Distributing Rails Applications - A Tutorial

Table of Contents

Distributing Rails Applications

A Tutorial

Sun Dec 24 19:01:32 UTC 2006

Erik Veenstra <erikveen@dds.nl>

1. Introduction

(Before we start: On real production servers, don't "deploy" a Rails application like this. That's not a good idea. But if you only want to ship your demo, well, read on...)

I get a lot of emails about packing and distributing Rails applications with Tar2RubyScript and RubyScript2Exe. It obviously wasn't easy to come up with the steps that have to be taken to transform a Rails application into a standalone application. Since I never built a Rails application myself, I wasn't even sure if it was possible at all. That's why I decided to write this tutorial.

In this tutorial, we'll go through the following steps:

1. Setup the environment
2. Create the SQLite database
3. Develop the Rails application
4. Create the RBA (= Ruby archive) from the application with Tar2RubyScript
5. Create the standalone executable with RubyScript2Exe

Before we begin, I assume that you've installed the following components (The version numbers are the versions I've installed myself.):

- Ruby (1.8.2)
- Rails (0.14.3)
- SQLite (2.8.15)
- Ruby-SQLite bindings (2.2.3)

Since a lot of information about the installation of each individual component can be found on the 'Net, I skip the do-this-to-install-that part. Well, we have to start somewhere...

For this tutorial, I used the RubyInstaller version of Ruby on Windows 2000 and installed both Rails and SQLite-Ruby with RubyGems. But you could use Linux as well. Every single step is the same on both platforms. The differences are just textual: back slashes instead of forward slashes, `c:\>` instead of # and `copy` instead of `cp`. That's it.

More general information about distributing Ruby applications (especially Tar2RubyScript, RubyScript and how they work together) can be found here.

2. Ingredients

2.1. Ruby

Ruby is "the interpreted scripting language for quick and easy object-oriented programming. It has many features to process text files and to do system management tasks (as in Perl). It is simple, straight-forward, extensible, and portable".

See the Ruby site for more information about Ruby.

2.2. Rails

Rails is "a full-stack, open-source web framework in Ruby for writing real-world applications with joy and less code than most frameworks spend doing XML sit-ups".

See the Rails site for more information about Rails.

2.3. SQLite

SQLite is "a self-contained, embeddable, zero-configuration SQL database engine".

See the SQLite site for more information about SQLite.

2.4. Ruby-SQLite Bindings

Ruby-SQLite "allows Ruby programs to interface with the SQLite database engine".

See the SQLite-Ruby site for more information about SQLite-Ruby.

2.5. Tar2RubyScript

Tar2RubyScript "transforms a directory tree, containing your application, into **one single Ruby script**, along with some code to handle this archive. This script can be distributed to our friends. When they've installed Ruby, they just have to double click on it and your application is up and running!"

See the Tar2RubyScript site for more information about Tar2RubyScript.

This is all you need: `tar2rubyscript.rb`. A "`gem install tar2rubyscript`" will do, too.

2.6. RubyScript2Exe

RubyScript2Exe "transforms your Ruby script into a standalone, compressed Windows, Linux or Mac OS X (Darwin) executable. You can look at it as a "compiler". Not in the sense of a

source-code-to-byte-code compiler, but as a "collector", for it collects all necessary files to run your script on an other machine: the Ruby script, the Ruby interpreter and the Ruby runtime library (stripped down for this script). Anyway, the result is the same: a standalone executable (*application*.exe). And that's what we want!"

See the RubyScript2Exe site for more information about RubyScript2Exe.

This is all you need: rubyscript2exe.rb. A "gem install rubyscript2exe" will do, too.

3. The Steps

3.1. Setup the Environment

First of all, create a simple Rails application and test it:

```
C:\rails> rails demo
C:\rails> cd demo\
C:\rails\demo> ruby script\server
```

Point your browser to `http://localhost:3000/`.

3.2. Create the SQLite Database

Now let's create a test database:

With Ruby:

```
C:\rails\demo> ruby
require "rubygems"
require_gem "sqlite-ruby"
SQLite::Database.new("demo_dev.db").execute(
"create table books (id integer primary key,   \
                     title varchar(255),       \
                     author varchar(255));")
^D # That's ctrl-D...
```

Or with SQLite:

```
C:\rails\demo> sqlite demo_dev.db
SQLite version 2.8.15
Enter ".help" for instructions
sqlite> create table books
   ...> (id integer primary key,
   ...> title varchar(255),
   ...> author varchar(255)
   ...> );
sqlite> .quit
```

And copy this empty database to a test database and a production database, for later usage:

```
C:\rails\demo> copy demo_dev.db demo_tst.db
C:\rails\demo> copy demo_dev.db demo_prd.db
```

3.3. Develop the Rails Application

Configure the application to use SQLite:

```
C:\rails\demo> notepad config\database.yml
```

Replace the contents of this file with something like this:

```
development:
  adapter: sqlite
  database: demo_dev.db

test:
  adapter: sqlite
  database: demo_tst.db

production:
  adapter: sqlite
  database: demo_prd.db
```

Create the model and the controller:

```
C:\rails\demo> ruby script\generate model Book
C:\rails\demo> ruby script\generate controller Book
C:\rails\demo> notepad app\controllers\book_controller.rb
```

Edit this file, so it looks like this:

```
class BookController < ApplicationController
  scaffold :book
end
```

Test it:

```
C:\rails\demo> ruby script\server
```

Point your browser to `http://localhost:3000/book/list` and add some new books.

3.4. Create the RBA from the Application with Tar2RubyScript

Now comes the trickiest part...

Tar2RubyScript transforms your application (the complete directory tree: program, configuration and user data) into a single Ruby script, called RBA (= Ruby archive). When running this RBA, it unpacks itself to a temporary directory before starting the embedded application. On termination, this directory is deleted, along with our user data... That's not what we want! So we have to move the user data to a safe place before running our application as an RBA.

(In the ideal world, we should "externalize" the logs and some config files as well.)

This means that we have to adjust our code:

```
C:\rails\demo> notepad config\environment.rb
```

Add this at the top of the file:

```
module Rails
  class Configuration
    def database_configuration
      conf = YAML::load(ERB.new(IO.read(database_configuration_file)).result)
```

```
      if defined?(TAR2RUBYSCRIPT)
        conf.each do |k, v|
          if v["adapter"] =~ /^sqlite/
            v["database"] = oldlocation(v["database"]) if v.include?("database")
            v["dbfile"]   = oldlocation(v["dbfile"])   if v.include?("dbfile")
          end
        end
      end
      conf
    end
  end
end
```

This overwrites `Rails::Configuration#database_configuration`, which was originally defined as:

```
module Rails
  class Configuration
    def database_configuration
      YAML::load(ERB.new(IO.read(database_configuration_file)).result)
    end
  end
end
```

What happens? When running an RBA, the programmer has to deal with two kind of locations: the location in which the user started the application (accessible with `oldlocation()`) and the temporary directory with the application itself, created by the RBA (accessible with `newlocation()`). By using `oldlocation()` in the code above, we simply adjust the data from `config/database.yml`.

Without the adjustment, `conf` would look like this:

```
{"development"=>{"database"=>"demo_dev.db", "adapter"=>"sqlite"}
 "production"=>{"database"=>"demo_prd.db", "adapter"=>"sqlite"}
 "test"=>{"database"=>"demo_tst.db", "adapter"=>"sqlite"}}
```

With the adjustment, `conf` would look like this:

```
{"development"=>{"database"=>"/full/path/to/demo_dev.db", "adapter"=>"sqlite"}
 "production"=>{"database"=>"/full/path/to/demo_prd.db", "adapter"=>"sqlite"}
 "test"=>{"database"=>"/full/path/to/demo_tst.db", "adapter"=>"sqlite"}}
```

Tar2RubyScript also needs `init.rb`, which is the entry point. So we create it:

```
C:\rails\demo> notepad init.rb
```

It looks like this:

```
at_exit do
  require "irb"
  require "drb/acl"
  require "sqlite"
end

load "script/server"
```

The `require`'s are a little trick. If you start and stop a Rails application, without any interaction with a browser, not all `require`'s of the program are encountered, so RubyScript2Exe might miss some libraries. Requiring them at the end avoids this problem. It's just bad behavior (in casu of Rails...) to `require` some libraries halfway a program...

Now it's time to pack the application into an RBA:

```
C:\rails\demo> cd ..
C:\rails> ruby tar2rubyscript.rb demo\
```

Tar2RubyScript creates `demo.rb`, which is the RBA. This RBA contains both the application and the databases. But we're not going to use this embedded user data (do you remember `oldlocation`?), so we copy the DB's to the current directory:

```
C:\rails> copy demo\*.db
```

Now we can test our RBA:

```
C:\rails> ruby demo.rb [-e environment]
```

Once again: the DB's we use when running as RBA are not the same as the DB's we use when running the application the old way! The sets are simply in another directory.

3.5. Create the standalone Executable with RubyScript2Exe

Creating a standalone executable is as simple as this:

```
C:\rails> ruby rubyscript2exe.rb demo.rb
^C (When Rails is started...) # That's ctrl-C...
```

RubyScript2Exe creates `demo.exe` (or `demo_linux` or `demo_darwin`, depending on your platform).

Now we copy `demo.exe` and `demo_*.db` to our USB memory stick, drive to our customer (he hasn't Ruby...) and simply start the application:

```
C:\rails> demo.exe [-e environment]
```

4. Conclusion

No installation of Ruby, no installation of RubyGems, no installation of the various gems, no installation of SQLite; There's only one executable!

Another conclusion concerns Rails. Rails has two architectural shortcomings (related to distributing the application):

- The user data isn't separated from the application itself.
- `require` is used halfway the program. This is not RubyScript2Exe-friendly.

As we've seen, both shortcomings can be worked around.