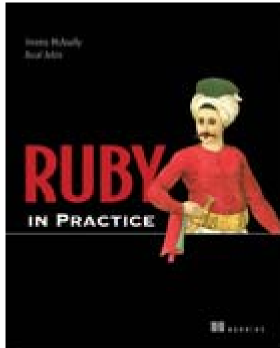
MANNING PUBLICATIONS

Using Plain-Text Files for Data Persistence

Excerpted from



Ruby in Practice
EARLY ACCESS EDITION

Jeremy McAnally and Assaf Arkin

MEAP Release: September 2007
Softbound print: August 2008 (est.) | 375 pages
ISBN: 1933988479

This article is based on chapter 9 from ***Ruby in Practice*** *by Jeremy McAnally and Assaf Arkin.*

Using YAML as a data persistence tool is an example of the more general case of using plain-text files for this purpose. You'll find other ways to do this in Ruby, such as the CSV (comma-separated values) facility in the standard library. XML files fall into this category, too.

None of these are full-fledged database systems. What they have in common with such systems, though, is the fact that they include information about the data, together with the data. CSV files don't contain *much* information about the data, but they do preserve ordering and, often, something like column or header information. XML preserves relationships in and among nested data structures; YAML does something similar, though with the avowed goal of being somewhat easier to read than XML, and more suitable for storing arbitrary Ruby data structures.

In looking at plain-text data storage techniques, then, we'll focus on YAML, as the one that offers the richest combination of complexity on the data end and editability on the text end.

Data persistence with YAML

YAML ("Yet Another Markup Language", or "YAML Ain't Markup Language", depending who you ask) is a data serialization format: Ruby objects go in, and a string representation comes out.

```
hash = { :one => 1, :two => 2, :colors => ["red","green","blue"] }
# => {:colors=>["red", "green", "blue"], :one=>1, :two=>2}
# require 'yaml'                #1
puts hash.to_yaml               #2
---
:colors:
- red
- green
- blue
:one: 1
:two: 2
```

YAML is part of the Ruby standard library: you just have to load it (#1) and then your objects can be serialized to YAML using the `to_yaml` method #2.

The strings generated by YAML conform to the YAML specification. YAML itself is not specific to Ruby; Ruby has an API for it, but so do numerous other languages.

Objects serialized to YAML can be read back into memory. Picking up from the last example:

```
y_hash = hash.to_yaml
# => "--- \n:colors: \n- red\n- green\n- blue\n:one: 1\n:two: 2\n"
new_hash = YAML.load(y_hash)
# => {:colors=>["red", "green", "blue"], :one=>1, :two=>2}
new_hash == hash
# => true
```

The YAML.load method can take either a string or an IO read handle as its argument, and deserializes the string or stream from YAML format into actual Ruby objects.

In the meantime, while objects are in serialized string form, you can edit them directly. In other words, YAML gives you a way to save data and also edit it in plain-text, human-readable form. (Part of the incentive behind the creation of YAML was to provide a plain-text format for representing nested data structures that wasn't quite as visually busy as XML.)

Problem

You need a way to automate the storage and retrieval of professional and personal contacts—an address book—but you want it to be in plain text so that you can edit the entries in a text editor as well as altering them programmatically.

Solution

We'll write some code that uses YAML, together with simple file IO operations, to provide a programmatic interface to a plain text file containing contact entries.

First things first: let's start with a test suite. Aside from the merits of writing tests in general, this will allow us to write examples of how the code should be used, before we've even written it. There's no better way to describe how you want an API to work than to write some tests that put it to use.

Listing 1, which you can place in a file called `contacts_y_test.rb` (the 'y' means this is for the YAML implementation), shows the class declaration and setup method for an appropriate test suite. The API for the contact code has already started to take shape. We'll create two classes: `Contact` and `ContactList`. The initializer for `Contact` will take the contact's name as the sole argument, and will yield the new `Contact` instance back to the block, where it can be used to set more values.

Listing 1: Class declaration and setup method for testing the contact code

```
require "test/unit"
require "contacts_y"                              #1
class TestContacts < Test::Unit::TestCase
  def setup
    @filename = "contacts"
    @list = ContactList.new(@filename)            #2

    @contact = Contact.new("Joe Smith")           #3
    joe.email = "joe@somewhere.abc"               #4
    joe.home[:street1] = "123 Main Street"        #5
    joe.home[:city] = "Somewhere"
    joe.work[:phone] = "(000) 123-4567"
    joe.extras[:instrument] = "Cello"

    @list << @contact                             #6
  end
```

In addition to test/unit, we'll load what will eventually be our implementation file: `contacts_y.rb` #1. After the loading preliminaries, a contact list gets instantiated along with a filename (#2), and a contact with a name. (#3). Beyond the name, the contact has an email address (#4), and several apparently deeper, hash-like data structures: home, work, extras. #5. The contact list object itself appears, not surprisingly, to have an array-like interface, judging by the appearance of the append operator. #6

Now it's time to write some tests for business logic. The setup up method inserts one contact object into the list. What about retrieving an object?

Listing 2: Testing the removal of a Contact object from a ContactList object

```
  def test_retrieve_contact_from_list             #1
    contact = @list["Joe Smith"]
    assert_equal("Joe Smith", contact.name)
  end

  def test_delete_contact_from_list               #2
    assert(!@list.empty?)
    @list.delete(@contact.name)
    assert(@list.empty?)
  end
end
```

Listing 2 shows a method that does exactly that. #1 It also includes a method that removes a contact from the list. #2 These two methods go in the test file, after the setup method. We'll also close out the class, so that we can write the implementation and get the test to pass.

Let's start with the `ContactList` class. We'll give each `ContactList` instance an array, in which it will store the actual contact objects. The business of the `ContactList` class itself will consist mostly of deciding what, and when, to pass along to this array: inserting contacts, removing contacts, and of course persisting contacts to a YAML file and reading contacts from a file.

The initial implementation of `ContactList` is shown in listing 3. (We're not using YAML yet, but we'll need it so it's being loaded.) Most of the action is in the `@contacts` array, which is expected to contain `Contact` objects. Listing 3 can be saved to `contacts_y.rb`.

Listing 3: Initial implementation of the ContactList class

```
require "yaml"
class ContactList
  attr_accessor :contacts

  def initialize(file)             #1
    @file = file
    @contacts = []
  end

  def <<(contact)                  #2
    @contacts << contact
  end

  def delete(name)                 #3
    @contacts.delete_if {|c| c.name == name }
  end

  def empty?                       #4
    @contacts.empty?
  end

  def size
    @contacts.size                 #5
  end

  def [](name)                     #6
    @contacts.find {|c| c.name == name }
  end
end
```

<#1 Create a new ContactList object, storing file name and creating @contacts array >
<#2 Add a contact to list, via @contacts array >
<#3 Delete a contact from list >
<#4 Test for emptiness, via @contacts array >
<#5 Size of list is size of @contacts array >
<#6 Fetch a contact by doing a name lookup in @contacts >

The `@contacts` list fields requests for array-like operations. Some of these operations work the way they do out of the box for any Ruby array (`empty?`, `size`, and `<<`). Others require special implementation to make sure they do the right thing for a list of contact objects; note that `remove` and `[]` use a name lookup to figure out which contact you want to operate on.

Next comes the implementation of the `Contact` class, which will be responsible for storing the contact information itself. We want to be able to store separate contact info for home and work. Each of these data sets will be stored as a hash, and accessed as an attribute of the contact object. The name and email properties will be separate, stored as individual attributes rather than parts of any of the hashes (see listing 4).

Listing 4: The Contact class we use to store contact records

```
class Contact
  attr_reader :name, :email, :home, :work, :extras
  attr_writer :name, :email
  def initialize(name)
    @name = name
    @home = {}
    @work = {}
    @extras = {}
  end
end
```

Now, run the tests:

```
$ ruby contacts_y_test.rb
Loaded suite contacts_y_test
Started
..
Finished in 0.000594 seconds.

2 tests, 3 assertions, 0 failures, 0 errors
```

Now let's get to the YAML side of things. We want a `ContactList` object to know how to save itself, in YAML format, to a file; and we want the `ContactList` class to know how to load a YAML file into a new `ContactList` instance. Listing 5 shows a test for these functions; you can paste this test into the existing test class.

Listing 5: Saving and loading a ContactList object

```
def test_save_and_load_list
  @list.save
  relist = ContactList.load(@filename)
  assert_equal(1,relist.size)
  contact = relist["Joe Smith"]
  assert_equal("Joe Smith", contact.name)
end
```

To get these new assertions to succeed, we have to add `save` and `load` methods to the `ContactList` class, as shown in listing 6.

Listing 6: Second set of methods for the ContactList class

```
def save
  File.open(@file, "w") do |fh|
    fh.puts(@contacts.to_yaml)                  #1
  end
end

def self.load(file)                             #2
  list = new(file)
  list.contacts = YAML.load(File.read(file))    #3
  list
end
```

These two methods can be pasted into the class definition for `ContactList`. (Note that load is a class method, hence defined directly on the class object `ContactList` (represented in context by "self". [#2])

It's in the `load` and `save` methods that you see the use of YAML. And it's very simple: When you want to save the list, you convert its `@contacts` array to YAML, and print the resulting string to the file. #2. When you want to read the list in, you use the load class method of YAML, passing it a string consisting of the contents of the file. #3 (You can also pass it an open `File` object.)

The new tests pass. And so, with rather little fanfare, we have data persistence. And running the tests will create a file called "contacts", containing the YAML representation of the Joe Smith contact. This brings us to the other side of the YAML coin: the ability to edit the YAML file itself. You do have to be a bit careful, because the YAML specification has rules you have to follow. But as long as you follow the YAML rules, you can make as many changes as you want to the file between reads.

Listing 7 shows the contacts file resulting from running the tests for the contact classes.

Listing 7: The contacts output file, in YAML format

```
---
- !ruby/object:Contact
  email: joe@somewhere.abc
  extras:
    :instrument: Cello
  home:
    :city: Somewhere
    :street1: 123 Main Street
  name: Joe Smith
  work:
    :phone: (000) 123-4567
```

You don't want to touch the first two lines, which are YAML's business. But you can change the values of the strings, or add more data, and all your changes will be happily absorbed into the in-memory `Contact` objects next time you use the `Contact` class.

Discussion

YAML provides easy serialization of objects to strings, and it's not much harder to save those strings to a file. Whether or not you decide it's technically correct to call this a database, it certainly has database-like properties. You don't *have* to edit your YAML files by hand; you can treat them as a black box. But it's nice to know that they're easy to edit.