Applications - Theory and Practice of Building, Packing and Distributing

Table of Contents

Distributing Ruby Applications

Theory and Practice of Building, Packing and Distributing Ruby Applications

Fri May 25 15:05:33 UTC 2007

Erik Veenstra <erikveen@dds.nl>

1. Introduction

This is how I build, pack and distribute my Ruby applications. Theory and practice. The ultimate goal is to be able to distribute just one executable (Linux, Darwin or Windows) which contains both the application and the Ruby interpreter.

The three main steps are:

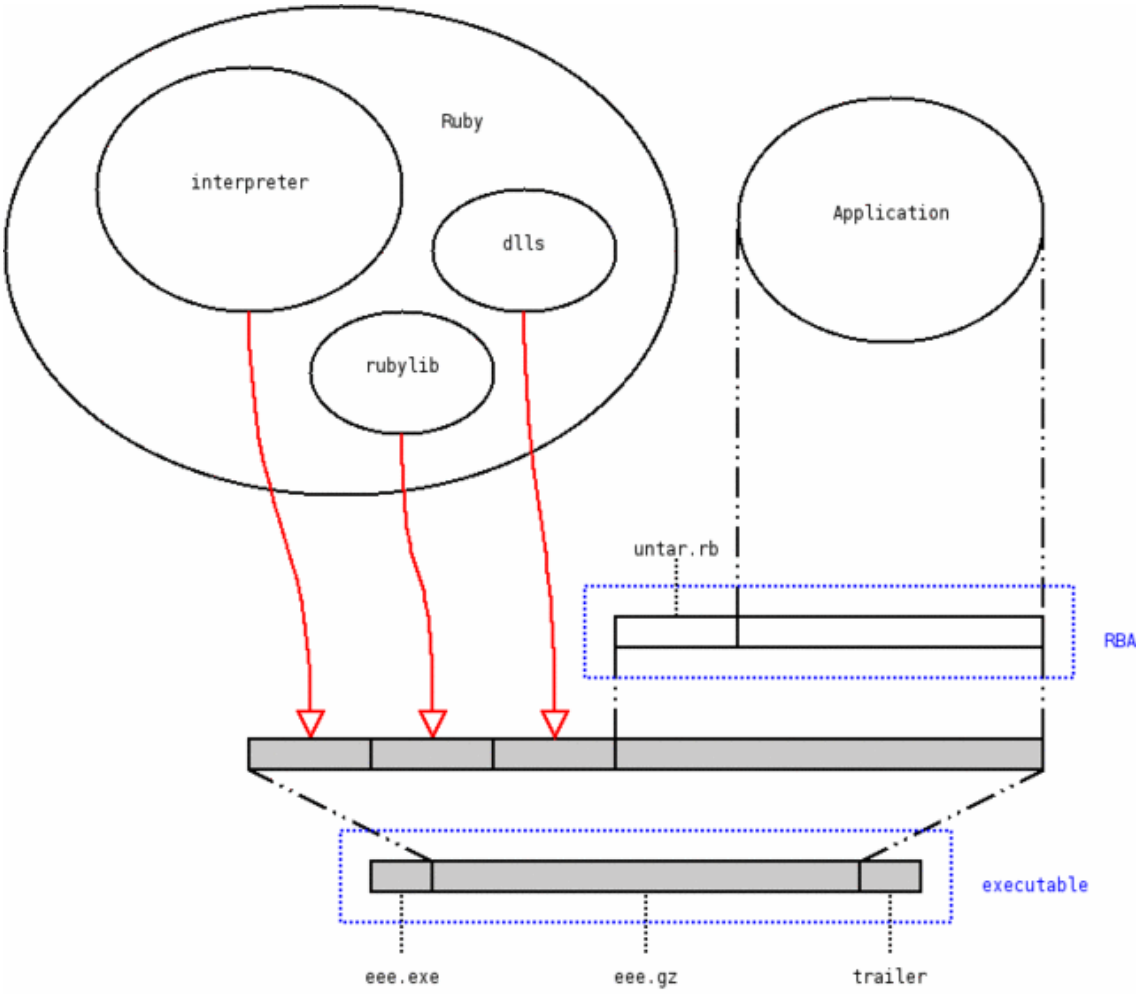
1. Create the application in a directory (and its subdirectories).
2. Convert this directory into an RBA (Ruby Archive) with Tar2RubyScript.
3. Convert this RBA into an executable with RubyScript2Exe.

I can distribute this executable to my customers as a standalone application. They don't need Ruby.

Customers don't need Ruby.

They don't need Ruby, because it's embedded in the executable, along with the application files and libraries and gems. Everything they need to run the application. To **run** the application. Not to install the application. This document describes a way to create an **`application.exe`**. Not a `setup.exe` or an `install.exe`.

(That's a lot of `exe`-s, but it works on Linux and Darwin, too.)


Ruby
interpreter
dlls
rubylib
Application
untar.rb
RBA
executable
eee.exe
eee.gz
trailer

2. Create the Application

An application is more than just a program or script. It consists of libraries, documentation, help files, configuration files, images, license, readme, and so on. Because a hard disk isn't one big directory, we traditionally group files in directories. When installing an application on a hard disk, we can do this grouping in two ways:

- Put all files of one type in one directory. All binaries in one place, all help files in another. That means that an application is scattered all over the place. That's the way it's done on Linux and most versions if Unix.
- Put all files that are part of one application in its own directory tree. Binaries, help files, configuration files, everything in one place. This time, if you want to search the help files for a term, you have to find those help files in the first place. In the Windows world, this is the preferred way of installing applications.

Without a good, uniform bookkeeping of your files ("Which files belong to which application?"), it's virtually impossible to uninstall or update an application. The one-app-one-dir approach gives us the housekeeping for free. Because Ruby has no uniform way of packing and distributing applications, putting all files of one application in its own directory tree seems like a safe bet.

User data has to be separated from the applications, under all circumstances. You don't want your data to be scattered all over the place!

Besides this one-app-one-dir approach, there's also the one-app-one-file approach. Like a ZIP-file or TAR-ball, but executable. It looks like an ideal way of keeping related files together. I'll return to this topic later on, because it turns out to be not only a theory: it has already been implemented. Anyway, you still need to build your application in one directory tree.

A couple of things you have to take care of when creating the application:

- The entry point of the application is `init.rb`. I'll explain this in "Pack the Application (the RBA)".
- Use only directories and files. No links, no permissions, no owners, no groups. All this is for being platform-independent as much as possible.
- Don't rely on shared libraries which are not part of the Ruby environment. Unless you are absolutely sure your customer has installed these libraries. For example: the Ruby bindings for TK are considered to be part of your Ruby environment, TK itself isn't. For that reason, I like RubyWebDialogs (pure Ruby!) and WxRuby (native widgets).

3. Pack the Application (the RBA)

The next step is packing it all into one file, which we can mail, or ship, or publish on the 'Net, or distribute to all the workstations. Traditionally, we put them in a ZIP-file or a TAR-ball, which has to be extracted manually by the customer. Not all customers know or want to know how to do that. They only want to run the application. The goal of most software companies is to reach as much people as possible: "More people" means "more money". So they make software for stupid customers and get rich. (It worked for Microsoft...) And open source developers want to make more people happy. To achieve this, we've created installers, e.g. NSIS.

I want to go back one step and ask this question: Do we really need to install an application before we can use it? We can rewrite this question: Why do we install an application, anyway? Mostly, because we have to recreate the environment in which the application has to live. We shouldn't bother our customer with recreating this environment. He just wants to double click on an icon. That's what he understands: "To start an application, you have to double click on the icon." If we manage to get our software down to this level, we can reach even more people. More people being happy, more money.

JAR's (Java Archives) are a common way in the Java universe to package applications, or libraries, or applets. It's one file, you can launch it by double clicking, it's sealed, it's simple. It's good. Can we do this with Ruby? Yes, we can. Do we have to change the runtime environment? No, we don't. But Java knows how to handle JAR's and Ruby doesn't know how to handle RBA's (Ruby Archives): We need extra tools! No we don't. And yes, it already exists. It's called Tar2RubyScript.

Tar2RubyScript transforms a directory tree, containing your application, into one single Ruby script, called the RBA (Ruby Archive), along with some code to handle this archive and start the application. This RBA can be distributed to our friends. When they've installed Ruby, they just double click, and your application is up and running! See the examples section of RubyWebDialogs for some demos.

To create the RBA:

```
$ ruby tar2rubyscript.rb application/
```

The result, `application.rb`, **is** the application and can run like any script:

```
$ ruby application.rb
```

Tar2RubyScript knows how to handle the embedded archive and is able to extract it to a temporary directory. After that, it simply loads `init.rb`, if present, in the newly created directory. This `init.rb` is our entry point.

4. Distribute the Application (the Executable)

Until now, we assumed that our customer has installed Ruby. Even worse: we assumed that he has installed the same libraries as we have. And the same gems. And that he has the same versions, or at least compatible versions. Well, that's a lot of assumptions. Unwelcome assumptions. It would be nice if we could assume that the customers hasn't installed Ruby at all. That's where RubyScript2Exe comes in.

RubyScript2Exe transforms the RBA (well, any Ruby script...) into a standalone Windows, Linux or 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 the application on an other machine: the RBA, the Ruby interpreter and the Ruby runtime library (stripped down for this application). Anyway, the result is the same: a standalone executable (*application*.exe). And that's what we want!

To create the executable:

```
$ ruby rubyscript2exe.rb application.rb
```

You can copy the result (*application*.exe on Windows, *application*_linux on Linux and *application*_darwin on Darwin) to another machine and run the application! Even on a machine without Ruby.

To list the contents of the executable and the contents of the RBA:

```
$ ./application_linux --eee-list
$ ./application_linux --tar2rubyscript-list
```

On Windows, it's:

```
c:\home\erik> application.exe --eee-list
c:\home\erik> application.exe --tar2rubyscript-list
```

5. Summary

To create the RBA and the executable:

```
$ ruby tar2rubyscript.rb application/
$ ruby rubyscript2exe.rb application.rb
```

To list the contents of the executable and the contents of the RBA:

```
$ ./application_linux --eee-list
$ ./application_linux --tar2rubyscript-list
```

Have a look at the sites of Tar2RubyScript and RubyScript2Exe for the usage and other detailed information of both products.

6. Tutorials

For now, I'm working on only one tutorial: Distributing Rails Applications.

7. Future

These sections are just theories. They are not not (yet) (fully) implemented and probably never will be. If you're satisfied with the previous sections and the executable, you can stop reading...

7.1. One Step

I'm contemplating the combination Tar2RubyScript and RubyScript2Exe in just one step:

```
$ ruby rubyscript2exe.rb application/
```

RubyScript2Exe already knows how to embed files and directories. Starting `ruby init.rb` in the given directory can't be too difficult...

7.2. Enriching the RBA

We've created an RBA and an executable. The RBA contains the application and the executable contains the RBA and everything else to run the application.

There's something wrong with this picture. We can't ship the RBA instead of the executable, unless our customer has installed the same libraries and gems as we have. At least the ones required or loaded by our application. And it would be nice if the versions were the same as well. Can't we just ship our libraries and gems along with our application to avoid these dependencies? Can't we just copy the necessary files from `sitelibdir` and `gemdir` into the RBA?

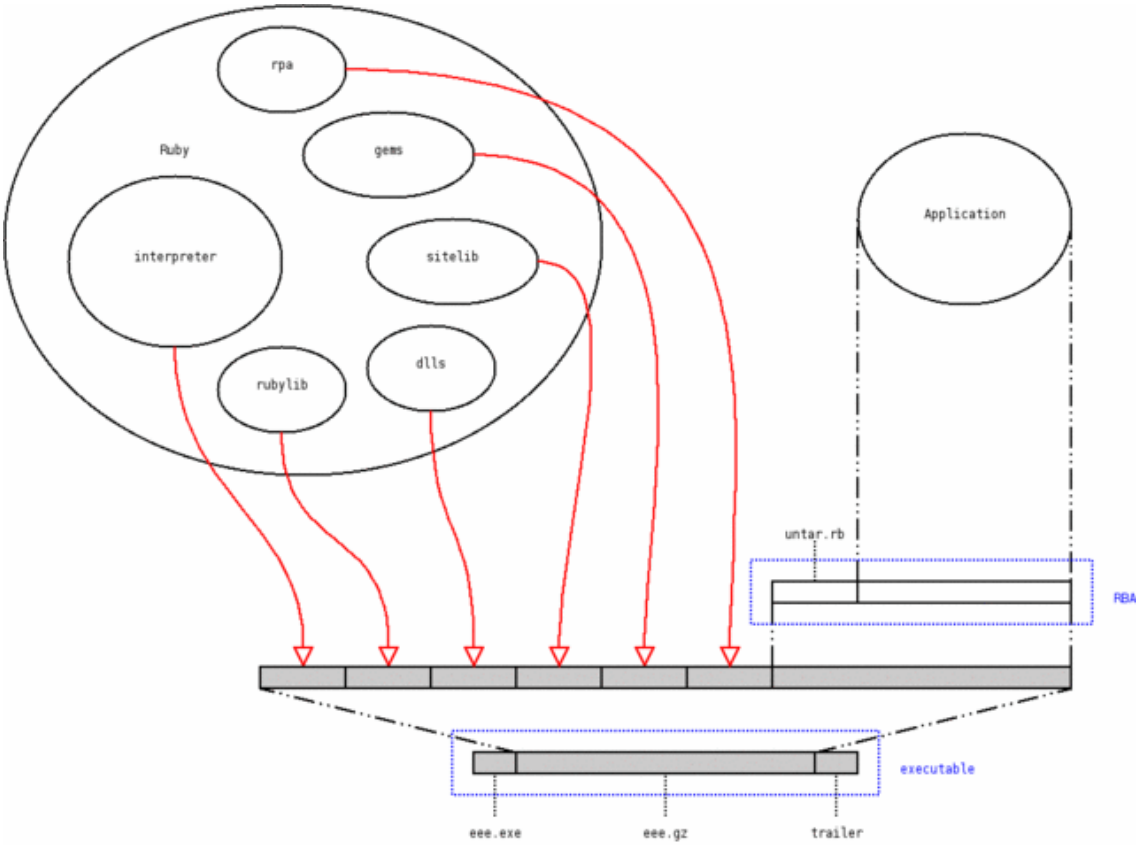
Yes we can, if Tar2RubyScript is able to detect and embed all necessary `sitelibdir`-files and `rubygem`-files (and their `dll`'s and `so`'s, determined recursively) and if RubyScript2Exe is able to detect and embed all necessary files except the `sitelibdir`-files and `rubygem`-files. Combining these features results in an enriched RBA and the same executable. The RBA can run on a plain Ruby installation and the executable can run on any machine (with the corresponding OS).

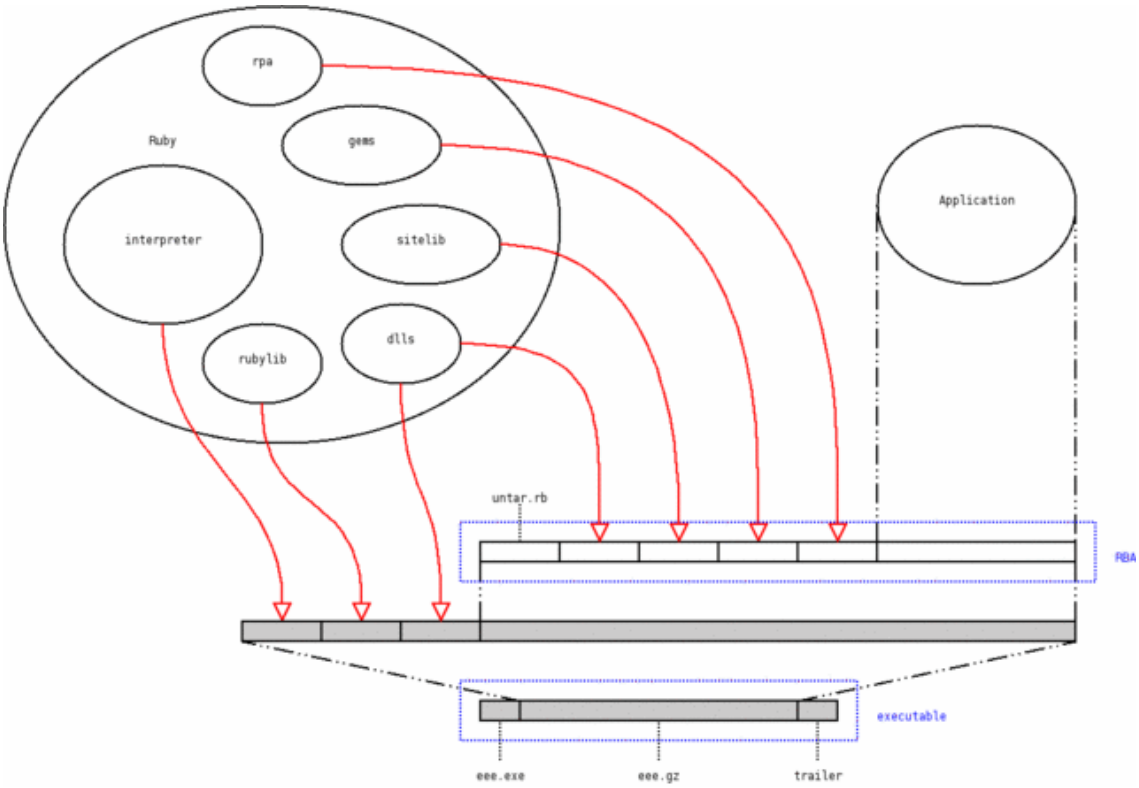
In an ideal world, this RBA contains just Ruby files and no `so`-files, so the application is platform agnostic, like Ruby itself. For example, the examples of RubyWebDialogs are perfectly enriched RBA's.

At the time of writing, Tar2RubyScript and RubyScript2Exe are not able to do this optimization.

The following two images show the differences.

7. Future


rpa
Ruby
gems
interpreter
sitelib
rubylib
dlls
Application
untar.rb
RBA
executable
eee.exe
eee.gz
trailer


rpa
Ruby
gems
interpreter
sitelib
rubylib
dlls
Application
untar.rb
RBA
executable
eee.exe
eee.gz
trailer

7. Future