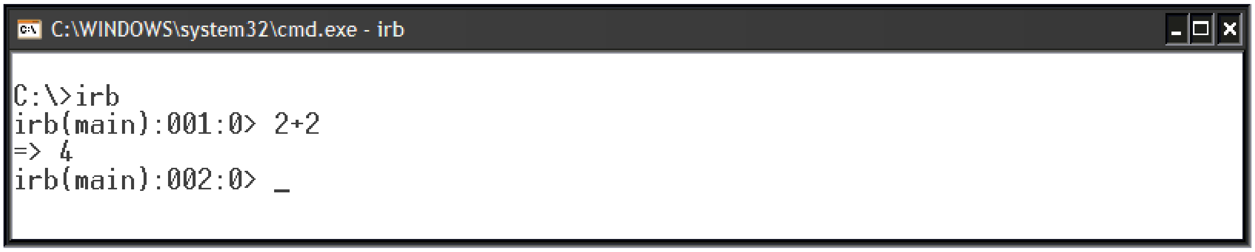
The interactive Ruby shell opens in the same command window and displays the irb prompt.



```
C:\>irb
irb(main):001:0> _
```

Figure 17: The interactive Ruby (irb) command prompt.

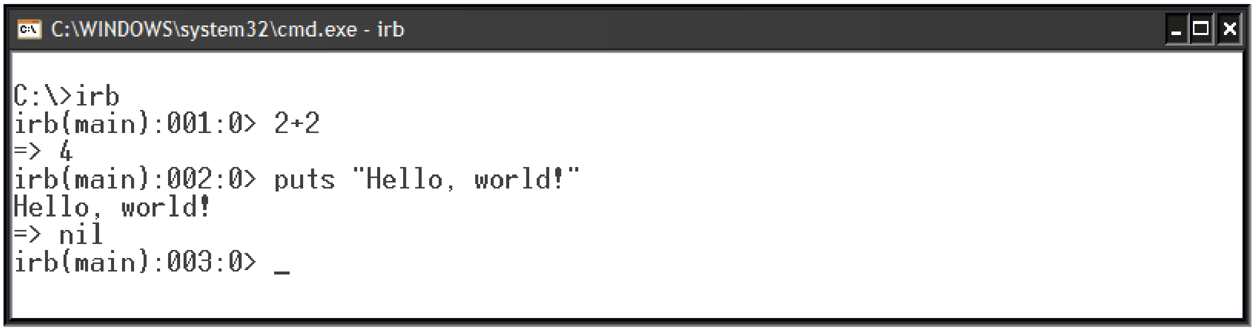
Anything you type at the irb command prompt is sent to the Ruby interpreter and the result is displayed below the command(s). Like the command window in Visual FoxPro, the interactive Ruby shell is a great way to experiment with individual statements and small chunks of code.



```
C:\>irb
irb(main):001:0> 2+2
=> 4
irb(main):002:0> _
```

Figure 18: Anything you enter in the interactive Ruby shell is processed by the Ruby interpreter and the result is displayed.

The puts function writes a string to the console, so it's a good way to display output in irb. Figure 19 shows the obligatory "Hello, world!" example.

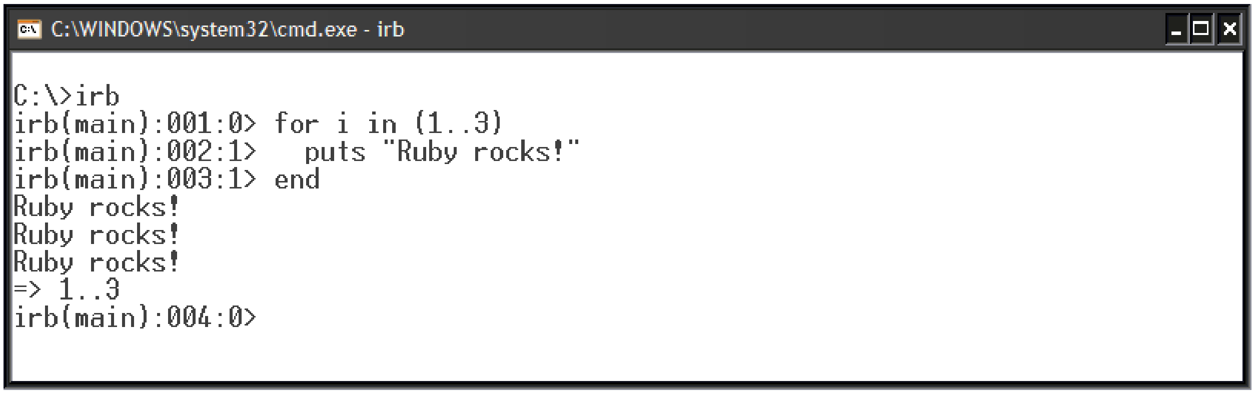


```
C:\>irb
irb(main):001:0> 2+2
=> 4
irb(main):002:0> puts "Hello, world!"
Hello, world!
=> nil
irb(main):003:0> _
```

Figure 19: The puts function writes a string to the irb console.

Notice that there are two lines of output, the string "Hello, world!" that we asked Ruby to display and also a nil value (nil is Ruby's version of null). That's because in Ruby, everything returns a value. When you're working in interactive Ruby the return value is always displayed in addition to any output generated by the statement(s) you entered.

You can enter multi-line commands in irb, too. For example, if you want to run a three-line for loop you can press the Enter key after each line and irb knows not to start interpreting the code until you have entered the end statement.



```
C:\>irb
irb(main):001:0> for i in (1..3)
irb(main):002:1>   puts "Ruby rocks!"
irb(main):003:1> end
Ruby rocks!
Ruby rocks!
Ruby rocks!
=> 1..3
irb(main):004:0>
```

Figure 20: You can run multi-line code blocks in interactive Ruby, too.

No doubt you noticed the syntax of a for loop in Ruby isn't quite the same as in Visual FoxPro or even other languages like JavaScript, C#, or C++. More on that later.

To exit from interactive Ruby, simply type quit.

Stand-alone editors

Ruby program files are just plain text files with a ".rb" file name extension, so you can use any text editor to work with them. This includes the lowly Notepad as well as any other free or commercial text editor.

To make development work easier, however, you'll want a text editor that can be integrated with the Ruby interpreter so you can run your code without leaving the editor. Some but by no means all of them are listed below.

- SciTE (free) – www.scintilla.org/SciTE.html
- EditPlus ($35) – www.editplus.com
- Komodo Edit (free) and Komodo IDE ($295) – www.activestate.com/komodo

When I first started working with Ruby I installed the SciTE text editor. It's a good editor and it's free, plus it has an output window to show results.

I've used EditPlus (as well as TextPad) for many years for general text editing purposes. Both have free plug-ins for various languages including Visual FoxPro and Ruby, which provide nice features like syntax coloring and in some cases auto-completion. However, I quickly settled on EditPlus for my Ruby work when I discovered it can easily be made to integrate with the Ruby interpreter so you can run your code and see your results without leaving the editor.

Naturally, any text editor that works for Ruby also works fine for HTML and other types of text files you'll be working with in Ruby on Rails.

TextMate appears to be the editor of choice for Ruby developers who work on a Mac. There is a similar editor for Windows called the e text editor (that's not a typo – the name of the editor is 'e'), but to get its full power you'll also need to install cygwin, a Linux emulator for Windows. Without cygwin installed you don't get the integration between the e editor and the Ruby interpreter that you'll want for development work.

IDE-style editors

There is no native integrated development environment (IDE) for either Ruby or for Rails, but it's not necessary to use one. IDEs provide a certain amount of power and convenience, but they do so at a cost: they require a learning curve of their own, they consume additional resources on your machine, they can be slow, and many of them are written in Java, which may require additional extensions to be installed on Windows machines.

If want to investigate IDEs, here is a list of some of the popular ones.

- Aptana Studio (free) – www.aptana.org
- RadRails (free) – www.radrails.org [a plug-in for Aptana Studio]
- Rubymine ($99) – www.jetbrains.com/ruby

The Ruby programming language

Ruby has a clean, easily understandable syntax that makes it easy to learn and use. Visual FoxPro developers seeing Ruby code for the first time will immediately notice some of the similarities. For example, Ruby statements are terminated with a line break and do not need to end with a semicolon.

Similarities to and differences from Visual FoxPro

One of the biggest differences is that Ruby is case sensitive while Visual FoxPro is not. Besides keeping variable names straight, the importance of case sensitivity will also become apparent as you read the section on Naming Conventions later in this paper.

Like Visual FoxPro, Ruby is an object-oriented language, although to an even greater extent than VFP. However, like Visual FoxPro, you can also write procedural code in Ruby.

Ruby uses some different terminology than Visual FoxPro. When talking about classes and objects in VFP we refer to methods and properties. In Ruby, a method is still a method but classes have class variables and objects have instance variables.

In Ruby, everything is an object! This enables some interesting coding conventions that will at first seem foreign to VFP developers; for example, you can call a method on an integer literal.

In Ruby, assignment statements use the = operator. However, the equality comparison operator is ==.

In Ruby, not only is everything an object but everything is also an expression, meaning that it returns a value. This enables chaining statements together, much in the same way we can do in VFP. For example, in VFP `a = 1 = 1` returns true, and in Ruby `a = 1 == 1` also returns true.

Unlike Visual FoxPro, Ruby programs are interpreted rather than compiled. The interpreter installed by InstantRails and the Ruby Installer is the one written by Ruby's founder Yukihiro Matsumoto ("Matz") and his team. There are other Ruby interpreters, too, including JRuby and Microsoft's IronRuby. Collectively and individually, these variations are sometimes referred to as Ruby implementations.

Ruby is a dynamic language, meaning that its behavior can be modified at runtime. For example, classes in Ruby are open and can be modified at runtime. This means you can do something like the following (although it's not recommended!):

Listing 1: Ruby classes can be modified at runtime. This example is taken from http://www.ruby-lang.org/en/documentation/ruby-from-other-languages/.

```
class Fixnum
   def +( foo)
      self - foo
   end
end
puts 2 + 2  # returns zero!
```

This code dynamically alters the behavior of the '+' method on the Fixnum class so it subtracts instead of adds. Fixnum is Ruby's integer class. The hash mark (#) indicates the start of a comment, which in Ruby can either be inline or at the beginning of a line.

Like Visual FoxPro, Ruby also uses dynamic variable typing (a.k.a. "duck typing"[9]), meaning that a variable can change its type at runtime simply by having a different type of value assigned to it.

Like VFP, Ruby supports many data types including strings, integers, floating-point numbers, and date and time values. Ruby also supports arrays, but in addition it supports ranges and hashes, structures which have no direct counterpart in VFP.

As in VFP, in Ruby strings can be delimited by single quotes or double quotes. However, in Ruby there a slight difference between the two. Ruby does not support the [ ] delimiters for strings; in Ruby, square brackets have another meaning.

Conditional statements, loop structures, methods, and other Ruby language structures are terminated with an end statement. Unlike VFP, however, in Ruby it's always just end, not endif, enddo, or end plus anything else.

In Ruby, indentation is used to show the hierarchical structure of code. As in VFP, indentation is optional.

Ruby supports the relational operators +, -, *, and / for addition, subtraction, multiplication, and division. Unlike VFP, Ruby also supports assignment operators like +=, *=, etc. In Visual FoxPro, these are simply IntelliSense shortcuts that the editor expand to the full syntax.

In Ruby, statements do not have to end with a semicolon. However, semicolons are used to separate multiple statements on the same line. Ruby's line continuation character is the backslash.

```
message = 'Now is the time ' \
          'for all good programmers ' \
          'to start learning Ruby and Rails.'
puts message  # a one-line message
```

In Ruby as in VFP, there is often more than one way to write functionally equivalent code. Most of Ruby’s basic syntax and code structures will look familiar to Visual FoxPro developers, but code in Ruby is often written using syntactical structures that don’t have direct parallels in VFP. This is sometimes referred to as writing code “the Ruby way”. You'll see examples of this as we go along.

Ruby naming conventions

The case-sensitive nature of Ruby is reflected in its naming conventions, most of which are enforced by the interpreter.

[9] If it walks like a duck and quacks like a duck, then it's a duck.

Local variable names begin with a lowercase letter. An underscore is used to separate words. Camel case should not used for variable names. These are widely used conventions but are not enforced.

```
my_var # is a local variable name
```

Global variable names must begin with a dollar sign and then follow the same conventions as for local variable names.

```
$my_var # is a global variable name
```

Instance variables are what we would call properties in Visual FoxPro. Instance variable names begin with an ampersand and then follow the same conventions as for local variable names.

```
@my_var # is an instance variable name
```

Class variables have no direct parallel in Visual FoxPro. Class variable names begin with a double ampersand and then follow the same conventions as for local variable names.

```
@@my_var # is a class variable name
```

Names for constant values are written in all uppercase.

```
PI # is a constant value
```

Class names and module names begin with an uppercase letter. Words are delineated using camelCase instead of underscores.

```
MyClassName # is a class name
```

Method names follow the same naming convention as local variable names, i.e., they're written in lowercase and words are separated by underscores. There are two special cases for method names. Methods that return a Boolean value (true or false) are by convention named with a trailing question mark, while methods that can alter the object on which they are called have names with a trailing exclamation point to indicate "use with care".

```
save # is a method name
do_something # is a method name
registered? # indicates a method that returns a Boolean value
register! # indicates a method that alters the object on which it is called
```

Ruby data types and objects

Strings

Ruby string literals are enclosed in either single quotes or double quotes. Single quoted string literals are rendered verbatim, while double quoted string literals support several backslash escape sequences. Ruby also performs variable substitution (formally called "string interpolation") on double-quoted string literals if the string contains an expression written in string interpolation format, which begins with a hash mark and is enclosed in curly braces.

```
animal = "duck"
puts "Daffy is a #{animal}"  # double-quoted string, variable substitution occurs
=> Daffy is a duck
puts 'Daffy is a #{animal}'  # single-quoted string, no variable substitution
=> Daffy is a #{animal}
```

Numeric values

Numeric literals can be integers or floating point numbers. Integers are instances of either the Fixnum class or, if additional precision requires it, the Bignum class. Floating point numbers are instances of the Float class. Any of these types can, of course, can also be the result of an expression.

Integer literals are written using just numerals with no decimal point. Remembering that in Ruby everything is an object, we can call the class method on a numeric literal to see what class it is.

```
1.class  # returns Fixnum
```

We can also call the class method on a numeric variable to see what class it is.

```
a = 1
a.class # returns Fixnum
b = 2**10
b.class # a kilobyte is still a Fixnum
c = 2**20
c.class # so is a megabyte
d = 2**30
d.class # but a gigabyte is a Bignum
```

Floating point numbers are written with at least one numeral to the left of the decimal, even if it's zero, plus a decimal point plus some number of numerals to the right of the decimal point.

```
a = 1.0
a.class # returns Float
```

When performing arithmetic operations where you want a floating point number as the result, be sure to include at least one floating point value in the expression.

```
a = 3 / 2   # returns 1, a is class Fixnum
b = 3.0 / 2 # returns 1.5, b is class Float
c = 3 / 2.0 # returns 1.5, c is class Float
```

Arrays

Arrays in Ruby are much like arrays in Visual FoxPro. In Ruby an array is written as a series of comma-separated values enclosed in square brackets, which of course is why square brackets cannot be used to enclose string literals as in VFP. Unlike VFP, Ruby arrays are zero-based.

```
a = [1,2,3] # define an array
a.class     # returns Array
a[0]        # returns 1
```

In Ruby, you can also refer to elements of an array starting from the last one.

```
a[-1]       # returns 3, the last element in the array
```

Hashes

A hash is Ruby is a structure that contains a set of objects called keys, where each key has an associated value. Ruby hashes are similar to collection objects in VFP.

A hash literal is enclosed in curly braces. The => operator is used to associate a value with its corresponding key. Multiple key/value pairs are separated by commas. Hashes are sometimes referred to as associative arrays, and like arrays their values are referenced using the index value enclosed in square brackets.

```
a = {1 => "a", 2 => "b"}
puts a.class # returns Hash
puts a[1]    # returns a
puts a[2]    # returns b
```

The preceding example used a numeric value as the hash key. Remember that everything in Ruby is an object, so in this example the hash keys are objects of class Fixnum and the associated values are strings.

We could also flip things around and use string objects with numeric values.

```
a = {"a" => 1, "b" => 2}
puts a.class # returns Hash
puts a["a"]  # returns 1
puts a["b"]  # returns 2
```

It's very common in Ruby programming to use a symbol as the hash key. Symbols, which are discussed in a following section, are written as an identifier prefixed with a colon. Using symbols, the first example can be written as:

```
b = { :first => "a", :second => "b"}
puts b[:first]  # returns a
puts b[:second] # returns b
```

Hashes are very commonly used in Ruby on Rails, so it's helpful to become comfortable with how they're defined and how they're used.

Ranges

A range object in Ruby defines a series with a starting value and an ending value. A range literal is enclosed in parentheses, with either two or three dots (period characters) separating the starting value from the ending value. Two dots means an inclusive range where both the starting and ending values are included, while three dots means an exclusive range where the ending value is not included.

Range objects typically have numeric values but can also have string values, as illustrated in the following examples.

```
a = (1..9) # defines a range from 1 to 9, inclusive
b = ("a"..."z") # defines a range from "a" through "y" ("z" is excluded)
```

Ranges have many uses in Ruby. One is for iteration. Here’s an example of using a range object to iterate in a for loop:

```
for x in (1..3)
  puts x
end
```

returns

```
1
2
3
```

Range objects are enumerable, meaning you can call iterator methods such as each on them. We’ll discuss iterators separately in the next section.

Another use for range objects is to test a value for membership in the range. This is roughly equivalent to Visual FoxPro’s BETWEEN( ) function. As an example, we can create a range that spans the dates for Southwest Fox 2010.

```
swfox2010 = (Time.local(2010,10,14)..Time.local(2010,10,18))
```

Then we can ask if the current date falls between the conference starting and ending dates.

```
swfox2010.member? Time.now
```

You'll remember that a question mark can be used to indicate a method that returns a Boolean (true/false) value. The member? method on the Range object is an example. At the time I'm writing this the statement returns false because I'm running it before the conference has started, but it will return true when I demo the code during the pre-con session.

Symbols

Internally, the Ruby interpreter maintains a symbol table that contains the names of all the classes, methods, and variables it's managing at any given time. The interpreter references the entries in the symbol table by their position, which is faster than doing a string comparison on their name.

Entries in the Ruby symbol table are available to a Ruby program, too. A program can reference a symbol by using a Symbol object. For convenience, Symbol objects are commonly referred to as just symbols.

A symbol literal is written as an identifier preceded by a colon. We saw an example of this in the section on hashes, where we used :first and :last as symbol literals. Symbol literals can be converted to strings, and vice versa, using methods built in to Ruby.

A key to understanding symbols is that two strings with the same value, for example "abc", can be different objects but will have the same symbol in the symbol table. We can see this from the following example:

```
string1 = "abc"   # a string object with value "abc"
string1.object_id # a numeric value identifying this string object, e.g., 24103356
string2 = "abc"   # a different string object with the same value
string2.object_id # a different numeric value than string1.object_id, e.g., 24060888

a = string1.to_sym    # returns :abc, which is a symbol
a.object_id # returns a numeric value identifying this object, e.g., 362516
b = string2.to_sym    # also returns :abc
b.object_id # returns the same value as string1.object_id because it's the same
object
```

In the literature you'll frequently find symbols defined as "the thing whose name is". This seems to me to be as good a way as any of getting your mind around what a symbol represents.

Symbols and hashes are frequently used as a way of passing parameters in Ruby.

Ruby operators

Ruby operators are very similar to those in Visual FoxPro. Table 1 lists some of the most commonly used Ruby operators and their corresponding operations.

Table 1 : The most commonly used Ruby operators and their operations.

Operator(s)	Operation(s)
+, -, *, /	Addition, subtraction, multiplication, division
**	Exponentiation
%	Modulus
==, !=	Equality, inequality
>, >=	Greater than, greater than or equal to
<, <=	Less than, less than or equal to
&&	Boolean AND
||	Boolean OR
?:	Ternary (three operands) conditional
+	String concatenation

There are two main differences between Ruby operators and Visual FoxPro operators that the VFP developer should remain aware of while reading or writing Ruby code.

- In Ruby, a single equals sign (=) is an assignment statement while a double equals sign (==) is an equality comparison operator.
- In Ruby, the Boolean operators are && and || instead of the words 'and' and 'or'. Ruby also supports the words 'and' and 'or' as Boolean operators, but they differ slightly from && and || in their implementation and are therefore less frequently used.[10]

The strange looking conditional operator '?: 'is explained in the next section.

Another difference to watch out for is that Ruby uses the hash mark (#) to begin a comment, while in Visual FoxPro # can be used as an inequality operator, meaning 'not equal to'. On the flip side, a FoxPro inline comment begins with a double ampersand (&&), which in Ruby is the Boolean AND operator.

Ruby conditionals

Ruby implements the traditional if…else…end syntax in the conventional manner. Parentheses around the condition expression are not required but are optional, as is the word then after the condition. The word end is required.

[10] For a good explanation of the difference between && and 'and', see http://www.themomorohoax.com/2008/12/08/and-vs-amperand-in-ruby.

```
if 1 == 1
  puts 'true'
else
  puts 'false'
end
```

This also works:

```
if 1 == 1 then
  puts 'true'
else
  puts 'false'
end
```

The word then is required for an inline if statement.

```
if 1 == 1 then puts 'true' else puts 'false' end
```

Multi-conditional if statements are implemented using the elsif syntax. Note the missing 'e' in elsif; this is (to me) an ugly exception to Ruby's otherwise elegant syntax.

```
# if...elsif...else
if 1 == 1
  puts 'true'
elsif 2 == 2
  puts 'also true'
else
  puts 'false'
end
```

In Ruby, you can also use if as a modifier, like this:

```
puts 'true' if 1 == 1
```

Ruby also support an unless statement, using the same structure as the if statement.

```
unless 1 == 2
   puts 'things are normal'
else
   puts 'things are weird'
end
```

Like if, unless can also be used as a modifier.

The ternary, or three operand, conditional ?: works just like the IIF function in Visual FoxPro.

```
puts 1 == 1? 'true' : 'false'
```

Finally, remember that everything in Ruby returns a value. This value can be assigned to a variable, as in the following example.

```
foo = if 1 == 1 then 'true' else 'false' end
puts foo
```

Ruby loop structures

Loop structures in Ruby are implemented with the while, until, and for statements. The while and until statements have the conventional behavior, the difference being that the while statement executes a block of code while a condition is true (i.e., until the condition becomes false), whereas the until statement executes a block of code until a condition becomes true. These two statements are widely used in Ruby, but it's the for loop that's probably the most familiar to Visual FoxPro developers.

Ruby's for loop has a slightly different syntax than you find in VFP or in other languages such as JavaScript, C#, and C++. The Ruby syntax is

```
for var in collection do
  some code
end
```

Var is the variable whose value is altered during execution of the loop. Collection is an enumerable object such as a range, hash, or array. In Ruby, do and end define a block of code. Whatever is after do and before its corresponding end is part of the same block. Inline code blocks are enclosed in curly braces.

Here's the example we saw earlier using a range object.

```
for x in (1..3)
  puts x
end
```

Here's a different example using an array:

```
countdown = [5,4,3,2,1,0]
for i in countdown
  puts i == 0? "Blast off!" : i
end
```

Did you recognize the use of the ?: operator in this example? This code returns

```
5
4
3
2
1
Blast off!
```

Note that the do keyword is optional in all three loop structures.

Ruby iterators

Some types of Ruby data objects are derived from the Enumerable class, which supports several iterator methods. Where available, iterators provide another way of writing loop structures.

Iterators are very commonly used in Ruby programs. Some of the iterators you'll see most often include times, each, upto, downto, and map. Using iterators like these in place of other longer or more cumbersome code structures is part of learning to write code "the Ruby way".

The Fixnum (integer) class supports the upto, downto, and times iterators methods. These can be used where otherwise you might have to write a for loop. For example, to display the a string expressing your enthusiasm for Ruby three times, you could write a for loop like this:

```
for i in 1..3
   puts "Ruby is cool."
end
```

However, you can use the times iterator to write the equivalent code "the Ruby way".

```
3.times {puts "Ruby rocks!"}
```

The first time I saw this syntax I thought, Holy Cow! What in the world is that all about?? Then I learned about iterators and the fact that you can call methods on any object, including literals, and quickly began to appreciate Ruby's beauty and power.

To illustrate the upto iterator method, the same code can be written as:

```
1.upto(3) { puts "Ruby rocks!"}
```

It's also possible to pass the iterated value to the code block using this syntax:

```
1.upto(3) { |n| puts "#{n}"}
```

In this example, each value from 1 to 3 is passed to the code block as the parameter 'n', whose value is then displayed using string interpolation as we've seen before. Running this code produces the expected result:

```
1
2
3
```

The each iterator method can be called on any object that qualifies as a collection, such as an array, a range, or a hash. This is illustrated in the following examples.

```
# Array
[1,2,3].each { |n| puts n}  # displays 1 2 3

# Range
(1..3).each  { |n| puts n}  # displays 1 2 3

# Hash
{1 => "one", 2 => "two", 3 => "three"}.each { |k,v| puts k, v}
# displays 1 one 2 two 3 three
```

Ruby program structure

As in any programming language, Ruby programs are formed by writing statements and using language structures such as the conditionals, loop structures, and iterators we've been looking at.

Earlier we said that although Ruby is a fully object-oriented language, you can still write what amounts to procedural code. The following is a complete, working Ruby program although there's not an object, class, or method in sight.

```
a = (1..3)
b = 2
for i in a
  puts i**b
end
```

Just to prove that Ruby can run this program outside of IRB or an editor, save this code as a Ruby program file, for example squares.rb, and run it from the Windows command prompt using

```
C:\> ruby squares.rb
```

Methods

Methods in Ruby are defined with the def keyword, followed by the name of the method and the parameters it accepts (if any) enclosed in parentheses. Methods terminate with the end keyword. The method code comprises everything following def up to its corresponding end.

As an example, here's a method to calculate the square of any number.

```
def square(n)
  n * n
end
```

(If you're wondering how this method can exist just hanging out in space by itself with no associated class, it's actually created as a method on a class that's not instantiated, namely the Object superclass.)

You can call this method in the conventional manner by invoking its name and passing in the desire value. In Ruby, a method returns the value of the last expression evaluated within the method code. Therefore a return statement is not necessary, although it is optional and can be used either for early exit from a method or simply as a matter of coding style if that's what you prefer.

```
puts square(5)  # returns 25
```

You can also write recursive methods in Ruby, as illustrated in the following example that calculates the factorial of a number.

```
def factorial(n)
    if n == 0
      1
    else
      n * factorial( n - 1)
    end
end
puts factorial(5) # returns 120
```

Classes

Of course, Ruby is a completely object oriented language so in actual practice Ruby programs are designed and built using classes, which in turn have methods and variables.

Classes in Ruby are defined with the class keyword. Class names begin with an uppercase letter and camelCase is employed if the class name involves more than one word. Like most other Ruby constructs, a class definition is terminated with the end keyword.

Here is an example of a Ballerina class with a method that returns what a ballerina wears.

```
class Ballerina
  def outfit
    "tu"*2
  end
end
```

In Ruby, an object is instantiated from a class by invoking the new method on the class name. As in Visual FoxPro and other languages, the resulting object reference is stored in a variable so the program can access the methods and variables on the object.

To use our Ballerina class we can write code like this:

```
ox = Ballerina.new
```

```
puts ox.outfit  # Note - parentheses are not needed since no parameters are passed
```

This returns the string "tutu", which is the outfit a ballerina wears.

In Ruby, classes can be subclassed via inheritance just as in other object oriented languages. The syntax for creating a subclass is the left angle bracket ("less than" sign). In the following example, class SubFoo is defined as a subclass of Foo.

```
Class SubFoo < Foo
```

In Ruby as in other object-oriented languages, subclasses inherit the behavior of their parent class. Naturally, the general intention of subclassing in any language is to modify the behavior of the parent class in order to implement increasing specialization as you move down the class hierarchy.

To illustrate how subclassing can be implemented in Ruby, let's start by defining a generic Bird class and giving it a fly method.

```
class Bird
  def fly
    puts "#{self.class} is flying"
  end
end
```

Note the reference to self in this code. In Ruby, the self keyword refers to the class being defined. It's equivalent to this in Visual FoxPro.

We don't intend to actually instantiate an object directly from the Bird class. Instead, we want to define some specific types of birds and specialize their flying behavior accordingly.

```
class Hummingbird < Bird
  def fly
    puts "#{self.class} darts to and fro"
  end
end
class Penquin < Bird
  def fly
    puts "#{self.class} prefers to swim"
  end
end
```

Now we can create an instance of both birds and see how they fly.

```
ox = Hummingbird.new
ox.fly  # "Hummingbird darts to and fro"
tux = Penquin.new
tux.fly  # "Penquin prefers to swim"
```

There's also a shorthand way of instantiating an object and calling a method on it in a single line. Instead of the last two lines above, we could write

```
Penquin.new.fly
```

which gives the same result.

Unlike Visual FoxPro, a Ruby class has a physical presence in runtime memory and its code is executed at runtime even if no objects have been instantiated from it. This gives rise to the concept of class variables, which have names beginning with @@. The runtime value of a class variable is available to the subclasses of the parent class, and can be referenced and manipulated in subclass code.

The following example illustrates the use of a class variable whose value is referenced and manipulated by subclasses. This example also incorporates concepts we've seen before, as well as introducing a few new ones that are explained below.

```
class Bird
  @@count = 0
  attr_accessor :name
  def initialize( name)
    @name = name
    self.add_one
  end
  def add_one
    @@count+=1
  end
  def how_many_other_birds
    @@count - 1
  end
end
```

Phew! What's going on here??

First of all, @@count is the class variable on the Bird class. Its purpose is to hold a value representing the number of birds that have been instantiated as subclasses of Bird.

In Ruby, the initialize method is automatically invoked to initialize new instances of a class. It's a special method that's basically equivalent to the Init( ) function in Visual FoxPro.

In this example, the initialize method accepts a parameter that is the name of the bird being created. When a Bird object is instantiated, the initialize method stores the name passed to it as the parameter in an instance variable called @name. It then calls its own add_one method, using a reference to self, to increment the value of the class variable @@count by one. Add_one is a method I created; it's not a special Ruby method like initialize.

The attr_accessor :name line is a shorthand way of creating what's called a reader method on the name variable. Notice the use of the Ruby symbol :name here. A reader method provides a way for code outside of an object to gain access to the value of an instance variable inside the object.

Finally, the Bird class definition includes a method named how_many_other_birds. This method enables each bird object to find out how many other birds it sees. The return value from this method is the value of the @@count class variable minus one. The reason for subtracting one is that the value of @@count includes the bird who's asking the question!

To exercise our code we can now create two birds, Tweety and Woody Woodpecker, and find out if they can see each other. Notice that this code uses string interpolation to get the name and count to display. Because the count in a numeric value, the Ruby to_s method is called to convert it to a string. You may remember that a backslash is the Ruby line continuation character.

```
tweety = Bird.new("Tweety")
puts "#{tweety.name} sees #{tweety.how_many_other_birds.to_s}" \
  " other bird"

woody = Bird.new("Woody Woodpecker")
puts "#{woody.name} sees #{woody.how_many_other_birds.to_s}" \
  " other bird"

puts "#{tweety.name} sees #{tweety.how_many_other_birds.to_s}" \
  " other bird"
```

Tweety is created first. He initially sees no other birds. Then Woody Woodpecker comes along. He can see Tweety so he reports that he can see one other bird. After Woody has arrived, Tweety now reports that he can see one other bird as well.

The actual output from this code is shown below.

```
Tweety sees 0 other bird
Woody Woodpecker sees 1 other bird
Tweety sees 1 other bird
```

I realize that was a lot to digest in one example, but I hope that by now you are getting comfortable enough with Ruby to start looking at more complete chunks of code like this one without needing to have each new concept explained via its own example.

Modules

In Ruby, modules are a higher order level of organization. Like a class, a module is a named collection of methods and variables. However, a module can contain classes in addition to methods and variables of its own. Unlike a class, modules cannot be instantiated and they have no hierarchy; in other words, you cannot subclass a module.

Modules serve several useful purposes. For one thing, modules define a namespace within which names are unique and isolated from identical names in other modules. For another, modules can be used as mixins. Mixins are Ruby's way of composing classes with behaviors from multiple sources without the complexities surrounding the use of actual multiple inheritance.

Module are defined using the module keyword. Like class names, module names begin with a capital letter. Our conference needs some music, so let's define a module that lets us sing and play instruments.

```
module Music
  def sing(song)
    case song
      when "la"
        "la la la"
      when "doremi"
        "do re mi fa so la te do"
      when "rain"
        "Singin' in the rain"
      else
        "Singin' at Southwest Fox"
    end
  end
  def play(instrument)
    case instrument
      when "tuba"
        "oompa"
      when "piano"
        "plunk"
      when "flute"
        "tweet"
      when "guitar"
        "strum"
      else
        "sound"
    end
  end
end
```

The two methods in the Music module also illustrate the use of the Ruby case statement, which was not covered earlier.

Now that we have a way of making music, let's write some code to do so. As a first example we append the following code at the bottom of the Music module definition above to make it a single program.

```
class Conference
  include Music  # The Conference class now has access to the sing and play methods.
end
ox=Conference.new
puts ox.sing("")        # "Singin' at Southwest Fox"
puts ox.play("guitar")  # "strum"
```

To be truly useful, of course, modules are typically stored in their own program files. We can store the Music module in its own program file named music.rb. To be able to include music.rb in another piece of code, we first have to bring it into runtime memory. This is done using either the require or load function.

Both require and load are special Ruby functions that load and execute an external file of Ruby source code. There is an internal difference in what they do, but all you really need to know for now is that they both accomplish the same purpose. Require is much more widely used than load.

In order for Ruby to know where to find the external file, it must be along the Ruby path or in a location relative to the code that needs to include it. The first example below uses a full path and file name to specify the location of music.rb on my machine. In the second example I'll show you a way of avoiding the need to specify the path explicitly every time.

```
require "/SWFox2010/Sessions/Ruby/Code/8_Modules/music"
def make_music
  include Music
  puts Music.sing("doremi")
  puts Music.play("tuba")
end
make_music()  # "do re mi fa so la te do", and "oompa"
```

Well, that's enough music for now, but we feel our conference also needs some animal friends. Here's a module that implements a speaking behavior for dogs and cats.

```
module Animals
  class Dog
    def make_sound
      "woof"
    end
  end
  class Cat
    def make_sound
      "meow"
    end
  end
end
```

In order to include this module in subsequent code without having to specify the full path to the file it resides in every time we include it, we can add the file's path to the Ruby load path. The Ruby load path is store in an array named '$:'. Remember that global variables start with a $, so you can figure out that $: is a global variable name where : is the name of the array. Although it looks funny at first, $: is simply the name of the Ruby load path.

In Ruby you can append a value to an array with the << operator. The following code displays the current load path and then appends the path to the folder containing the animals.rb file in which the Animals module is defined.

```
puts $:
$: << "/SWFox2010/Sessions/Ruby/Code/8_Modules"
```

Now we can require the Animals module without having to explicitly state where animals.rb is located. Once we have access to the Dog and Cat classes, we can create some specific animal subclasses and call the make_sound method to hear what each animal has to say.

Let's create a dog first.

```
require "animals"
class Labrador < Animals::Dog
  def speak
    make_sound  # parentheses are optional; make_sound() also works.
  end
end
Roscoe = Labrador.new
puts "Roscoe says " + Roscoe.speak  # "Roscoe says woof"
```

The Labrador class is defined a subclass of the Dog class in the Animals module. Because of this, the Labrador can call the make_sound method directly.

Now let's add a cat.

```
class HouseCat < Animals::Cat
  def speak
    make_sound
  end
end
Fluffy = HouseCat.new
puts "Fluffy says " + Fluffy.speak  # "Fluffy says meow"
```

It's often the case that a specific animal will have a different behavior than the default behavior defined for the superclass animal in the Animals module. In this case the subclass can override the default behavior of a method simply by implementing its own behavior instead. This is classic inheritance; the only difference is that the superclass is defined in another module.

Let's add a Siamese cat. If you know anything about cats, you may know that Siamese typically don't meow in the same way as other cats; instead, they make more of a yowling sound. In the following code our Siamese overrides the default meow with its own sound.

```
class Siamese < Animals::Cat
   def speak
      "rawwwwr!"
   end
end
Pookie = Siamese.new
puts "Pookie says " + Pookie.speak  # "Pookie says rawwwwr!"
```

Ruby language summary

The purpose of this section on The Ruby programming language has been to give you a solid if abbreviated introduction to Ruby. After reading it and studying its examples you should have had enough exposure and experience with Ruby to be able to read and understand the code you'll be seeing in the next section on the Ruby on Rails framework.

The Rails framework

Ruby on Rails is a framework written in Ruby for Ruby developers to create web applications. Ruby on Rails, usually referred to as just Rails for short, is actually much newer than the Ruby language. Ruby (the language) was first released in 1993, while the first version of Rails (the framework) wasn't released until 2004.

The two do not share the same authors, either. The Ruby language was created by Japanese programmer Yukihiro Matsumoto ("Matz"). The Rails framework, on the other hand, was created by Danish programmer David Heinemeier Hansson ("DHH"), who is a partner in the Chicago-based company 37Signals. Ruby on Rails evolved from the development work done to create one of 37Signals' products called Basecamp. DHH is still actively involved in the life and growth of the Rails framework including the just-released Rails 3.0. He'll be a keynote speaker at the RubyConf X conference in New Orleans in November, 2010. You can follow @dhh on Twitter if you're interested.

The home for Ruby on Rails on the Web is http://rubyonrails.org. The banner at the top of the home page reflects the design philosophy behind Rails:

> "Web development that doesn't hurt."

Sounds appealing, doesn't it? The home page also includes one of Rails' mantras, which is "convention over configuration". What this means that if you follow Rails' conventions—for example, in naming models, views, controllers, and database tables—then you don't have to do much if any extra configuration in order to get your application to work because it just does. We'll see the benefits of conforming to conventions as we learn how to build a Rails application.

The MVC pattern

Rails is based on the Model – View – Controller (MVC) design pattern for application architecture. MVC is a simple yet powerful pattern that has been implemented in many frameworks including the recent Microsoft ASP.NET MVC. The MVC pattern is easy to comprehend and its implementation in Ruby on Rails is easy to follow.

There are many ways to diagrammatically illustrate the MVC pattern. If you search the Web you can find some fairly complex examples, but in its basic form the diagram simply consists of three boxes and a couple of lines.

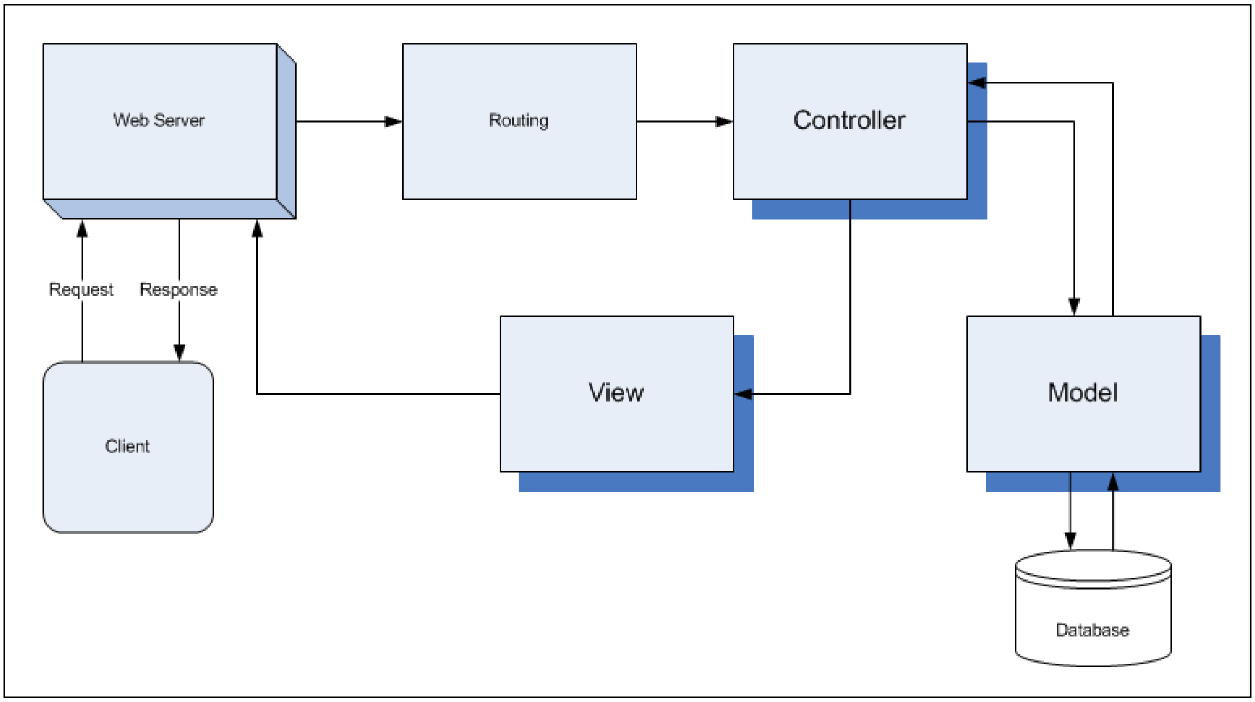


Figure 21: This MVC pattern diagram illustrates the interaction between the Model, the View, and the Controller in Rails.

In a nutshell, the Model in the MVC pattern is the data, or more accurately an object representation of the data. The View is the presentation of the data to the user. The Controller is the piece that interacts with the user on one side and with the other two MVC components on the other side in order to return the correct response to the user's request.

In an MVC pattern the basic flow is that a user sends a request to the Controller, which invokes the appropriate Model to get the required data and then uses the appropriate View to return the information to the user. In a Web application, the request is a HTTP request coming from a user via their browser. The View is typically returned to the user as an HTML document, although other formats such as XML are also possible.

Routes and REST

An MVC application naturally has many models, many views, and many controllers. In Rails there is a fourth element in the pattern called routes. Routing sits between the user and the Controller, acting as a dispatcher to determine which Controller is invoked to handle which requests. This is implemented by using RESTful URLs.

REST is an acronym for Representational State Transfer. There are entire books on the subject, but for our purposes as a beginning Rails developer about all we need to know is that a RESTful URL is structured in such a way as to identify the appropriate controller and the action that controller is supposed to take. We'll learn more about REST when we look at examples of how Rails URLs are constructed and used.

The ActiveRecord ORM

ORM is an acronym for Object-Relational Mapping. An ORM provides a mechanism for mapping data from a database to an object in an object-oriented program design.

Rails uses an ORM called ActiveRecord. Through the use of naming conventions, Rails makes the association of data with its object counterparts virtually transparent to the developer. In other words, if you use the standard Rails naming conventions you do not need to do any configuration to associate an entity object in a Rails application with its physical table in the database.

As an example, the following code defines a class named Product which is subclassed from the ActiveRecord Base class. By convention, class names are written in the singular form while table names take the plural. The class named Product (singular) is therefore automatically associated with a database table named products (plural).

```
class Product < ActiveRecord::Base
```

Rails is pretty smart when it comes to non-standard English language plurals. For example, it knows that a Person class should be associated with a table named people.

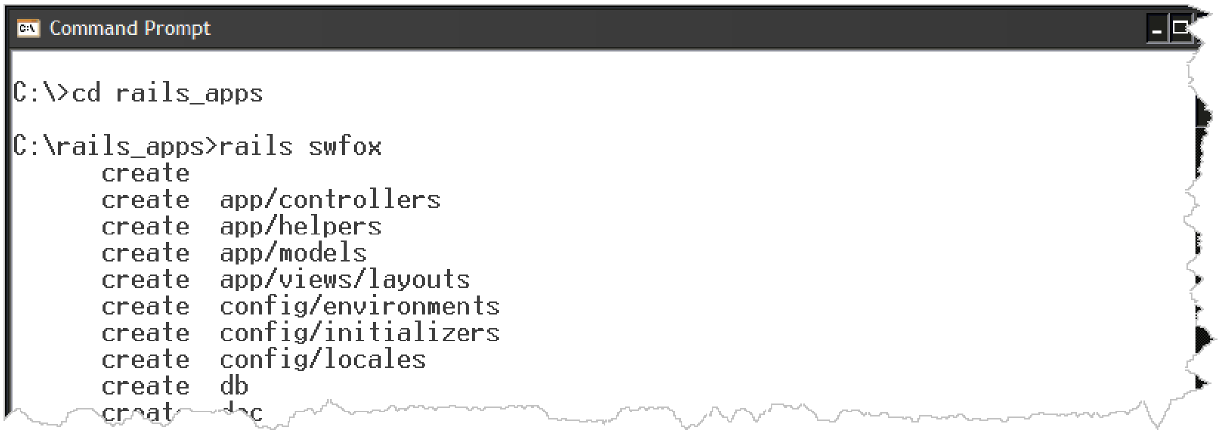
ActiveRecord provides all kinds of useful services in a Rails application. One example is validations, which we'll look at by example later on.

Creating a Rails app

Creating the basic structure of a Rails app is about as simple as it gets. First create a folder for your Rails apps on your hard drive. I use C:\rails_apps but you can use anything you like. Open a command window, change to your Ruby applications folder, think of a name for your application (like 'swfox') and tell Rails to create it like this:

```
C:\rails_apps> rails swfox
```

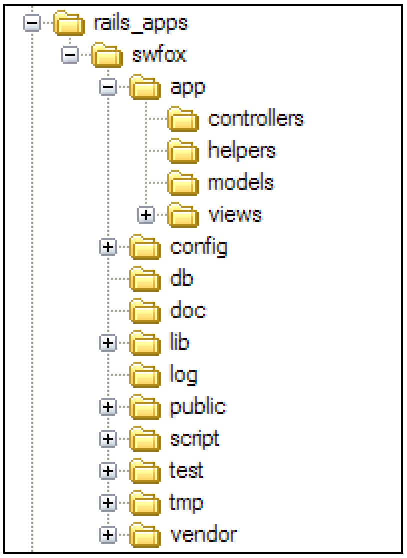
This creates an entire folder structure along with all the files necessary for a fully functional skeleton application. Figure 22 shows some of the output from the application generator, which is displayed as it runs.



```
C:\>cd rails_apps

C:\rails_apps>rails swfox
      create
      create  app/controllers
      create  app/helpers
      create  app/models
      create  app/views/layouts
      create  config/environments
      create  config/initializers
      create  config/locales
      create  db
      create  doc
```

Figure 22: From the command prompt in the appropriate folder, type 'rails swfox' to create a skeleton Rails application in the C:\rails_apps\swfox folder.

Figure 23 shows the folder structure created by Rails for the swfox application. The app folder has been expanded so you can see the subfolders where the files that implement the MVC design pattern are located. Through the use of explicit folder names, Rails makes it quite clear where each component of the MVC pattern resides.



```
rails_apps
  swfox
    app
      controllers
      helpers
      models
      views
    config
    db
    doc
    lib
    log
    public
    script
    test
    tmp
    vendor
```

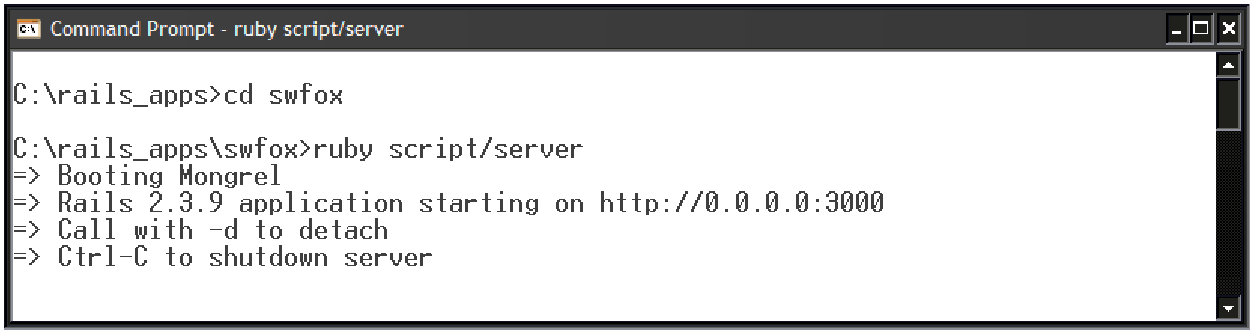
Figure 23: Rails creates an entire folder structure and files for the new application in one step.

Running the app

Your new Rails application doesn't really do anything at this point, but it does have a home page and it is a complete application that's ready to run. To launch the Rails Web server (WEBrick or Mongrel, depending on what's installed) and make the application ready to respond to requests from the browser, go to the command prompt and type

```
C:\rails_apps\swfox> ruby script/server
```

'Server' is a Ruby code file in the script folder. You can open it in your text editor to see what's in it (it isn't much). The server takes a few seconds to get ready, during which time you'll see a few lines appear in the command window.



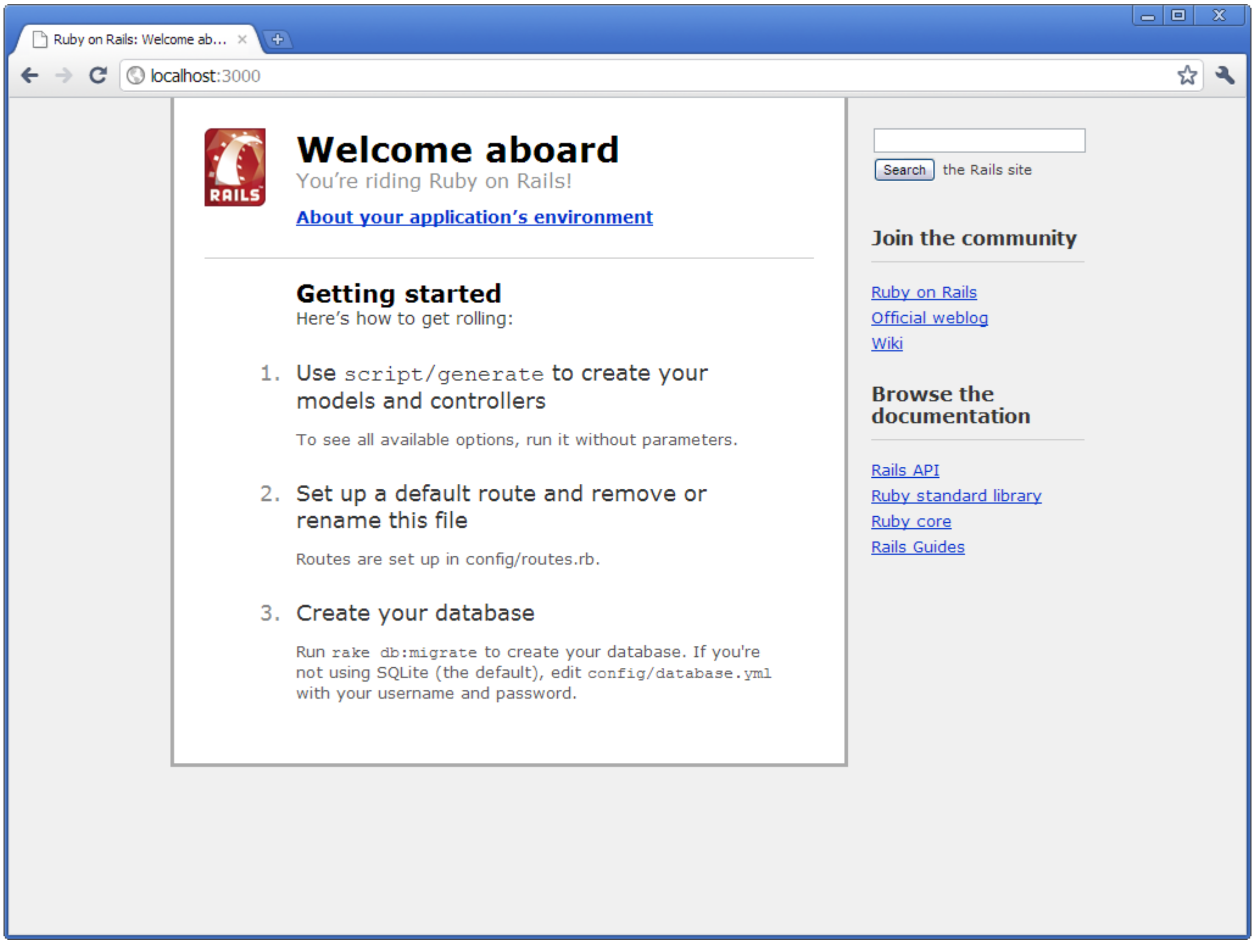
```
C:\rails_apps>cd swfox

C:\rails_apps\swfox>ruby script/server
=> Booting Mongrel
=> Rails 2.3.9 application starting on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
```

Figure 24: To launch the app in the Web server, change to the app folder and type 'ruby script/server'.

Figure 24 shows the Mongrel server initialized and ready to respond to requests from the browser. By default, Rails applications communicate with the local server on port 3000. To access the default page for your new Rails app, open your browse and navigate to

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

Rails processes the request and returns the default home page, as shown in Figure 25.



Welcome aboard

You're riding Ruby on Rails!

About your application's environment

Getting started

Here's how to get rolling:

1. Use `script/generate` to create your models and controllers

   To see all available options, run it without parameters.

2. Set up a default route and remove or rename this file

   Routes are set up in config/routes.rb.

3. Create your database

   Run `rake db:migrate` to create your database. If you're not using SQLite (the default), edit `config/database.yml` with your username and password.

Search the Rails site

Join the community

Ruby on Rails
Official weblog
Wiki

Browse the documentation

Rails API
Ruby standard library
Ruby core
Rails Guides

Figure 25: The default page of a new Rails application is accessible at http://localhost:3000. I generally use Firefox for Web development work, but Chrome has a nice, clean interface that makes for good screenshots.

The home page is pretty basic, but notice that it contains instructions designed to help you take the next steps. We'll explore these steps in the following sections of this paper. The links on the right of the home page take you to useful sites for additional information.

When you're ready to terminate the app and shut down the server, return to the command window and press Ctrl-C.[11]

Backing up your work in Rails is easy: simply make a copy of everything in the application folder, which in this case is C:\rails_app\swfox. If you're just playing around and don't want to save anything, you can always just delete the folder. There are no registry entries, hidden files or folders, or other leftovers to be concerned about.

[11] I've noticed that in some cases the server doesn't shut down until it receives another request. If this happens, just return to the browser and do a page refresh. It will either timeout or come back 404 in the browser, but the server will exit and you'll be returned to the command prompt in the command window.

Creating a database

Rails apps are designed to interact with a database. So far we don't have a database nor any code in place for interacting with one, so let's create some. Rails provides a tool for this purpose called the scaffold generator, sometimes referred to as just scaffolding.

The scaffold generator is an amazing tool. Without writing a single line of code, you can create database tables along with their corresponding models, views, and controllers. All you have to do is to specify the table name plus its column names and data types. The result is a ready-to-use Web application with basic CRUD[12] functions built in.

The scaffold generator is also a great learning tool. When you become an experienced Rails developer you may or may not want to use scaffolding as your starting point, but while learning it's a good way to get started. Not only does it relieve you of the need to create everything manually, but you can inspect the files the scaffold generator creates to see what's in them and how they relate to one another.

We want our SWFox application to keep track of people who are attending and/or speaking at Southwest Fox 2010. To do this we need a table to store each person's name, address, email address, and a flag to indicate whether or not they are a speaker. We can have the scaffold generator do all this for us with the following command:

```
C:\rails_apps\swfox> ruby script/generate scaffold person name:string address:string
email:string speaker:boolean
```

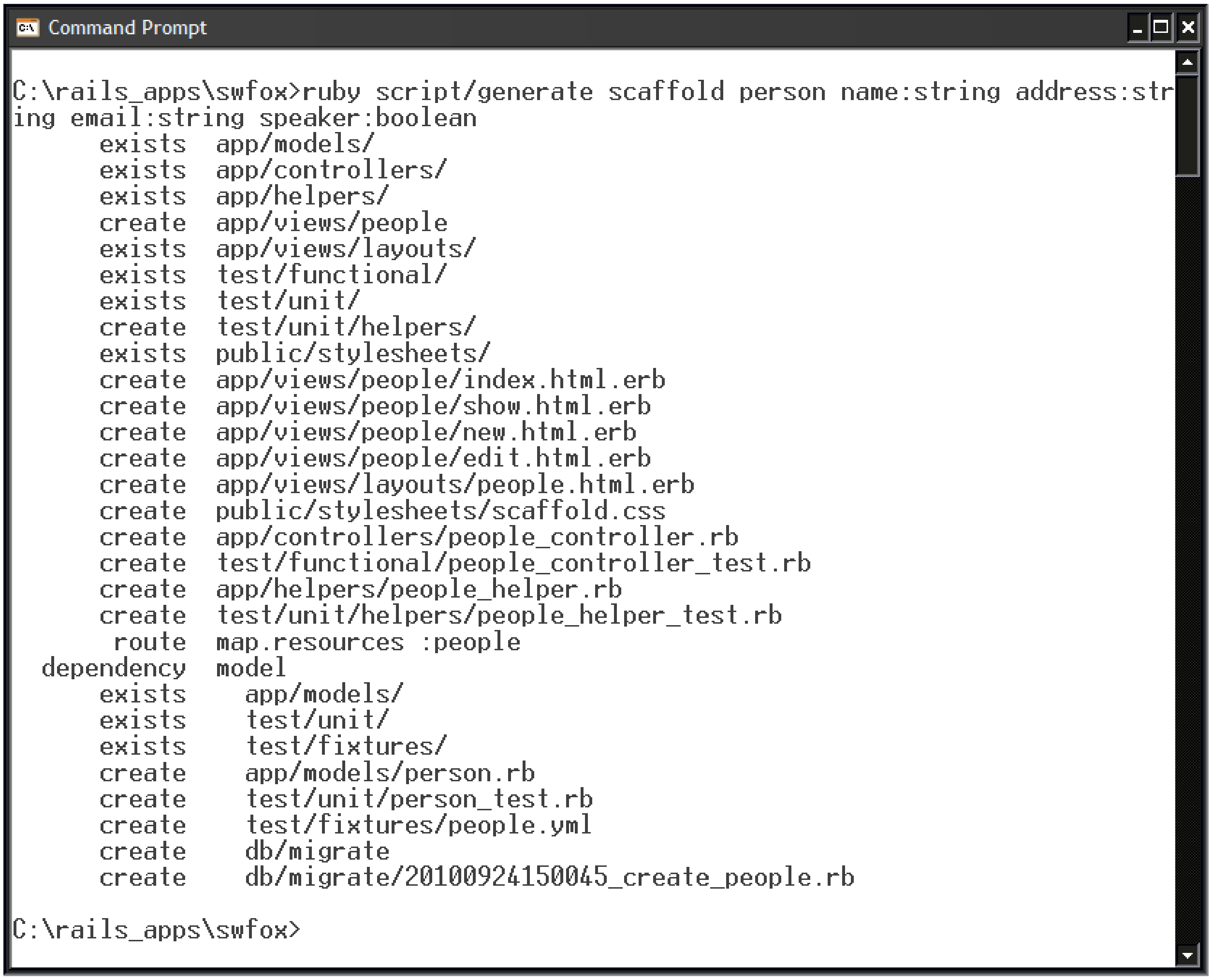
The scaffold generator displays its progress as it runs. Notice in Figure 26 that many of the folders already exist; this is because Rails already created the skeleton folder structure. The scaffold generator creates a couple of new folders and several files. The command prompt returns when the scaffolding process is complete.

Figure 26 shows several interesting things about what the scaffold generator has done. For one thing, we told it we wanted an entity named person but notice that it created a folder called app/views/people. Rails knows that the plural of person is people. The use of plural and singular is one of Rails' conventions, which are important to follow if you want to avoid having to manually edit configuration files (remember "convention over configuration").

Another thing to notice are the file name extension. Several are Ruby source code files, which you already know have a .rb file name extension. This is a Web app, but where are the .html files? Notice that there are several files with .erb file name extensions. The .erb file name extension stands for "embedded Ruby". These are HTML documents or document fragments with calls to Ruby code embedded in them, much in the same way that you can embed calls in ASP.NET or West Wind Web Connection. We'll come back to these later.

[12] Create, read, update, and delete.

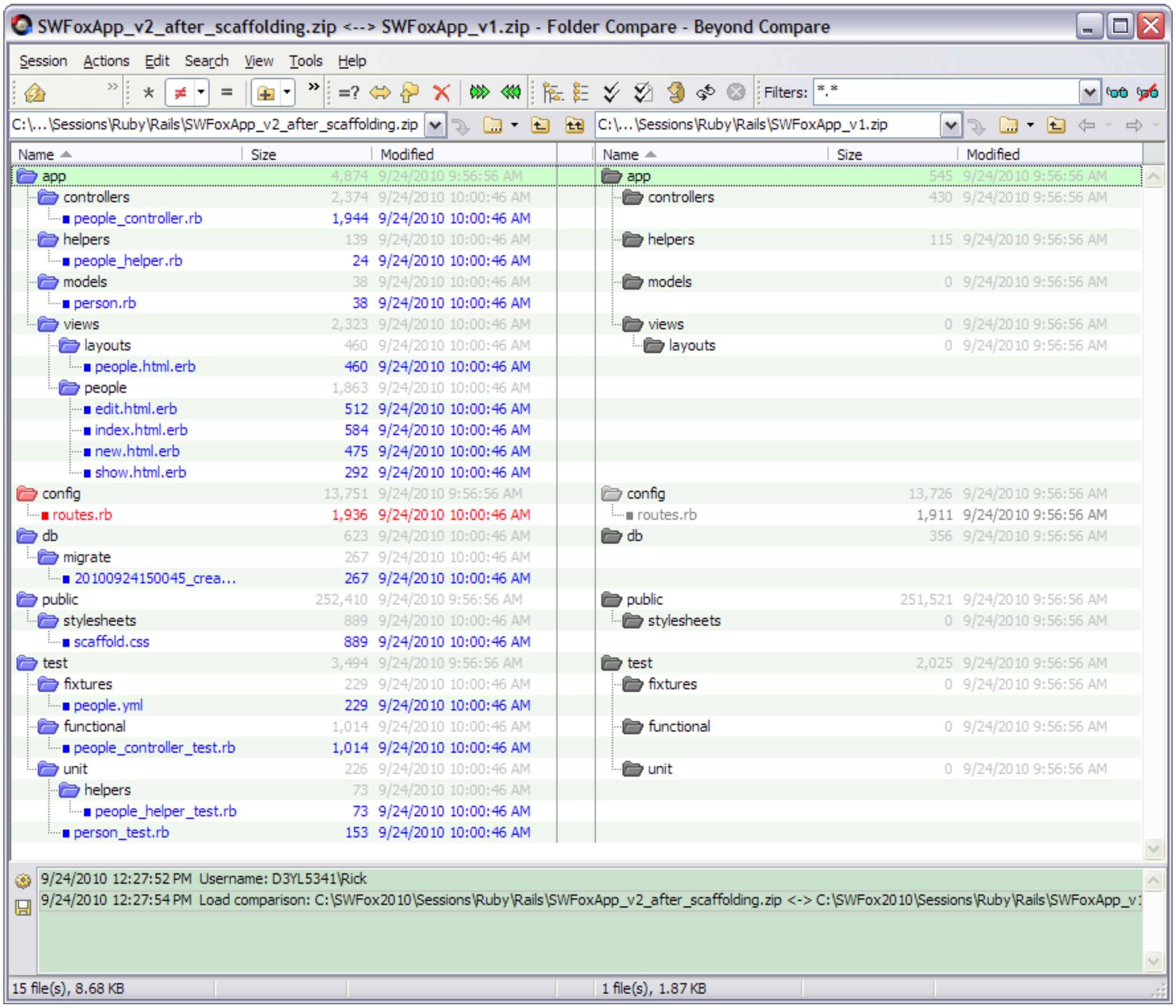
Finally, notice in Figure 26 that the scaffold generator created a db/migrate folder and created one file in it. This is the subject of the next topic, database migrations.



```
C:\rails_apps\swfox>ruby script/generate scaffold person name:string address:str
ing email:string speaker:boolean
      exists  app/models/
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/people
      exists  app/views/layouts/
      exists  test/functional/
      exists  test/unit/
      create  test/unit/helpers/
      exists  public/stylesheets/
      create  app/views/people/index.html.erb
      create  app/views/people/show.html.erb
      create  app/views/people/new.html.erb
      create  app/views/people/edit.html.erb
      create  app/views/layouts/people.html.erb
      create  public/stylesheets/scaffold.css
      create  app/controllers/people_controller.rb
      create  test/functional/people_controller_test.rb
      create  app/helpers/people_helper.rb
      create  test/unit/helpers/people_helper_test.rb
       route  map.resources :people
  dependency  model
      exists    app/models/
      exists    test/unit/
      exists    test/fixtures/
      create    app/models/person.rb
      create    test/unit/person_test.rb
      create    test/fixtures/people.yml
      create    db/migrate
      create    db/migrate/20100924150045_create_people.rb

C:\rails_apps\swfox>
```

Figure 26: In response to a single command, the scaffold generator creates all the folders and files necessary to perform basic CRUD functions on the person entity in our Web app.

The easiest way to see what the scaffold generator created is to look at a before and after picture of the application folders and files. Figure 27 shows a comparison of the SWFox application folders and files before and after scaffolding was run. The "before" side is on the right and the "after" side is on the left.



SWFoxApp_v2_after_scaffolding.zip <--> SWFoxApp_v1.zip - Folder Compare - Beyond Compare

Filters: *.*

C:\...\Sessions\Ruby\Rails\SWFoxApp_v2_after_scaffolding.zip

C:\...\Sessions\Ruby\Rails\SWFoxApp_v1.zip

Name	Size	Modified	Name	Size	Modified
app	4,874	9/24/2010 9:56:56 AM	app	545	9/24/2010 9:56:56 AM
controllers	2,374	9/24/2010 10:00:46 AM	controllers	430	9/24/2010 9:56:56 AM
people_controller.rb	1,944	9/24/2010 10:00:46 AM			
helpers	139	9/24/2010 10:00:46 AM	helpers	115	9/24/2010 9:56:56 AM
people_helper.rb	24	9/24/2010 10:00:46 AM			
models	38	9/24/2010 10:00:46 AM	models	0	9/24/2010 9:56:56 AM
person.rb	38	9/24/2010 10:00:46 AM			
views	2,323	9/24/2010 10:00:46 AM	views	0	9/24/2010 9:56:56 AM
layouts	460	9/24/2010 10:00:46 AM	layouts	0	9/24/2010 9:56:56 AM
people.html.erb	460	9/24/2010 10:00:46 AM			
people	1,863	9/24/2010 10:00:46 AM			
edit.html.erb	512	9/24/2010 10:00:46 AM			
index.html.erb	584	9/24/2010 10:00:46 AM			
new.html.erb	475	9/24/2010 10:00:46 AM			
show.html.erb	292	9/24/2010 10:00:46 AM			
config	13,751	9/24/2010 9:56:56 AM	config	13,726	9/24/2010 9:56:56 AM
routes.rb	1,936	9/24/2010 10:00:46 AM	routes.rb	1,911	9/24/2010 9:56:56 AM
db	623	9/24/2010 10:00:46 AM	db	356	9/24/2010 9:56:56 AM
migrate	267	9/24/2010 10:00:46 AM			
20100924150045_crea...	267	9/24/2010 10:00:46 AM			
public	252,410	9/24/2010 9:56:56 AM	public	251,521	9/24/2010 9:56:56 AM
stylesheets	889	9/24/2010 10:00:46 AM	stylesheets	0	9/24/2010 9:56:56 AM
scaffold.css	889	9/24/2010 10:00:46 AM			
test	3,494	9/24/2010 9:56:56 AM	test	2,025	9/24/2010 9:56:56 AM
fixtures	229	9/24/2010 10:00:46 AM	fixtures	0	9/24/2010 9:56:56 AM
people.yml	229	9/24/2010 10:00:46 AM			
functional	1,014	9/24/2010 10:00:46 AM	functional	0	9/24/2010 9:56:56 AM
people_controller_test.rb	1,014	9/24/2010 10:00:46 AM			
unit	226	9/24/2010 10:00:46 AM	unit	0	9/24/2010 9:56:56 AM
helpers	73	9/24/2010 10:00:46 AM			
people_helper_test.rb	73	9/24/2010 10:00:46 AM			
person_test.rb	153	9/24/2010 10:00:46 AM			

```
9/24/2010 12:27:52 PM  Username: D3YL5341\Rick
9/24/2010 12:27:54 PM  Load comparison: C:\SWFox2010\Sessions\Ruby\Rails\SWFoxApp_v2_after_scaffolding.zip <-> C:\SWFox2010\Sessions\Ruby\Rails\SWFoxApp_v1
```

15 file(s), 8.68 KB

1 file(s), 1.87 KB

Figure 27: A visual comparison of the application's folders and files before and after scaffolding reveals what the scaffold generator created.[13]

Database migrations

Database migrations are like version control for databases. Rails database migrations are run from ruby source code files using the Rake tool. Rake is Ruby's version of the Unix make tool and is widely used in Rails.

Database migrations are a service provided by ActiveRecord. As a Visual FoxPro developer, if you've ever used Doug Hennig's excellent Stonefield Database Toolkit (SDT) to update the structure of your VFP databases you have a sense of what database migrations do.

[13] This comparison was generated by Beyond Compare from Scooter Software, Inc. The image, like all images in this paper, was captured using SnagIt 10.0 from TechSmith Corporation. Both are outstanding tools that I recommend highly.

Database migrations, however, are a two-way street, enabling you to both update and roll back changes to the structure of tables in your database.

Before our SWFox web app can access the entity, we have to actually create it in a database. Earlier, we configured Rails to use the SQLite as its database, so when we run the database migration it will create a people table in SQLite.

Look at the last create statement in Figure 26. Notice that the name of the database migration file created by the scaffold generator (20100924150045_create_people.rb) includes a date and time as well as being descriptive of what it does. Different database migration files will have different names, which can be sequenced chronologically via the datetime stamp in the file name.

To run the database migration, type the following command at the command prompt:

```
C:\rails_apps\swfox> rake db:migrate
```

This creates the people table in a SQLite database file named development.sqlite3, which is written to C:\rails_apps\swfox\db. It's called development.sqlite3 because at the moment Rails is running in development mode – more on that later.

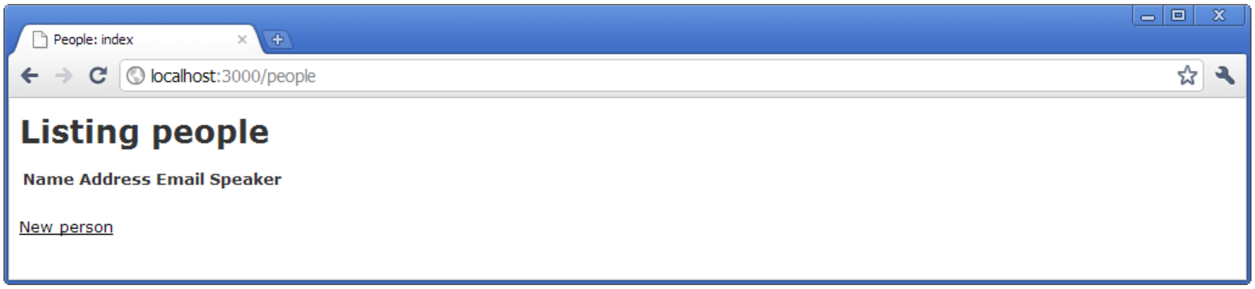
Listing 2 shows the contents of the database migration file we just ran. As you can see, it's just Ruby source code, which you now know how to read and understand.

Listing 2: This database migration file contains methods to update and roll back the structure of the people table. Because this is a new table, the update method (up) does a create while the roll back method (down) does a drop.

```
class CreatePeople < ActiveRecord::Migration
  def self.up
    create_table :people do |t|
      t.string :name
      t.string :address
      t.string :email
      t.boolean :speaker
      t.timestamps
    end
  end

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

Now we can start the Rails server running again, return to our browser and navigate to localhost:3000/people. This is a RESTful URL which tells Rails we want to talk to the people controller. The people table now exists in our database, which, because we followed conventions, is automatically hooked up to our Web application through the magic of Rails.

We haven't added any rows to the people table yet, so Rails displays an empty list with a link to add a new person.



Listing people

Name Address Email Speaker

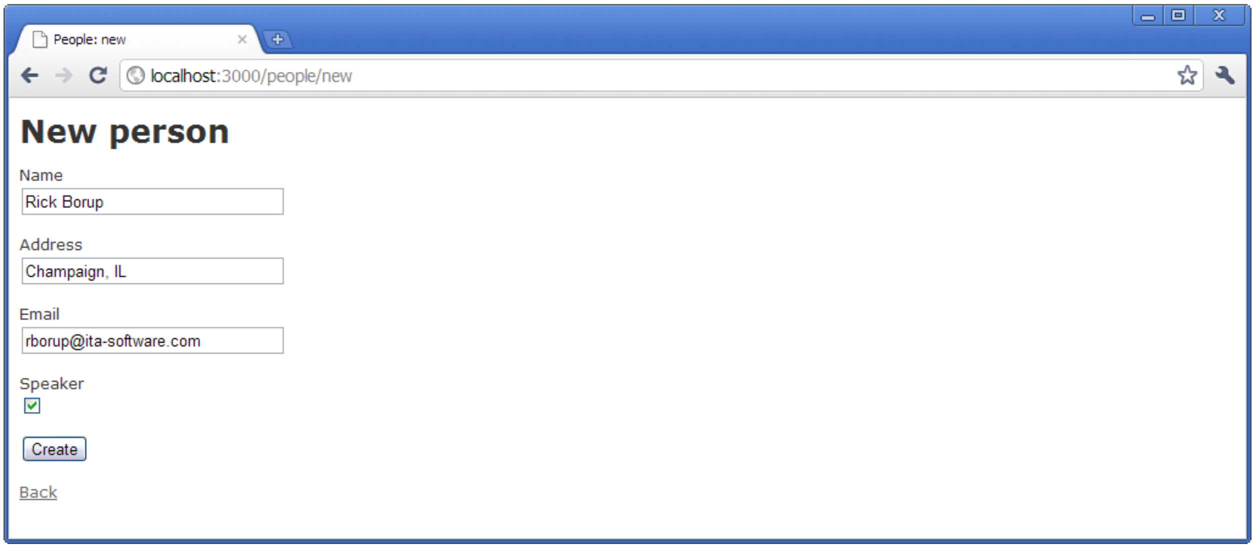
New person

Figure 28: There are no rows in the people table yet, so Rails just shows an empty list with a link to add a new person.

Congratulations – you're now running on Rails!

Using the app

From this point forward we can use the CRUD functions built in to the Rails application to add, edit, and delete people from the database. Clicking the New person link brings up a page containing a form to enter the data.



New person

Name
Rick Borup

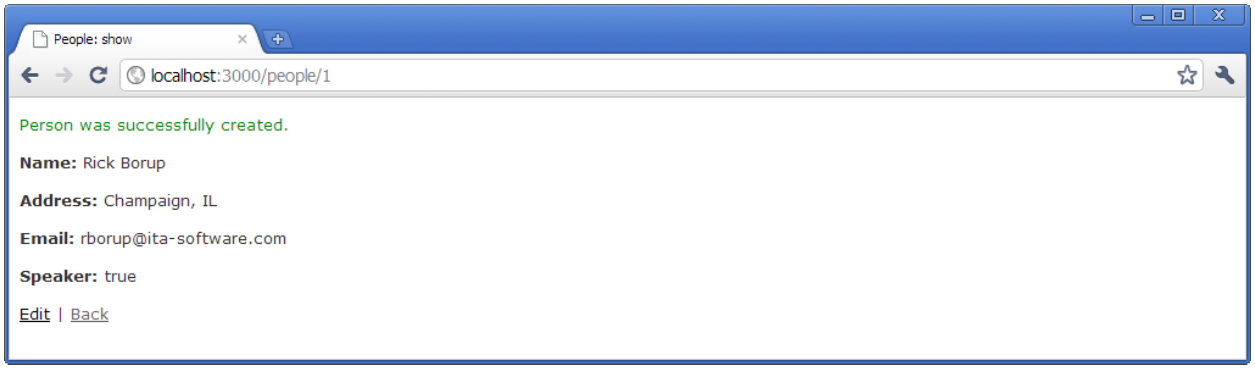
Address
Champaign, IL

Email
rborup@ita-software.com

Speaker
☑

Figure 29: Rails scaffolding automatically created a form to add a new person to the people table.

Clicking the Create button adds the new row to the database table. The response to the browser is a page reflecting the data that was added along with a message indicating success.



Person was successfully created.

Name: Rick Borup

Address: Champaign, IL

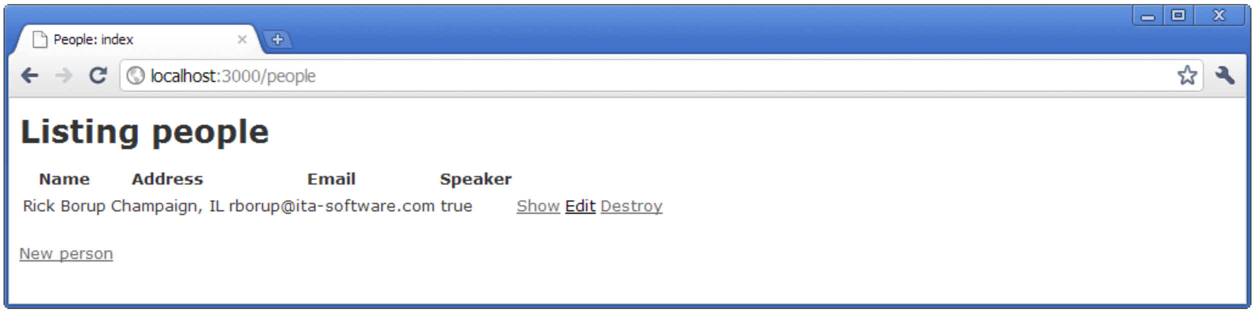
Email: rborup@ita-software.com

Speaker: true

Edit | Back

Figure 30: The browser displays a response reflecting the data that was entered along with a message indicating success.

After adding a person to the people table, the people listing page in the Rails app now shows one entry. Notice that there are links for Show (show detail), Edit, and Destroy (delete).



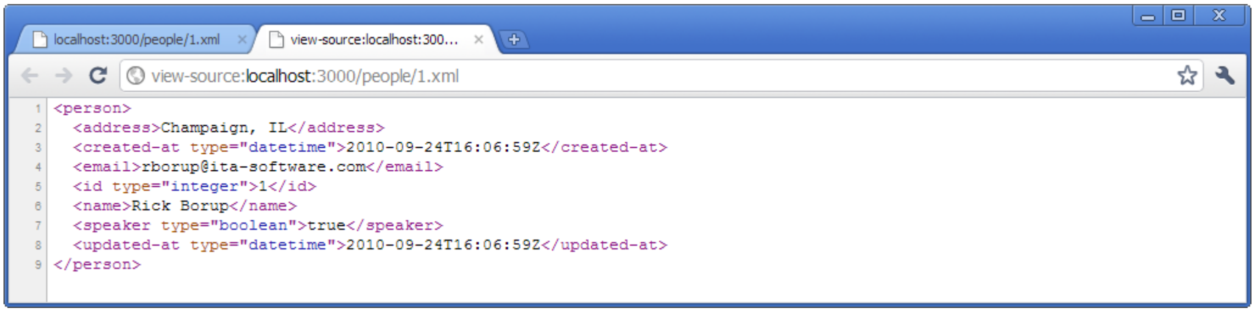
Listing people

Name	Address	Email	Speaker			
Rick Borup	Champaign, IL	rborup@ita-software.com	true	Show	Edit	Destroy

New person

Figure 31: The people list shows one row per person, with links to show detail, edit, or delete.

Side Note: Out of the box, Rails can return data either as HTML or as XML. To get the data back as XML, use a RESTful URL to specify that format. The URL to retrieve the data for the person with ID 1 in the default HTML format is localhost:3000/people/1. The URL to retrieve the same data as XML is localhost:3000/people/1.XML. The response looks like this:


localhost:3000/people/1.xml
view-source:localhost:300...
view-source:localhost:3000/people/1.xml

```
<person>
  <address>Champaign, IL</address>
  <created-at type="datetime">2010-09-24T16:06:59Z</created-at>
  <email>rborup@ita-software.com</email>
  <id type="integer">1</id>
  <name>Rick Borup</name>
  <speaker type="boolean">true</speaker>
  <updated-at type="datetime">2010-09-24T16:06:59Z</updated-at>
</person>
```

Figure 32: Rails can return XML as well as HTML.

Peering behind the curtains

How does Rails make all of this magic work? To understand that, we simply need to peer behind the curtains: everything we need to know can be found by looking at the files in the C:\rails_apps\swfox folder and its subfolders.

Config files

The files in the C:\rails_apps\swfox\config folder contain the configuration information for the Rails app.

Boot.rb is the file that boots Rails at startup. It starts with a big warning not to change it. Don’t.

Database.yml is the database configuration file. It's here that Rails establishes which database to use. Rails is designed to enable running in one of three modes: development, testing, or production. Each of these modes can have its own database and other individual configuration settings. The database.yml file for the SWFox application is listed below.

Listing 3: The database.yml file contains the database configuration settings.

```
# SQLite version 3.x
#   gem install sqlite3-ruby (not necessary on OS X Leopard)
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000

# Warning: The database defined as "test" will be erased and
# re-generated from your development database when you run "rake".
# Do not set this db to the same as development or production.
test:
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000

production:
  adapter: sqlite3
  database: db/production.sqlite3
  pool: 5
  timeout: 5000
```

Environment.rb contains the environment settings for the app. This is a Ruby source code file, and out of the box it's mostly comments to help you modify the configuration if necessary. You can inspect the environment.rb file on your own machine or in the session code for more details.

Routes.rb is where the routes are defined. As mentioned earlier, Rails uses routes to determine which controller is invoked to handle a request. There's generally no need to change anything in this file if you follow conventions. Listing 4 is a synopsis of the contents of the routes.rb file for the SWFox app.

Listing 4: A partial listing of the routes.rb file for the SWFox app.

```
ActionController::Routing::Routes.draw do |map|
  map.resources :people

  # The priority is based upon order of creation: first created -> highest priority.

  # Sample of regular route:
  #   map.connect 'products/:id', :controller => 'catalog', :action => 'view'
  # Keep in mind you can assign values other than :controller and :action

  # See how all your routes lay out with "rake routes"

  # Install the default routes as the lowest priority.
  # Note: These default routes make all actions in every controller accessible via
  # GET requests. You should consider removing or commenting them out if you're
  # using named routes and resources.

  map.connect ':controller/:action/:id'
  map.connect ':controller/:action/:id.:format'
end
```

Database files

The application's development database file are located in C:\rails_apps\swfox\db. The database file name is development.sqlite3, as referenced in the database.yml configuration file. Also in this folder is a file named schema.rb, which is the source for the current state of the database. Database migrations are based on information stored in this file.

The database migration command files are located in C:\rails_apps\swfox\db\migrate. Because we've only run one migration, there is only one file in this folder at the present time. Refer back to Listing 2 for details.

Side Note: As Visual FoxPro developers, we're accustomed to being able to look inside our databases with tools like browse and edit. How do we do that with a SQLite database? The SQLite3 command line is one way, but another way is to use a very cool add-on for the Firefox browser called SQLite Manager. This add-on lets you visually inspect and interact with any SQLite database on your system. Figure 33 illustrate how the people table looks when viewed using the SQLite Manager.

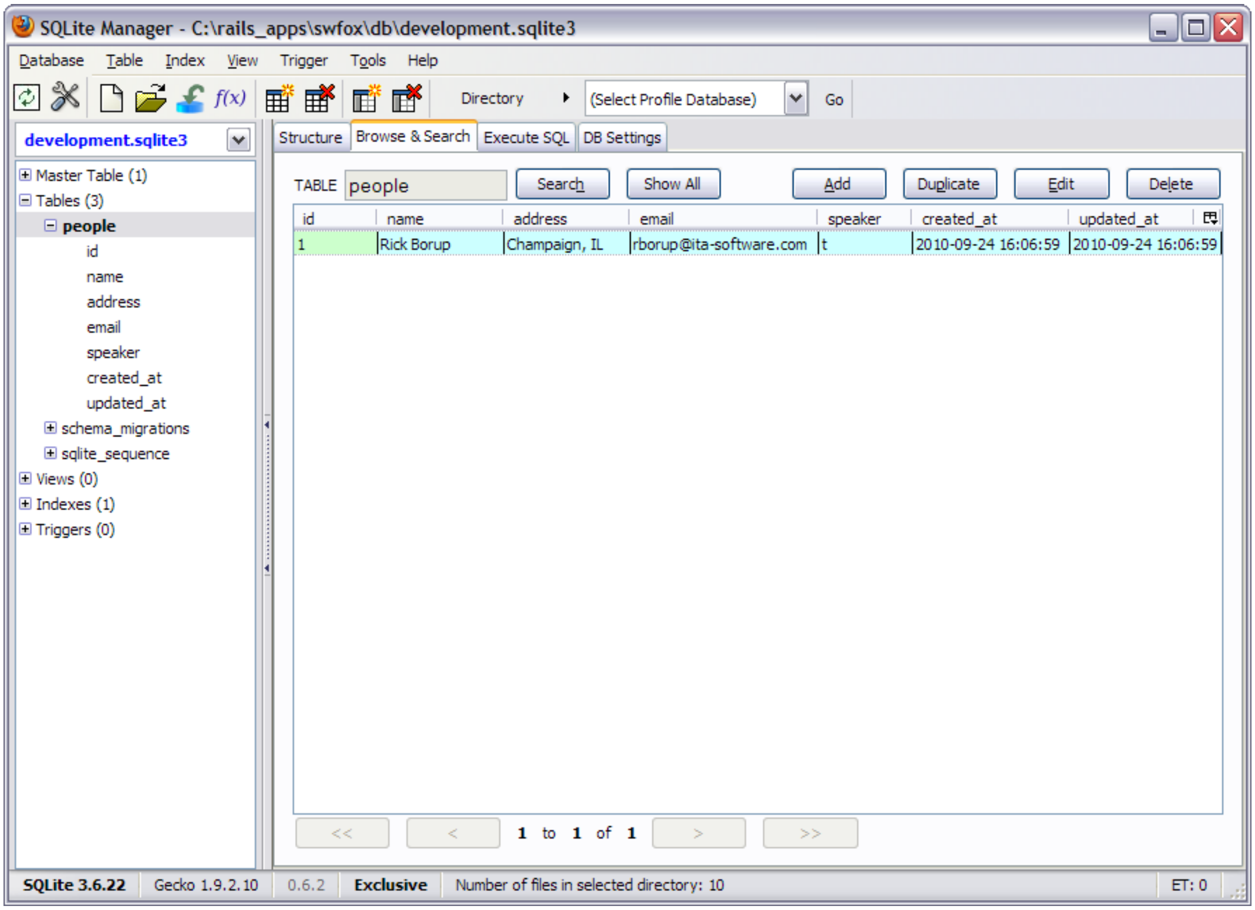


TABLE people

id	name	address	email	speaker	created_at	updated_at
1	Rick Borup	Champaign, IL	rborup@ita-software.com	t	2010-09-24 16:06:59	2010-09-24 16:06:59

Figure 33: SQLite Manager is a free add-on for the Firefox Web browser that enables visual interaction with a SQLite database.

MVC files

A Rails application's specific functionality is defined in the files located in C:\rails_apps\swfox\app. This folder is where the MVC pattern is implemented. Accordingly, it contains subfolders named models, views, and controllers. Each of these subfolders contains the files that implement that portion of the MVC pattern. There is also a fourth subfolder named helpers, which contains files that provide functionality that can be used by other files in the application.

Controllers

Rails applications have a default controller named application_controller.rb. This controller handles all requests that don't have a specific controller of their own.

In our SWFox application we also have a controller for actions related to the people entity. The people controller file, people_controller.rb, is one of the files that were created by the scaffold generator, as shown in Figure 27.

The people controller file is a bit too long to reproduce in its entirety here in the body of the paper, but Listing 5 is a snippet to give you an idea of what a controller looks like.

Listing 5: A snippet from people_controller.rb.

```
class PeopleController < ApplicationController
  # GET /people
  # GET /people.xml
  def index
    @people = Person.all

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

  # GET /people/1
  # GET /people/1.xml
  def show
    @person = Person.find(params[:id])

    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @person }
    end
  end

  # <snip>
end
```

You can see that class PeopleController is subclassed from ApplicationController. The index method renders data using the view defined in index.html.erb, while the show method renders data using the view defined in show.html.erb. Notice that XML output is an option in both cases.

Views

The views are defined in the files contained in C:\rails_apps\swfox\views. The views folder has two subfolders, layouts and people.

Layouts are kind of like templates: they provide the overall structure of an HTML document while leaving the details to the views. Our SWFox app has one layout called people.html.erb, as shown in Listing 6.

Listing 6: A layout defines the overall structure of an HTML document.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
```

```
  <meta http-equiv="content-type" content="text/html;charset=UTF-8" />
  <title>People: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>
<p style="color: green"><%= notice %></p>
<%= yield %>
</body>
</html>
```

There are a couple of important things to notice about the layout file. First, it's pretty much straightforward HTML. Second, it contains embedded Ruby expressions, which is why it has a name ending in .erb (meaning **e**mbedded **R**u**b**y). Notice that the format of the embedded Ruby expressions is the same as what you may be used to seeing in other dynamic web pages such as ASP.NET or West Wind Web Connection in VFP.

The yield statement deserves special mention. This is a special keyword in Ruby that's used to temporarily return control to a calling method. It's widely used in Ruby internals to implement all kinds of functionality such as the iterator methods we saw earlier. Iterator methods yield a series of values, each of which is returned in sequence to the code that's acting on the series. Yield's functionality in a Rails layout is conceptually similar, allowing the view to fill in whatever detail it's supplying to the finished HTML document.

The other subfolder under views is the people folder. It exists because we defined a people table from which data will be drawn to render in a view. There are four views in the people folder, one each for the edit, index, new, and show actions. Listing 7 is the code from show.html.erb, the view that shows the detail for one specific person.

Listing 7: The show.html.erb view is used to show the detail for a specific row in the people table.

```
<p>
  <b>Name:</b>
  <%=h @person.name %>
</p>
<p>
  <b>Address:</b>
  <%=h @person.address %>
</p>
<p>
  <b>Email:</b>
  <%=h @person.email %>
</p>
<p>
  <b>Speaker:</b>
  <%=h @person.speaker %>
</p>
<%= link_to 'Edit', edit_person_path(@person) %> |
<%= link_to 'Back', people_path %>
```

Note that this file also contains embedded Ruby expressions, which are used to insert the appropriate instance variables into the result (remember that instance variables have names beginning with an @).

The h in an expression like <%=h @person.name %> invokes the h( ) function, which is short for html_escape( ), on the value of @person.name. This function is used as a safety measure to wrap variables whose values are not themselves supposed to contain HTML. It's a way of protecting your app from HTML injection attacks via parameters or from data in HTML forms.

Together, the layout and the view combine to create the HTML document returned to the browser in response to the request. Listing 8 shows the complete HTML document returned by the SWFox app in response to a request for http://localhost:3000/people/1.

Listing 8: The layout and the view combine to form the completed HTML document.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content="text/html;charset=UTF-8" />
  <title>People: show</title>
  <link href="/stylesheets/scaffold.css?1285340445" media="screen" rel="stylesheet"
type="text/css" />
</head>
<body>
<p style="color: green"></p>
<p>
  <b>Name:</b>
  Rick Borup
</p>
<p>
  <b>Address:</b>
  Champaign, IL
</p>
<p>
  <b>Email:</b>
  rborup@ita-software.com
</p>
<p>
  <b>Speaker:</b>
  true
</p>
<a href="/people/1/edit">Edit</a> |
<a href="/people">Back</a>
</body>
</html>
```

Models

The models reside in C:\rails_apps\swfox\models. In our SWFox application there is only one database table, so at this point there is only one model. Its file name is person.rb.

In its current form, the model is trivially simple: it's just a Person class that's subclassed from ActiveRecord::Base. At this point the model doesn't have to do anything except to exist.

Listing 9: In the model for the people table, the Person class is subclassed from ActiveRecord::Base.

```
class Person < ActiveRecord::Base
end
```

In an upcoming section we'll update the model to implement some data validations.

RESTful URLs

As mentioned earlier, Rails and its implementation of the MVC design pattern rely on the use of RESTful URLs. RESTful URLs take the following forms:

```
http://localhost:3000/controller
http://localhost:3000/controller/action
http://localhost:3000/controller/action/id
http://localhost:3000/controller/action/id.format
```

You can see that the URL specifies the name of the server (localhost:3000 in our example), followed by the name of the controller. This string can optionally be followed by the name of the desired action, then the id of the entity to be used (if one needs to be specified), and finally the output format if other than the default.

It's easy to see how this works by looking at some examples. The http:// portion of each URL is omitted for greater readability.

To list people, use the default action on the people controller.

```
localhost:3000/people
```

To add a new person, invoke the 'new' action on the people controller.

```
localhost:3000/people/new
```

To show the detail for a specific person, invoke the 'show' action and pass the ID of the desired entity.

```
localhost:3000/people/show/1
```

To retrieve the detail for a specific person as XML, do the same as above but append the .xml format request.

```
localhost:3000/people/show/1.xml
```

To edit a person's record, invoke the 'edit' method and pass the id.

```
localhost:3000/people/edit/1
```

And so on. If you keep the basic format of a RESTful URL in mind – server name, controller name, action, id – then it's easy to see how any given URL relates to the controllers in a Rails app.

Rails validations

Rails validations are provided by ActiveRecord. Validations are implemented in the model. ActiveRecord validations have self-explanatory names, such as

```
validates_presence_of
validates_uniqueness_of
validates_length_of
validates_numericality_of
```

In our SWFox application, let's say we want to prevent bad or incomplete data from getting into our database. We decide that the name field needs to be required, and that the email address must be unique within the people table. We can easily implement these validations by editing the model for the people entity, person.rb, and adding the appropriate validations inside the Person class definition.

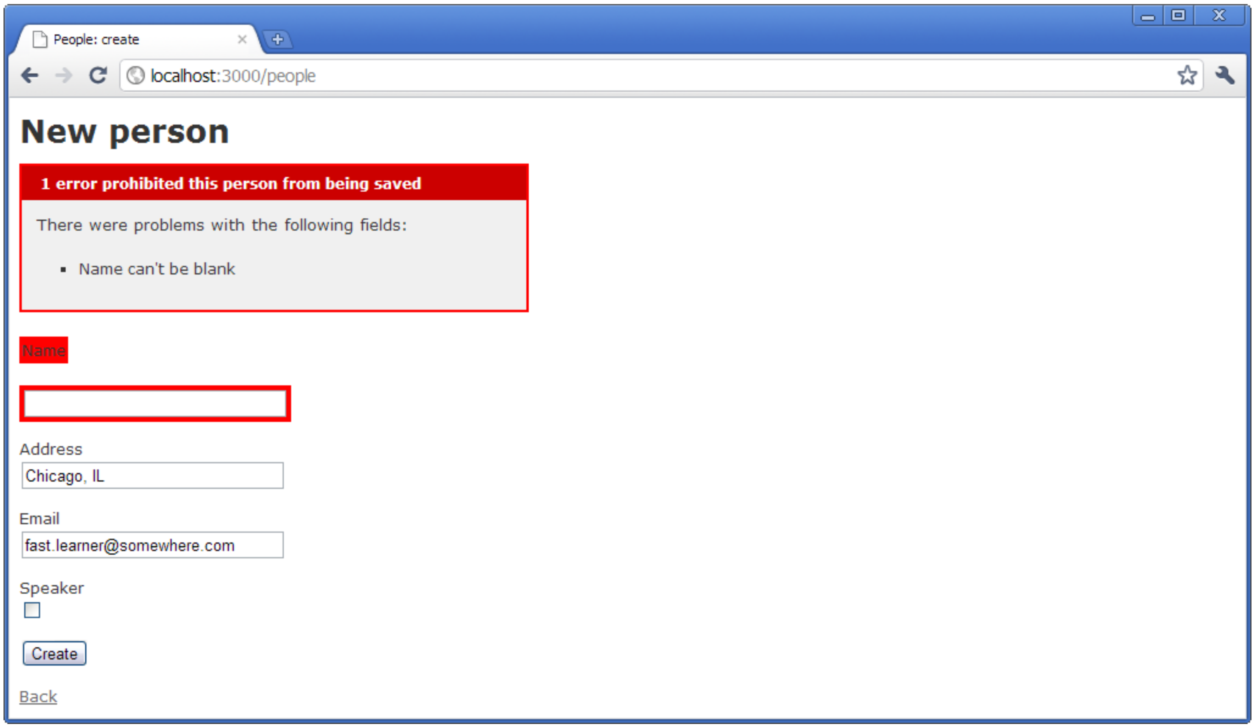
Listing 10: Validations are implemented in the model.

```
class Person < ActiveRecord::Base
   validates_presence_of :name
   validates_uniqueness_of :email
end
```

Notice the use of Ruby symbols for :name and :email.

Our application will now reject any attempt to add or edit a person if the name field is empty or if the email address provided has already been used by another person. If one of these conditions occurs, Rails also displays an appropriate error message without you having had to write any code.

Figure 34 shows the response if we try to add a person without a name.



New person

1 error prohibited this person from being saved

There were problems with the following fields:

- Name can't be blank

Name

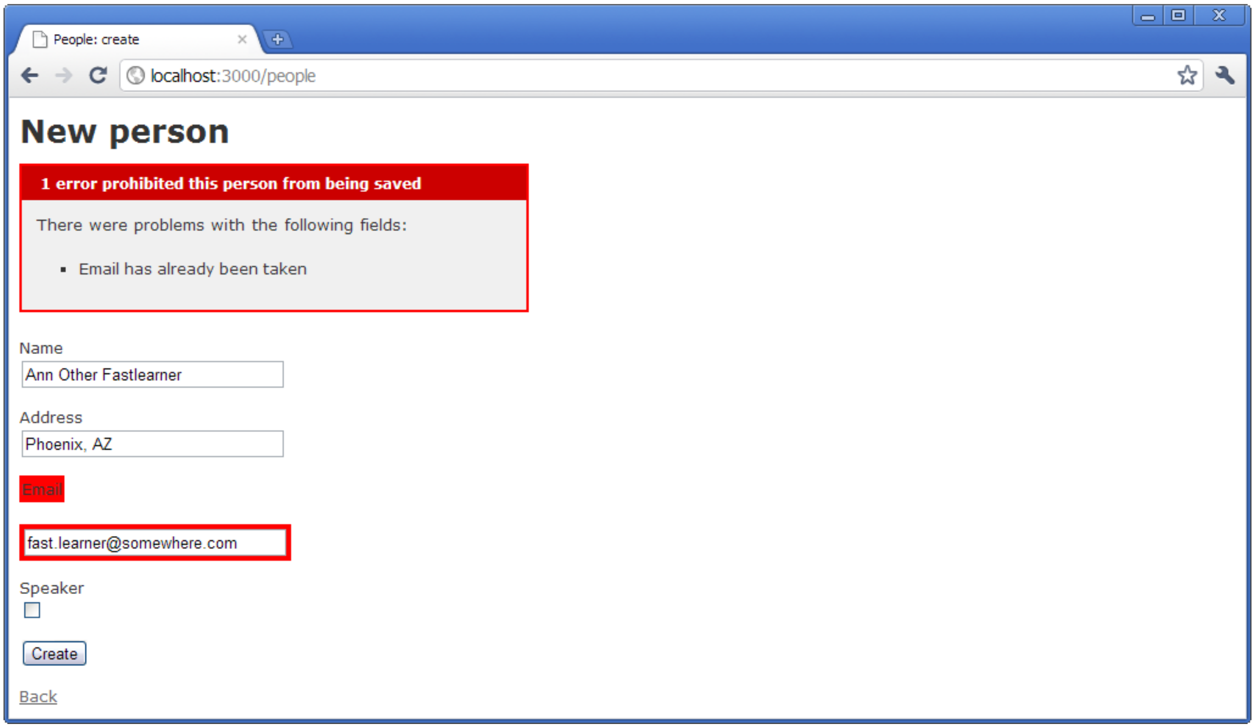
Address
Chicago, IL

Email
fast.learner@somewhere.com

Speaker
□

Figure 34. The validates_presence_of :name validation detects the missing name and the browser reports the error.

If we correct the error, save the record, and then try to add another person with the same email address, this fails because of the validates_uniqueness_of :email validation. As before, the browser shows an appropriate error message.



New person

1 error prohibited this person from being saved

There were problems with the following fields:

- Email has already been taken

Name
Ann Other Fastlearner

Address
Phoenix, AZ

Email
fast.learner@somewhere.com

Speaker
□

Figure 35: The validates_uniqueness_of :email validation detects a duplicate email address and the browser reports the error.

Resources

Although we've spent about sixty pages – and, if you attended the pre-conference session, about three classroom hours – looking at Ruby and Rails, we've really only just scratched the surface. Fortunately, there is nothing if not an abundance of resources available to help you continue on your way to mastery of both. This includes not only printed material but also any number of Web articles, tutorials, videos, and other useful information.

The complete list of references I used in preparing this paper is too long to include, but I am indebted to all of them. If time and energy permit, I'll post a complete list on my website. However, I would like to cite five books that were particularly helpful. In no particular order they are:

Flanagan, D., and Matsumoto, Y. 2008. The Ruby Programming Language. Sebastopol, CA: O'Reilly Media, Inc. ISBN 978-0-596-51617-8

Thomas, D., with Fowler, C. and Hunt, A. 2009. Programming Ruby 1.9. The Pragmatic Programmers, LLC. ISBN 978-1-934356-08-1

Ruby, S., Thomas, D. and Heinemeier, D.H. 2009. Agile Web Development with Rails, Third Edition. The Pragmatic Programmers, LLC. ISBN 978-1-9343561-6-6

St. Laurent, S., and Dumbill, E. 2009. Learning Rails. Sebastopol, CA: O'Reilly Media, Inc. ISBN 978-0-596-51877-6

Cangiano, A. 2009. Ruby on Rails® for Microsoft Developers. Indianapolis, IN: Wiley Publishing, Inc. ISBN 978-0-470-37495-5

The session code zip file for this presentation contains the entire SWFox Rails application we generated in the Rails portion of this paper. You can certainly use it for reference if you want to, but for the best learning experience I strongly recommend following the steps in this paper and creating your own Rails application. It's not difficult, and you'll get a lot more out of it that way.

As for other resources, remember GIYF – Google™ is your friend.

Summary

Ruby is a dynamic and exciting language. The Ruby on Rails framework makes it easy to develop database Web sites. Together, knowledge of the two gives you another great tool to add to your software development toolkit. I encourage you to use what you've learned from this paper as a launching pad on your continuing path towards mastery of Ruby and Rails.