Table of Contents

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

13. Databases and Persistence

We all want to leave behind something that will outlast us, and Ruby processes are no exception. Every program you write leaves some record of its activity, even if it's just data written to standard output. Most larger programs take this one step further: they store data from one run in a structured file, so that on another run they can pick up where they left off. There are a number of ways to persist data, from simple to insanely complex.

Simple persistence mechanisms like YAML let you write Ruby data structures to disk and load them back later. This is great for simple programs that don't handle much data. Your program can store its entire state in a disk file, and load the file on its next invocation to pick up where it left off. If you never keep more data than can fit into memory, the simplest way to make it permanent is to store it with YAML, Marshal, or Madeleine, and reload it later (see Recipes 13.1, 13.2, and 13.3). Madeleine also lets you revisit the *prior* states of your data.

If your dataset won't fit in memory, you need a database: a way of storing data on disk (usually in an indexed binary format) and retrieving parts of it quickly. The Berkeley database is the simplest database we cover: it operates like a hash, albeit a hash potentially much bigger than any you could keep in memory (Recipe 13.6).

But when most people think of a "database" they think of a relational database: MySQL, Postgres, Oracle, SQLite, or the like. A persistence mechanism stores data as Ruby data structures, and a Berkeley DB stores data as a hash of strings. But relational databases store data in the form of structured records with typed fields.

Because the tables of a relational database can have a complex structure and contain gigabytes of data, their contents are not accessed like normal Ruby data structures. Instead they're queried with SQL, a special programming language based on relational algebra. Most of the development time that goes into Ruby database libraries is spent trying to hide this fact. Several libraries hide the details of communication between a Ruby program and a SQL database; the balance of this chapter is devoted to showing how to use them.

Every relational database exposes a C API, and Ruby bindings to each API are available. We show you how to use the two most popular open source databases: MySQL (Recipe 13.9) and Postgres (Recipe 13.10).[1] But every database has different bindings, and speaks a slightly different variant of SQL. Fortunately, there are other libraries that hide these differences behind a layer of abstraction. Once you install the bindings, you can install abstraction layers atop them and rely on the abstraction layer to keep track of the differences between databases.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

[1] SQLite deserves an honorable mention because, unlike other relational databases, it doesn't require a server to run. The client code can directly query the database file. This makes things a lot easier to set up. Note that SQLite has two incompatible file formats (version 2 and version 3), and a gem exists for each version. You probably want the `sqlite3-ruby` gem.

Ruby's simplest database abstraction library is DBI (it's modeled after Perl's DBI module). It does nothing more than provide a uniform interface to the different database bindings. You still have to write all the SQL yourself (and if you're serious about database neutrality, you must use the lowest common denominator of SQL), but you only have to learn a single binding API.

The more popular database abstraction libraries are ActiveRecord (the library of choice for Rails applications) and Og. Not only do these libraries hide the differences between databases, they hide most of the actual SQL. The database tables are represented as Ruby classes, the rows in the database tables as instances of those classes. You can find, create, and modify database rows by manipulating normal-looking Ruby objects. Neither Og nor ActiveRecord can do everything that raw SQL can, so you may also need to use DBI or one of the database-specific bindings.

One standard argument for database abstraction layers is that they make it easy to switch an application's underlying database without having to rewrite all the code. They certainly do make this easier, but it almost never happens.[2] The real advantage is that with abstraction layers, you don't have to learn all the different database bindings. Even if you never change databases for any given project, throughout your career you'll find yourself using different databases on different projects. Learning how to use a database abstraction layer can save you from having to learn multiple database-specific bindings.

[2] What does happen is that you may write a product designed to work with whatever database the user has installed. You can't always require that your users run a specific database.

Whether you use ActiveRecord, Og, DBI, or database-specific bindings, you'll need an actual database for your code to connect to. The recipes in this chapter assume you've got a database called `cookbook` and that you connect to it with the username "cookbook_user" and the password "password".

Here's how to set up `cookbook` as a MySQL database:

```
$ mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.

Your MySQL connection id is 6 to server version: 4.0.24_Debian-10-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> create database cookbook;
Query OK, 1 row affected (0.00 sec)

mysql> grant all privileges on cookbook.* to 'cookbook_user'@'localhost' identified
by 'password';
Query OK, 0 rows affected (0.00 sec)
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Here's how to set `cookbook` up as a Postgres database (you'll probably need to run these commands as the `postgres` user):

```
$ createuser
Enter name of user to add: cookbook_user
Enter password for new user: password
Enter it again: password
Shall the new user be allowed to create databases? (y/n) y
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER

$ createdb cookbook
CREATE DATABASE
```

To avoid showing you the database connection code in every single recipe, we've factored it out into a library. If you want to run the code in this chapter's recipes, you should put the following code in a file called `cookbook_dbconnect.rb`. Keep it in the directory where you keep the recipe code, or somewhere in your library include path, so that `require 'cookbook_dbconnect'` will work.

This file defines database connection functions for DBI, ActiveRecord, and Og:

```
# cookbook_dbconnect.rb
require 'rubygems'
require 'dbi'
require 'active_record'
require 'og'
```

The `with_db` method gets a database connection through DBI and runs a code block in the context of that connection:

```
def with_db
  DBI.connect("dbi:Mysql:cookbook:localhost",
              "cookbook_user", "password") do |c|
    yield c
  end
end
```

The `activerecord_connect` method only needs to be called once at the beginning of a program: after that, ActiveRecord will acquire database connections as needed.

```
def activerecord_connect
  ActiveRecord::Base.establish_connection(:adapter => "mysql",
                                          :host => "localhost",
                                          :username => "cookbook_user",
                                          :password => "password",
                                          :database => "cookbook")
end
```

For your reference, this table presents the ActiveRecord adapter names for various kinds of databases.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Table 13-1.

Database	Adapter name
MySQL	mysql
PostgreSQL	postgresql
Oracle	oci
Microsoft SQL Server	sqlserver
SQLite 2	sqlite
SQLite 3	sqlite3
DB2	db2

The `og_connect` also needs to be called only once. One caveat: you must call it *after* you've defined the classes for your Og data model.

```
def og_connect
  Og.setup( { :destroy => false,
              :store => :mysql,
              :user => "cookbook_user",
              :password => "password",
              :name => "cookbook" } )
end
```

This version of `cookbook_dbconnect` assumes you're running against a MySQL database. For a different database, you just need to change the database name so that DBI, ActiveRecord, and Og know which adapter they should use.

Here are some resources for more information about databases in Ruby:

- http://ruby-dbi.rubyforge.org/
- http://www.rubyonrails.org/show/ActiveRecord
- http://www.rubygarden.com/index.cgi/Libraries/og_tutorial.rdoc

Recipe 13.1. Serializing Data with YAML

Problem

You want to serialize a data structure and use it later. You may want to send the data structure to a file, then load it into a program written in a different programming language.

Solution

The simplest way is to use the built-in `yaml` library. When you `require yaml`, all Ruby objects sprout `to_yaml` methods that convert them to the YAML serialization format. A YAML string is human-readable, and it intuitively corresponds to the object from which it was derived:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
require 'yaml'

10.to_yaml               # => "--- 10\n"
'ten'.to_yaml            # => "--- ten\n"
'10'.to_yaml             # => "--- \"10\"\n"
```

Arrays are represented as bulleted lists:

```
puts %w{Brush up your Shakespeare}.to_yaml
# --
# - Brush
# - up
# - your
# - Shakespeare
```

Hashes are represented as colon-separated key-value pairs:

```
puts ({ 'star' => 'hydrogen', 'gold bar' => 'gold' }).to_yaml
# --
# star: hydrogen
# gold bar: gold
```

More complex Ruby objects are represented in terms of their classes and member variables:

```
require 'set'
puts Set.new([1, 2, 3]).to_yaml
# --- !ruby/object:Set
# hash:
#   1: true
#   2: true
#   3: true
```

You can dump a data structure to a file with `YAML.dump`, and load it back with `YAML.load`:

```
users = [{:name => 'Bob', :permissions => ['Read']},
         {:name => 'Alice', :permissions => ['Read', 'Write']}]

# Serialize
open('users', 'w') { |f| YAML.dump(users, f) }

# And deserialize
users2 = open("users") { |f| YAML.load(f) }
# => [{:permissions=>["Read"], :name=>"Bob"},
#     {:permissions=>["Read", "Write"], :name=>"Alice"}]
```

YAML implementations are available for Perl, Python, Java, PHP, JavaScript, and OCaml, so if you stick to the "standard" data types (strings, arrays, and so on), the serialized file will be portable across programming languages.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

If you've ever used Python's `pickle` module or serialized a Java object, you know how convenient it is to be able to dump an object to disk and load it back later. You don't have to define a custom data format or write an XML generator: you just shove the object into a file or a database, and read it back later. The only downside is that the serialized file is usually a binary mess that can only be understood by the serialization library.

YAML is a human-readable and somewhat cross-language serialization standard. Its format describes the simple data structures common to all modern programming languages. YAML can serialize and deserialize any combination of strings, booleans, numbers, dates and times, arrays (possibly nested arrays), and hashes (again, possibly nested ones).

You can also use `YAML` to serialize Ruby-specific objects: symbols, ranges, and regular expressions. Indeed, you can use YAML to serialize instances of custom classes: YAML serializes the class of the object and the values of its instance variables. There's no guarantee, though, that other programming languages will understand what you mean.[3]

[3] Ruby can also read YAML descriptions of Perl's regular expressions.

Not only is YAML human-readable, it's human-writable. You can write YAML files in a text editor and load them into Ruby as objects. If you're having trouble with the YAML representation of a particular data structure, your best bet is to define a simple version of that data structure in an `irb` session, dump it to YAML, and work from there.

```
quiz_question = ['What color is Raedon?', ['Blue', 'Albino', '*Yellow']]
puts quiz_question.to_yaml
# --
# - What color is Raedon?
# - - Blue
#   - Albino
#   - "*Yellow"
```

Before you get drunk with power, you should know that YAML shares the limitations of other serialization schemes. Most obviously, you can only deserialize objects in an environment like the one in which you serialized them. Suppose you convert a `Set` object to YAML in one Ruby session:

```
require 'yaml'
require 'set'
set = Set.new([1, 2, 3])
open("set", "w") { |f| YAML.dump(set, f) }
```

In another Ruby session, you might try to convert the YAML back into a `Set`, without first requiring the `set` library:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# Bad code -- don't try this!
require 'yaml'
set = open("set") { |f| YAML.load(f) }
# => #<YAML::Object:0xb7bd8620 @ivars={"hash"=>{1=>true, 2=>true, 3=>true}},
#      @class="Set">
```

Instead of a `Set`, you've got an unresolved object of class `YAML::Object`. The set has been loaded from the file and deserialized, but Ruby can't resolve its class name.

YAML can only serialize data; it can't serialize Ruby code or system resources (such as filehandles or open sockets). This means some objects can't be fully converted to YAML. The following code successfully serializes and deserializes a `File` object, but the deserialized `File` isn't open and doesn't point to anything in particular:

```
handle = open('a_file', 'w')
handle.path
# => "a_file"

handle2 = YAML.load(YAML.dump(handle))
# => #<File:0xb7bd9a58>
handle2.path
# IOError: uninitialized stream
```

The essence of the `File` object—its handle to a file on disk, granted by the operating system—has been lost.

Objects that contain Ruby code will lose their code when dumped to YAML. This means that `Proc` and `Binding` objects will turn up empty. Objects with singleton methods will be dumped without them. Classes can't be dumped to YAML at all.

But these are all edge cases. Most data structures, even complex ones, can be serialized to YAML and stay readable to boot.

See Also

- Ruby standard library documentation for the `yaml` library
- The YAML web page (http://www.yaml.org/)
- Recipe 12.12, "Reading and Writing Configuration Files"
- An episode of the Ruby Quiz focused on creating a serializable `Proc` object (http://www.rubyquiz.com/quiz38.html)

Recipe 13.2. Serializing Data with Marshal

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

You want to serialize a data structure to disk faster than YAML can do it. You don't care about the readability of the serialized data structure, or portability to other programming languages.

Solution

Use the `Marshal` module, built into Ruby. It works more or less like `YAML`, but it's much faster. The `Marshal.dump` method transforms a data structure into a binary string, which you can write to a file and reconstitute later with `Marshal.load`.

```
Marshal.dump(10)                            # => "\004\010i\017"
Marshal.dump('ten')                         # => "\004\010\"\010ten"
Marshal.dump('10')                          # => "\004\010\"\a10"

Marshal.load(Marshal.dump(%w{Brush up your Shakespeare}))
# => ["Brush", "up", "your", "Shakespeare"]

require 'set'
Marshal.load(Marshal.dump(Set.new([1, 2, 3])))
# => #<Set: {1, 2, 3}>
```

Discussion

`Marshal` is what most programmers coming from other languages expect from a serializer. It's fast (much faster than `yaml`), and it produces unreadable blobs of binary data. It can serialize almost anything that `yaml` can (see Recipe 13.1 for examples), and it can also handle a few cases that `yaml` can't. For instance, you can use `Marshal` to serialize a reference to a class:

```
Marshal.dump(Set)                 # =>"\004\010c\010Set"
```

Note that the serialized version of `Set` is little more than a reference to the class. Like YAML, Marshal depends on the presence of the original classes, and you can't deserialize a reference to a class you don't have.[4] With YAML, you'll get an unresolved `YAML::Object`; with Marshal, you get an `ArgumentError`:

[4] This also means that if you add methods to a class, then serialize the class, your methods don't get saved.

```
#!/usr/bin/ruby -w

Marshal.load("\004\010c\010Set")
# ArgumentError: undefined class/module Set
```

Like YAML, Marshal only serializes data structures. It can't serialize Ruby code (like `Proc` objects), or resources allocated by other processes (like filehandles or database connections). However, the two libraries differ in their error handling. YAML tends to

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

serialize as much as it can: it can serialize a `File` object, but when you deserialize it, you get an object that doesn't point to any actual file. Marshal just gives you an error when you try to serialize a file:

```
open('output', 'w') { |f| Marshal.dump(f) }
# TypeError: can't dump File
```

See Also

- Recipe 13.1, "Serializing Data with YAML," has more on serialization in general

Recipe 13.3. Persisting Objects with Madeleine

Problem

You want to store objects in RAM and persist them between independent executions of the program. This will let your program recall its state indefinitely and access it very quickly.

Solution

Use the Madeleine library available as the `madeleine` gem. It transparently persists any Ruby object that can be serialized with `Marshal`. Unlike a conventional database persistence layer, Madeleine keeps all of its objects in RAM at all times.

To use Madeleine, you have to decide which objects in your system need to be serialized, and which ones you might have saved to a database traditionally. Here's a simple Madeleine-backed program for conducting yes/no polls, in which agreement adds one to a total and disagreement subtracts one:

```
#!/usr/bin/ruby -w
# poll.rb
require 'rubygems'
require 'madeleine'

class Poll
  attr_accessor :name
  attr_reader :total

  def initialize(name)
    @name = name
    @total = 0
  end

  def agree
    @total += 1
  end

  def disagree
    @total -= 1
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
  end
end
```

So far there's been no Madeleine code, just a normal class with instance variables and accessors. But how will we store the state of the poll between invocations of the polling program? Since instances of the `Poll` class can be serialized with `Marshall`, we can wrap a `Poll` object in a `MadeleineSnapshot`, and keep it in a file:

```
poll = SnapshotMadeleine.new('poll_data') do
  Poll.new('Is Ruby great?')
end
```

The `system` accessor retrieves the object wrapped by `MadeleineSnapshot`:

```
if ARGV[0] == 'agree'
  poll.system.agree
elsif ARGV[0] == 'disagree'
  poll.system.disagree
end

puts "Name: #{poll.system.name}"
puts "Total: #{poll.system.total}"
```

You can save the current state of the object with `take_snapshot`:

```
poll.take_snapshot
```

Here are a few sample runs of the `poll.rb` program:

```
$ ruby poll.rb agree
Name: Is Ruby great?
Total: 1

$ ruby poll.rb agree
Name: Is Ruby great?
Total: 2

$ ruby poll.rb disagree
Name: Is Ruby great?
Total: 1
```

Discussion

Recall this piece of code:

```
poll = SnapshotMadeleine.new('poll_data') do
  Poll.new('Is Ruby great?')
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The first time that code is run, Madeleine creates a directory called `poll_data`. Then it runs the code block. The result of the code block is the object whose state will be tracked in the `poll_data` directory.

On subsequent runs, the `poll_data` directory already exists, and Madeleine loads the current state of the `Poll` object from the latest snapshot in the directory. It doesn't run the code block.

Here are the contents of `poll_data` after we run the program three times:

```
$ ls poll_data
00000000000000000001.snapshot
00000000000000000002.snapshot
00000000000000000003.snapshot
```

Every time we call `poll.take_snapshot`, Madeleine serializes the `Poll` object to a snapshot file in `poll_data`. If the data ever gets corrupted, you can remove the corrupted snapshot files and revert to a previous version of the data.

A clever trick for programs like our poll application is to use `Kernel#at_exit` to automatically save the state of an object when the program ends. This way, even if your program is killed by a Unix signal, or throws an exception, your data will be saved.[5]

[5] Of course, these things might happen when your data is in an inconsistent state and you don't *want* it to be saved.

```
at_exit { poll.take_snapshot }
```

In applications where a process runs indefinitely, you can save snapshots at regular intervals by spawning a separate thread:

```
def save_recurring_snapshots(madeleine_object, time_interval)
  loop do
    madeleine_object.take_snapshot
    sleep time_interval
  end
end

Thread.new { save_recurring_snapshots(poll, 24*60*60) }
```

See Also

- Recipe 3.12, "Running a Code Block Periodically"
- Recipe 13.2, "Serializing Data with Marshal"
- The Madeleine design rules document lays out the conditions your code must meet if you want to snapshot it with Madeleine (http://madeleine.sourceforge.net/docs/designRules.html)

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

- The RDoc documentation for Madeleine (http://madeleine.sourceforge.net/docs/api/)
- For more on the technique of object prevalence, see the web site for the Prevayler Java project, especially the "Articles" section (http://www.prevayler.org/wiki.jsp)

Recipe 13.4. Indexing Unstructured Text with SimpleSearch

Problem

You want to index a number of texts and do quick keyword searches on them.

Solution

Use the SimpleSearch library, available in the `SimpleSearch` gem.

Here's how to create and save an index:

```
require 'rubygems'
require 'search/simple'

contents = Search::Simple::Contents.new
contents << Search::Simple::Content.
             new('In the beginning God created the heavens…',
                 'Genesis.txt', Time.now)
contents << Search::Simple::Content.new('Call me Ishmael…',
                                        'MobyDick.txt', Time.now)
contents << Search::Simple::Content.new('Marley was dead to begin with…',
                                        'AChristmasCarol.txt', Time.now)

searcher = Search::Simple::Searcher.load(contents, 'index_file')
```

Here's how to load and search an existing index:

```
require 'rubygems'
require 'search/simple'

searcher = nil
open('index_file') do |f|
  searcher = Search::Simple::Searcher.new(Marshal.load(f), Marshal.load(f),
                                          'index_file')
end

searcher.find_words(['begin']).results.collect { |result| result.name }
# => ["AChristmasCarol.txt", "Genesis.txt"]
```

Discussion

SimpleSearch is a library that makes it easy to do fast keyword searching on unstructured text documents. The index itself is represented by a `Searcher object`, and each document you feed it is a `Content` object.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

To create an index, you must first construct a number of `Content` objects and a `Contents` object to contain them. A `Content` object contains a piece of text, a unique identifier for that text (often a filename, though it could also be a database ID or a URL), and the time at which the text was last modified. `Searcher.load` transforms a `Contents` object into a searchable index that gets serialized to disk with Marshal.

The indexer analyzes the text you gives it, removes stop words (like "a"), truncates words to their roots (so "beginning" becomes "begin"), and puts every word of the text into binary data structures. Given a set of words to find and a set of words to exclude, SimpleSearch uses these structures to quickly find a set of documents.

Here's how to add some new documents to an existing index:

```
class Search::Simple::Searcher
  def add_contents(contents)
     Search::Simple::Searcher.create_indices(contents, @dict,
                                             @document_vectors)
     dump                              # Re-serialize the file
  end
end

contents = Search::Simple::Contents.new
contents << Search::Simple::Content.new('A spectre is haunting Europe…',
                                        'TheCommunistManifesto.txt', Time.now)
searcher.add_contents(contents)
searcher.find_words(['spectre']).results[0].name
# => "TheCommunistManifesto.txt"
```

SimpleSearch doesn't support incremental indexing. If you update or delete a document, you must recreate the entire index from scratch.

See Also

- The SimpleSearch home page (http://www.chadfowler.com/SimpleSearch/)
- The sample application within the `SimpleSearch` gem: `search-simple.rb`
- Recipe 13.2, "Serializing Data with Marshal"
- For a more sophisticated indexer, see Recipe 13.5, "Indexing Structured Text with Ferret"

Recipe 13.5. Indexing Structured Text with Ferret

Problem

You want to perform searches on structured text. For instance, you might want to search just the headline of a news story, or just the body.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

The Ferret library can tokenize and search structured data. It's a pure Ruby port of Java's Lucene library, and it's available as the `ferret` gem.

Here's how to create and populate an index with Ferret. I'll create a searchable index of useful Ruby packages, stored as a set of binary files in the `ruby_packages/` directory.

```
require 'rubygems'
require 'ferret'

PACKAGE_INDEX_DIR = 'ruby_packages/'
Dir.mkdir(PACKAGE_INDEX_DIR) unless File.directory? PACKAGE_INDEX_DIR
index = Ferret::Index::Index.new(:path => PACKAGE_INDEX_DIR,
                                 :default_search_field => 'name|description')
index << { :name => 'SimpleSearch',
           :description => 'A simple indexing library.',
           :supports_structured_data => false,
           :complexity => 2 }
index << { :name => 'Ferret',
           :description => 'A Ruby port of the Lucene library.
                            More powerful than SimpleSearch',
           :supports_structured_data => true,
           :complexity => 5 }
```

By default, queries against this index will search the "name" and "description" fields, but you can search against any field:

```
index.search_each('library') do |doc_id, score|
  puts index.doc(doc_id).field('name').data
end
# SimpleSearch
# Ferret

index.search_each('description:powerful AND supports_structured_data:true') do
|doc_id, score|
  puts index.doc(doc_id).field("name").data
end
# Ferret

index.search_each("complexity:<5") do |doc_id, score|
  puts index.doc(doc_id).field("name").data
end
# SimpleSearch
```

Discussion

When should you use Ferret instead of SimpleText? SimpleText is good for unstructured data like plain text. Ferret excels at searching structured data, the kind you find in databases.

Relational databases are good at finding exact field matches, but not very good at locating keywords within large strings. Ferret works best when you need full text search but you want to keep some of the document structure. I've also had great success using Ferret[6] to bring together data from disparate sources (some in databases, some not) into one structured, searchable index.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

[6] Actually, I was using Lucene. Same idea.

There are two things you can do with Ferret: add text to the index, and query the index. Ferret offers you a lot of control over both activities. I'll briefly cover the most interesting features.

You can feed an index by passing in a hash of field names to values, or you can feed it fully formed `Ferret::Document` objects. This gives you more control over which fields you'd like to index. Here, I'll create an index of news stories taken from a hypothetical database:

```
# This include will cut down on the length of the Field:: constants below.
include Ferret::Document

def index_story(index, db_id, headline, story)
  doc = Document.new
  doc << Field.new("db_id", db_id, Field::Store::YES, Field::Index::NO)
  doc << Field.new("headline", headline, Field::Store::YES, Field::Index::TOKENIZED)
  doc << Field.new("story", story, Field::Store::NO, Field::Index::TOKENIZED)
  index << doc
end

STORY_INDEX_DIR = 'news_stories/'
Dir.mkdir(STORY_INDEX_DIR) unless File.directory? STORY_INDEX_DIR
index = Ferret::Index::Index.new(:path => STORY_INDEX_DIR)

index_story(index, 1, "Lizardoids Control the Media, Sources Say",
            "Don't count on reading this story in your local paper anytime
             soon, because …")

index_story(index, 2, "Where Are My Pants? An Editorial",
            "This is an outrage. The lizardoids have gone too far! …")
```

In this case, I'm storing the database ID in the `Document`, but I'm not indexing it. I don't want anyone to search on it, but I need some way of tying a `Document` in the index to a record in the database. That way, when someone does a search, I can print out the headline and provide a link to the original story.

I treat the body of the story exactly the opposite way: the words get indexed, but the original text is not stored and can't be recovered from the `Document` object. I'm not going to be displaying the text of the story along with my search results, and the text is already in the database, so why store it again in the index?

The simplest way to search a Ferret index is with `Index#search_each`, as demonstrated in the Solution. This takes a query and a code block. For each document that matched the search query, it yields the document ID and a number between 0 and 1, representing the quality of the match.

You can get more information about the search results by calling `search` instead of `search_each`. This gives you a `Ferret::Search::TopDocs` object that contains the search results, as well as useful information like how many documents were matched. Call `each` on a `TopDocs` object and it'll act just as if you'd called `search_each`.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Here's some code that does a search and prints the results:

```
def search_news(index, query)
  results = index.search(query)
  puts "#{results.size} article(s) matched:"

  results.each do |doc_id, score|
    story = index.doc(doc_id)
    puts " #{story.field("headline").data} (score: #{score})"
    puts " http://www.example.com/news/#{story.field("db_id").data}"
    puts
  end
end

search_news(index, "pants editorial")
# 1 article(s) matched:
#  Where Are My Pants? An Editorial (score: 0.0908329636861293)
# http://www.example.com/news/2
```

You can weight the fields differently to fine-tune the results. This query makes a match in the headline count twice as much as a match in the story:

```
search_news(index, "headline:lizardoids^1 OR story:lizardoids^0.5")
# 2 article(s) matched:
#  Lizardoids Control the Media, Sources Say (score: 0.195655948031232)
#  http://www.example.com/news/1
#
#  Where Are My Pants? An Editorial (score: 0.0838525491562421)
#  http://www.example.com/news/2
```

Queries can be strings or `Ferret::Search::Query` objects. Pass in a string, and it just gets parsed and turned into a `Query`. The main advantage of creating your own `Query` objects is that you can put a user-friendly interface on your search functionality, instead of making people always construct Ferret queries by hand. The `weighted_query` method defined below takes a single keyword and creates a `Query` object equivalent to the rather complicated weighted query given above:

```
def weighted_query(term)
  query = Ferret::Search::BooleanQuery.new
  query << term_clause("headline", term, 1)
  query << term_clause("story", term, 0.5)
end

def term_clause(field, term, weight)
  t = Ferret::Search::TermQuery.new(Ferret::Index::Term.new(field, term))
  t.boost = weight
  return Ferret::Search::BooleanClause.new(t)
end
```

Ferret can be clumsy to use. It's got a lot of features to learn, and sometimes it seems like you spend all your time composing small objects into bigger objects (as in `weighted_query` above, which creates instances of four different classes). This is partly because Ferret is so flexible, and partly because the API comes mainly from Java. But nothing else works as well for searching structured text.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- The Ferret homepage (http://ferret.davebalmain.com/)
- The Ferret Query Language, described in the RDoc for the `QueryParser` class (http://ferret.davebalmain.com/api/classes/Ferret/QueryParser.html)
- Apache Lucene, the basis for Ferret, lives at http://lucene.apache.org/java/

Recipe 13.6. Using Berkeley DB Databases

Problem

You want a simple, fast database that doesn't need a server to run.

Solution

Ruby's standard `dbm` library lets you store a database in a set of standalone binary files. It's not a SQL database: it's more like a fast disk-based hash that only stores strings.

```
require 'dbm'

DBM.new('random_thoughts') do |db|
  db['tape measure'] =
    "What if there was a tape measure you could use as a yo-yo?"
  db[23] = "Fnord."
end

DBM.open('random_thoughts') do |db|
  puts db['tape measure']
  puts db['23']
end
# What if there was a tape measure you could use as a yo-yo?
# Fnord.

DBM.open('random_thoughts') { |db| db[23] }
# TypeError: can't convert Fixnum into String

Dir['random_thoughts.*']
# => ["random_thoughts.pag", "random_thoughts.dir"]
```

Discussion

The venerable Berkeley DB format lets you store enormous associative datasets on disk and quickly access them by key. It dates from before programming languages had built-in hash structures, so it's not as useful as it used to be. In fact, if your hash is small enough to fit in memory, it's faster to simply use a Ruby hash that you serialize to disk with `Marshal`.

If you do need to use a `DBM` object, you can treat it almost exactly like a Ruby hash: it supports most of the same methods.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

There are many, many implementations of the Berkeley DB, and the file formats differ widely between versions, so DBM files are not very portable. If you're creating your own databases, you should use the generic `dbm` library. It provides a uniform interface to all the DBM implementations, using the best library you have installed on your computer.[7]

[7] Actually, it uses the best DBM library you had installed when you installed the `dbm` Ruby extension.

Ruby also provides `gdbm` and `sdbm` libraries, interfaces to specific database formats, but you should only need these if you're trying to load a Berkeley DB file produced by some other program.

There's also the SleepyCat library, a more ambitious implementation of the Berkeley DB that implements features of traditional databases like transactions and locking. Its Ruby bindings are available as a third-party download. It's still much closer to a disk-based data structure than to a relational database, and the basic interface is similar to that of `dbm`, though less Ruby-idiomatic:

```
require 'bdb'

db = BDB::Hash.create('random_thoughts2.db', nil, BDB::CREATE)
db['Why do we park on a driveway but'] = 'it never rains but it pours.'
db.close

db = BDB::Hash.open('random_thoughts2.db', nil, 'r')
db['Why do we park on a driveway but']
# => "it never rains but it pours."
db.close
```

The SleepyCat library provides several different hashlike data structures. If you want a hash whose keys stay sorted alphabetically, you can create a `BDB::Btree` instead of a `BDB::Hash`:

```
db = BDB::Btree.create('element_reviews.db', nil, BDB::CREATE)
db['earth'] = 'My personal favorite element.'
db['water'] = 'An oldie but a goodie.'
db['air'] = 'A good weekend element when you're bored with other elements.'
db['fire'] = 'Perhaps the most overrated element.'

db.each { |k,v| puts k }
# air
# earth
# fire
# water

db['water']                             # => "An oldie but a goodie."
db.close
```

See Also

- On Debian GNU/Linux, the DBM extensions to Ruby come in separate packages from Ruby itself: `libdbm-ruby, libgdbm-ruby, and libsdbm-ruby`

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

- You can get the Ruby binding to the Sleepycat library at http://moulon.inra.fr/ruby/bdb.html
- Confused by all the different, mutually incompatible implementations of the Berkeley DB idea? Try reading "Unix Incompatibility Notes: DBM Hash Libraries" (http://www.unixpapa.com/incnote/dbm.html)
- If you need a *relational* database that doesn't require a server to run, try SQLite: it keeps its databases in standalone files, and you can use it with ActiveRecord or DBI; its Ruby binding is packaged as the `sqlite3-ruby` gem, and its home page is at http://www.sqlite.org/

Recipe 13.7. Controlling MySQL on Unix

Problem

The standard Ruby database interfaces assume you're connecting to a preexisting database, and that you already have access to this database. You want to create and administer MySQL databases from within Ruby.

Solution

Sam Ruby came up with an elegant solution to this problem. The `mysql` method defined below opens up a pipe to a MySQL client program and sends SQL input to it:

```
def mysql(opts, stream)
  IO.popen("mysql #{opts}", 'w') { |io| io.puts stream }
end
```

You can use this technique to create, delete, and administer MySQL databases:

```
mysql '-u root -p[password]', <<-end
  drop database if exists website_db;
  create database website_db;
  grant all on website_db.* to #{`id -un`.strip}@localhost;
end
```

Discussion

This solution looks so elegant because of the `<<-end` declaration, which allows you to end the string the same way you end a code block.

One shortcoming of this solution is that the `IO.popen` call opens up a one-way communication with the MySQL client. This makes it difficult to call SQL commands and get the results back. If that's what you need, you can use `IO.popen` interactively; see Recipe 23.1.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- Recipe 23.1, "Scripting an External Program"

Recipe 13.8. Finding the Number of Rows Returned by a Query

Problem

Writing a `DBI` program, you want an efficient way to see how many rows were returned by a query.

Solution

A `do` command returns the number of rows affected by the command, so that one's easy. To demonstrate, I'll create a database table that keeps track of my prized collection of lowercase letters:

```
require 'cookbook_dbconnect'

with_db do |c|

   c.do %{drop table if exists letters}

  c.do %{create table letters(id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
                              letter CHAR(1) NOT NULL)}
  letter_sql = ('a'..'z').collect.join('"),("')

  c.do %{insert into letters(letter) values ("#{letter_sql}")}
end
# => 26
```

When you execute a query, you get back a `StatementHandle` object representing the request. If you're using a MySQL database, you can call `rows` on this object to get the number of rows in the result set:

```
vowel_query = %{select id from letters where letter in ("a","e","i","o","u")}
with_db do |c|
  h = c.execute vowel_query
  "My collection contains #{h.rows} vowels."
end
# => "My collection contains 5 vowels."
```

If you're not using MySQL, things are a bit trickier. The simplest thing to do is simply retrieve all the rows as an array, then use the array's size as the number of rows:

```
with_db do |c|
  vowels = c.select_all(vowel_query)
  "My collection still contains #{vowels.size} vowels."
end
# => "My collection still contains 5 vowels."
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

But this can be disastrously inefficient; see below for details.

Discussion

When you select some items out of a Ruby array, say with `Array#grep`, Ruby gives you the results in a brand new array. Once the array has been created, there's no cost to checking its size by calling `Array#size`.

A database query acts differently. Your query might have matched millions of rows, and each result might contain kilobytes of data. This is why normally you iterate over a result set instead of using `select_all` to get it as an array. Getting the whole result set at once might use a huge amount of memory, which is why using `select_all` can be disastrous.

You've got two other options. If you're going to be iterating over the entire dataset anyway, *and* you don't need the count until you're all done, you can count the rows as you go. This will save memory over the `fetch_all` approach:

```
with_db do |c|
  rows = 0

  c.execute(vowel_query).each do |row|
    rows += 1
    # Process the row…
  end
  "Yup, all #{rows} vowels are still there."
end
# => "Yup, all 5 vowels are still there."
```

Otherwise, your only choice is to run *two* queries: the actual query, and a slightly modified version of the query that uses SELECT COUNT instead of SELECT. A method like this will work for simple cases (cases that don't contain GROUP BY statements). It uses a regular expression to turn a SELECT query into a SELECT COUNT query, runs both queries, and returns both the count and the query handle.

```
module DBI
  class DatabaseHandle
    def execute_with_count(query, *args)
      re = /^\s*select .* from/i
      count_query = query.sub(re, 'select count(*) from')
      count = select_one(count_query)
      [count, execute(query)]
    end
  end
end

with_db do |c|
  count, handle = c.execute_with_count(vowel_query)
  puts "I can't believe none of the #{count} vowels " +
    "have been stolen from my collection!"

  puts 'Here they are in the database:'
  handle.each do |r|
    puts "Row #{r['id']}"
  end
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# I can't believe none of the 5 vowels have been stolen from my collection!
# Here they are in the database:
# Row 1
# Row 5
# Row 9
# Row 15
# Row 21
```

See Also

- The Ruby DBI tutorial describes the MySQL `rows` trick but says not to depend on it; we figure as long as you know about the alternatives, you're not dependent on the database-specific shortcut (http://www.kitebird.com/articles/ruby-dbi.html)

Recipe 13.9. Talking Directly to a MySQL Database

Problem

You want to send SQL queries and commands directly to a MySQL database.

Solution

Do you really need to do this? Almost all the time, it's better to use the generic DBI library. The biggest exception is when you're writing a a Rails application, and you need to run a SQL command that you can't express with ActiveRecord.[8]

[8] You could use DBI with ActiveRecord, but most Rails programmers go straight to the database.

If you really want to communicate directly with MySQL, use the Ruby bindings to the MySQL client library (found in the `mysql` gem). It provides an interface that's pretty similar to DBI's.

Here's a MySQL-specific version of the method `with_db`, defined in this chapter's introduction. It returns a `Mysql` object, which you can use to run queries or get server information.

```
require 'rubygems'
require 'mysql'

def with_db
  dbh = Mysql.real_connect('localhost', 'cookbook_user', 'password',
                           'cookbook')
  begin
    yield dbh
  ensure
    dbh.close
  end
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The `Mysql#query` method runs any SQL statement, whether it's a SELECT query or something else. When it runs a query, the return value is a result-set object (a `MysqlRes`); otherwise, it's `nil`. Here it is running some SQL commands:

```
with_db do |db|
  db.query('drop table if exists secrets')
  db.query('create table secrets( id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
                                  secret LONGTEXT )')
  db.query(%{insert into secrets(secret) values
     ("Oh, MySQL, you're the only one who really understands me.")})
end
```

And here's a query:

```
with_db do |db|
  res = db.query('select * from secrets')
  res.each { |row| puts "#{row[0]}: #{row[1]}" }
  res.free
end
# 1: Oh, MySQL, you're the only one who really understands me.
```

Discussion

Like the database connection itself, the result set you get from `query` wants to be closed when you're done with it. This calls for yet another instance of the pattern seen in `with_db`, in which setup and cleanup are delegated to a method that takes a code block. Here's some code that alters `query` to take a code block:

```
class Mysql
  alias :query_no_block :query
  def query(sql)
    res = query_no_block(sql)
    return res unless block_given?
    begin
      yield res
    ensure
      res.free if res
    end
  end
end
```

Now we can write more concise query code, and not have to worry about freeing the result set:

```
with_db do |db|
  db.query('select * from secrets') do |res|
    res.each { |row| puts "#{row[0]}: #{row[1]}" }
  end
end
# 1: Oh, MySQL, you're the only one who really understands me.
```

The method `MysqlRes#each` yields you the rows of a result set as arrays. `MysqlRes#each_hash` also gives you one row at a time, but in hash form: you can access

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

a row's fields by name instead of position. `MysqlRes#num_rows` gives you the number of rows matched by a query.

```
with_db do |db|
  db.query('select * from secrets') do |res|
    puts "#{res.num_rows} row(s) matched:"
    res.each_hash do |hash|
      hash.each { |k,v| puts " #{k} = #{v}" }
    end
  end
end
# 1 row(s) matched:
# id = 1
# secret = Oh, MySQL, you're the only one who really understands me.
```

The MySQL interface provides no protection against SQL injection attacks. If you're sending SQL containing the values of possibly tainted variables, you'll need to quote those values yourself.

See Also

- Recipe 13.15, "Preventing SQL Injection Attacks," for more on SQL injection
- "Using the Ruby MySQL Module" (http://www.kitebird.com/articles/ruby-mysql.html)
- MySQL bindings (http://www.tmtm.org/en/mysql/ruby/)

Recipe 13.10. Talking Directly to a PostgreSQL Database

Problem

You want to send SQL queries and commands directly to a PostgreSQL database.

Solution

As with the MySQL recipe preceding this one, ask: do you really need to do this? The generic DBI library usually works just fine. As before, the main exception is when you need to make low-level SQL calls from within a Rails application.

There are two APIs for communicating with a PostgreSQL database, and both are available as gems. The `postgres` gem provides a Ruby binding to the C client library, and the `postgres-pr` gem provides a pure Ruby interface.

Here's a Postgres-specific version of the method `with_db`, defined in the chapter intro. It returns a `PGconn` object, which you can use to run queries or get server information. This code assumes you're accessing the database through TCP/IP on port 5432 of your local machine.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
require 'rubygems'
require 'postgres'

def with_db
  db = PGconn.connect('localhost', 5432, '', '', 'cookbook',
                      'cookbook_user', 'password')
  begin
    yield db
  ensure
    db.close
  end
end
```

The `PGconn#exec` method runs any SQL statement, whether it's a SELECT query or something else. When it runs a query, the return value is a result-set object (a `PGresult`); otherwise, it's `nil`. Here it is running some SQL commands:

```
with_db do |db|
  begin
    db.exec('drop table secrets')
  rescue PGError
    # Unlike MySQL, Postgres does not have a "drop table unless exists"
    # command. We can simulate it by issuing a "drop table" command and
    # ignoring any error due to the table not existing in the first place.
    # This is essentialy what MySQL's "drop table unless exists" does.
  end

  db.exec('create table secrets( id SERIAL PRIMARY KEY,
                                 secret TEXT )')
  db.exec(%{insert into secrets(secret) values
            ('Oh, Postgres, you\\'re the only one who really understands me.')})
end
```

Here's a query:

```
with_db do |db|
  res = db.query('select * from secrets')
  res.each { |row| puts "#{row[0]}: #{row[1]}" }
end
# 1: Oh, Postgres, you're the only one who really understands me.
```

Discussion

Note the slight differences between the Postgres implementation of SQL and the MySQL implementation. The "drop table if exists" syntax is MySQL-specific. Postgres names the data types differently, and expects string values to be single-quoted.

Like the database connection itself, the result set you get from `exec` wants to be closed when you're done with it. As we did with `query` in the MySQL binding, we can alter `exec` to take an optional code block and do the cleanup for us:

```
class PGconn
  alias :exec_no_block :exec
  def exec(sql)
    res = exec_no_block(sql)
    return res unless block_given?
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
    begin
      yield res
    ensure
      res.clear if res
    end
  end
end
```

Now we can write more concise query code, and not have to worry about freeing the result set:

```
with_db do |db|
  db.exec('select * from secrets') do |res|
    res.each { |row| puts "#{row[0]}: #{row[1]}" }
  end
end
# 1: Oh, Postgres, you're the only one who really understands me.
```

The method `PGresult#each` yields you the rows of a result set as arrays, and `PGresult#num_tuples` gives you the number of rows matched by a query. The Postgres database binding has no equivalent of the MySQL binding's `each_hash`, but you can write one pretty easily:

```
class PGresult
  def each_hash
    f = fields
    each do |array|
      hash = {}
      fields.each_with_index do |field, i|
        hash[field] = array[i]
      end
      yield hash
    end
  end
end
```

Here it is in action:

```
with_db do |db|
  db.exec("select * from secrets") do |res|
    puts "#{res.num_tuples} row(s) matched:"
    res.each_hash do |hash|
      hash.each { |k,v| puts " #{k} = #{v}" }
    end
  end
end
# 1 row(s) matched:
#  id = 1
# secret = Oh, Postgres, you're the only one who really understands me.
```

See Also

- The Postgres reference (http://www.postgresql.org/docs/manuals/)
- The reference for the Ruby Postgres binding (http://ruby.scripting.ca/postgres/)

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

- If you can't get the native Postgres binding installed, try the `postgres-pr` gem; it implements a pure Ruby client to the Postgres server, with more or less the same interface as the native binding
- The `PGconn.quote` method helps you defend against SQL injection attacks; see Recipe 13.15, "Preventing SQL Injection Attacks," for more

Recipe 13.11. Using Object Relational Mapping with ActiveRecord

Problem

You want to store data in a database without having to use SQL to access it.

Solution

Use the ActiveRecord library, available as the `activerecord` gem. It automatically defines Ruby classes that access the contents of database tables.

As an example, let's create two tables in the MySQL database `cookbook` (see the chapter introduction for more on creating the database itself). The `blog_posts` table, defined below in SQL, models a simple weblog containing a number of posts. Each blog post can have a number of comments, so we also define a `comments` table.

```
use cookbook;

DROP TABLE IF EXISTS blog_posts;
CREATE TABLE blog_posts (
  id INT(11) NOT NULL AUTO_INCREMENT,
  title VARCHAR(200),
  content TEXT,
  PRIMARY KEY (id)
) ENGINE=InnoDB;

DROP TABLE IF EXISTS comments;
CREATE TABLE comments (
  id INT(11) NOT NULL AUTO_INCREMENT,
  blog_post_id INT(11),
  author VARCHAR(200),
  content TEXT,
  PRIMARY KEY (id)
) ENGINE=InnoDB;
```

Here are two Ruby classes to represent those tables, and the relationship between them:

```
require 'cookbook_dbconnect'
activerecord_connect                  # See chapter introduction

class BlogPost < ActiveRecord::Base
  has_many :comments
end

class Comment < ActiveRecord::Base
  belongs_to :blog_post
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Now you can create entries in the tables without writing any SQL:

```
post = BlogPost.create(:title => 'First post',
                       :content => "Here are some pictures of our iguana.")

comment = Comment.create(:blog_post => post, :author => 'Alice',
                         :content => "That's one cute iguana!")

post.comments.create(:author => 'Bob', :content => 'Thank you, Alice!')
```

You can also query the tables, relate blog posts to their comments, and relate comments back to their blog posts:

```
blog_post = BlogPost.find(:first)

puts %{#{blog_post.comments.size} comments for "#{blog_post.title}"}
# 2 comments for "First post"

blog_post.comments.each do |comment|
  puts "Comment author: #{comment.author}"
  puts "Comment: #{comment.content}"
end
# Comment author: Alice
# Comment: That's one cute iguana!
# Comment author: Bob
# Comment: Thank you, Alice!

first_comment = Comment.find(:first)
puts %{The first comment was made on "#{first_comment.blog_post.title}"}
# The first comment was made on "First post"
```

Discussion

ActiveRecord uses naming conventions, database introspection, and metaprogramming to hide much of the work involved in defining a Ruby class that corresponds to a database table. All you have to do is define the classes (`BlogPost` and `Comment`, in our example) and the relationships between them (`BlogPost has_many :comments, Comment belongs_to :blog_post`).

Our tables are designed to fit ActiveRecord's conventions about table and field names. The table names are lowercase, pluralized noun phrases, with underscores separating the words. The table names `blog_posts` and `comments` correspond to the Ruby classes `BlogPost` and `Comment`.

Also notice that each table has an autoincremented id field named `id`. This is a convention defined by ActiveRecord. Foreign key references are also named by convention: `blog_post_id` refers to the `id` field of the `blog_posts` table. It's possible to change ActiveRecord's assumptions about naming, but it's simpler to just design your tables to fit the default assumptions.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

For "normal" columns, the ones that don't participate in relationships with other tables, you don't need to do anything special. ActiveRecord examines the database tables themselves to find out which columns are available. This is how we were able to use accessor methods for `blog_posts.title` without explicitly defining them: we defined them in the database, and ActiveRecord picked them up.

Relationships between tables are defined within Ruby code, using decorator methods. Again, naming conventions simplify the work. The call to the `has_many` decorator in the `BlogPost` definition creates a one-to-many relationship between blog posts and comments. You can then call `BlogPost#comments` to get an array full of comments for a particular post. The call to `belongs_to` in the `Comment` definition creates the same relationship in reverse.

There are two more decorator methods that describe relationships between tables. One of them is the `has_one` association, which is rarely used: if there's a one-to-one relationship between the rows in two tables, then you should probably just merge the tables.

The other decorator is `has_and_belongs_to_many`, which lets you join two different tables with an intermediate join table. This lets you create many-to-many relationships, common in (to take one example) permissioning systems.

For an example of `has_and_belongs_to_many`, let's make our blog a collaborative effort. We'll add an `users` table to contain the posts' authors' names, and fix it so that each blog post can have multiple authors. Of course, each author can also contribute to multiple posts, so we've got a many-to-many relationship between users and blog posts.

```
use cookbook;

DROP TABLE IF EXISTS users;
CREATE TABLE users (
  id INT(11) NOT NULL AUTO_INCREMENT,
  name VARCHAR(200),
  PRIMARY KEY (id)
) ENGINE=InnoDB;
```

Because a blog post can have multiple authors, we can't just add an `author_id` field to the `blog_posts` table. That would only give us space for a single author per blog post. Instead, we create a join table that maps authors to blog posts.

```
use cookbook;

DROP TABLE IF EXISTS blog_posts_users;
CREATE TABLE blog_posts_users (
  blog_post_id INT(11),
  user_id INT(11)
) ENGINE=InnoDB;
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Here's another naming convention. ActiveRecord expects you to name a join table with the names of the tables that it joins, concatenated together with underscores. It expects the table names to be in alphabetical order (in this case, the `blog_posts` table comes before the `users` table).

Now we can create a `User` class that mirrors the `users` table, and modify the `BlogPost` class to reflect its new relationship with `users`:

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :blog_posts
end

class BlogPost < ActiveRecord::Base
  has_and_belongs_to_many :authors, :class_name => 'User'
  has_many :comments, :dependent => true
end
```

The `has_and_belongs_to_many` decorator method defines methods that navigate the join table. We specify the `:class_name` argument because otherwise ActiveRecord has no idea which ActiveRecord class corresponds to an "authors" relationship. Without `:class_name`, it would look for a nonexistent `Author` class.

With the relationships in place, it's easy to find blog posts for an author, and authors for a blog post:

```
# Retroactively make Bob and Carol the collaborative authors of our
# first blog post.
User.create(:name => 'Bob', :blog_posts => [post])
User.create(:name => 'Carol', :blog_posts => [post])

author = User.find(:first)
puts "#{author.name} has made #{author.blog_posts.size} blog post(s)."
# Bob has made 1 blog post(s).

puts %{The blog post "#{post.title}" has #{post.authors.size} author(s).}
# The blog post "First post" has 2 author(s).
```

As with the `has_many` or `belongs_to` relationships, the `has_and_belongs_to_many` relationship gives you a `create` method that lets you create new items and their relationships to other items:

```
author.blog_posts.create(:title => 'Second post',
                         :content => 'We have some cats as well.')
```

And since the `blog_posts` method returns an array-like object, you can iterate over it to find all the blog posts to which a given user contributed:

```
author.blog_posts.each do |post|
  puts %{#{author.name}'s blog post "#{post.title}" } +
       "has #{post.comments.size} comments."
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# Bob's blog post "First post" has 2 comments.
# Bob's blog post "Second post" has 0 comments.
```

If you want to delete an item from the database, you can use the `destroy` method available to all ActiveRecord objects:

```
BlogPost.find(:first).destroy
```

However, deleting a blog post does not automatically remove all the comments associated with that blog post. You must tell ActiveRecord that comments cannot exist independently of a blog post, like so:

```
class BlogPost < ActiveRecord::Base
  has_many :comments, :dependent => destroy
end
```

Why doesn't ActiveRecord do this automatically? Because it's not always a good idea. Think about authors: unlike comments, authors *can* exist independently of a blog post. Deleting a blog post shouldn't automatically delete all of its authors. ActiveRecord depends on you to make this kind of judgment, using your knowledge about your application.

See Also

- http://rails.rubyonrails.com/classes/ActiveRecord/Associations/ClassMethods.html
- Recipe 15.7, "Understanding Pluralization Rules," for more on the connection between the table name and the ActiveRecord class name

Recipe 13.12. Using Object Relational Mapping with Og

Credit: Mauro Cicio

Problem

You want to store data in a database, without having to use SQL to create or access the database.

Solution

Use the Og (ObjectGraph) library, available as the `og` gem. Where ActiveRecord has a database-centric approach to object-relational mapping, Og is Ruby-centric. With ActiveRecord, you define the database schema ahead of time and have the library figure

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

out what the Ruby objects should look like. With Og, you define the Ruby objects and let the library take care of creating the database schema.

The only restriction Og imposes on your class definitions is that you must use special versions of the decorator methods for adding attribute accessors. For instance, instead of calling `attribute` to define accessor methods, you call `property`.

Here we define a basic schema for a weblog program, like that defined in Recipe 13.11:

```
require 'cookbook_dbconnect'
require 'og'

class BlogPost
  property :title, :content, String
end

class Comment
  property :author, :content, String
  belongs_to :og_post, BlogPost
end

# Now that Comment's been defined, add a reference to it in BlogPost.
class BlogPost
  has_many :comments, Comment
end
```

After defining the schema, we call the `og_connect` method defined in the chapter introduction. Og automatically creates any necessary database tables:

```
og_connect
# Og uses the Mysql store.
# Created table 'ogcomment'.
# Created table 'ogblogpost'.
```

Now we can create a blog post and some comments:

```
post = BlogPost.new
post.title = "First post"
post.content = "Here are some pictures of our iguana."
post.save!

[["Alice", "That's one cute iguana!"],
 ["Bob", "Thank you, Alice!"]].each do |author, content|
  comment = Comment.new
  comment.blog_post = post
  comment.author = author
  comment.content = content
  comment.save!
end
```

As with ActiveRecord, we can query the tables, relate blog posts to their comments, and relate comments back to their blog posts:

```
post = BlogPost.first
puts %{#{post.comments.size} comments for "#{post.title}"}
# 2 comments for "First post"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
post.comments.each do |comment|
  puts "Comment author: #{comment.author}"
  puts "Comment: #{comment.content}"
end
# Comment author: Alice
# Comment: That's one cute iguana!
# Comment author: Bob
# Comment: Thank you, Alice!

puts %{The first comment was made on "#{Comment.first.blog_post.title}"}
# The first comment was made on "First post"
```

Discussion

Like the ActiveRecord library, Og implements Martin Fowler's Active Record Pattern. While ActiveRecord does this by making all classes derive from the base class `ActiveRecord::Base`, Og does it by using custom attribute accessors instead of the traditional Ruby accessors. In this example, `Comment` and `BlogPost` are POR (Plain Old Ruby) classes, with accessor methods like `author` and `author=`, but those methods were defined with Og decorators instead of the standard Ruby decorators. This table shows the mapping between the two sets of decorators.

Table 13-2.

Standard Ruby accessors	Og accessors
attribute	roperty
attr_accessor	prop_accessor
attr_reader	prop_reader
attr_writer	prop_writer

Each of the Og decorator methods takes a Ruby class as its last argument: `String`, `Integer`, or the like. Og uses this to define the type of the corresponding database row. You can also specify `Object` as a field type, and Og will transparently store YAML representations of arbitrary Ruby objects in the corresponding database field.

ActiveRecord defines all kinds of conventions about how you're supposed to name your database tables and fields. Og doesn't care: it names database tables and fields that correspond to the names you use in your Ruby code.

Just as with ActiveRecord, relationships between Og tables are defined within Ruby code, using decorator methods. The API is almost exactly the same as ActiveRecord's. In the Solution section, we saw how to create a one-to-many relationship between blog posts and comments: by calling `belongs_to` in `Comment` and `has_many` in `BlogPost`. This relationship makes it possible to simply call `BlogPost#comments` and get an array of comments on a post.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Og defines two more decorator methods for describing relationships between tables. One of them is the `has_one` association, which is rarely used: if there's a one-to-one relationship between the rows in two tables, then you should probably just merge the tables.

The other decorator is `many_to_many`, which lets you to join two different tables with an intermediate join table. This lets you create many-to-many relationships, common in (to take one example) permissioning systems.

For an example of `many_to_many`, let's make our blog a collaborative effort. We'll add a `User` class that holds the posts' authors' names, and fix it so that each blog post can have multiple authors. Of course, each author can also contribute to multiple posts, so we've got a many-to-many relationship between users and blog posts. Og needs to know the class definition in order to create the necessary database tables, so the following code snippet should appear before the `og_connect` invocation in your program:

```
class Person
  property :name, String
  many_to_many :posts, BlogPost
end
```

The `many_to_many` decorator tells Og to create a table to store the people, and a join table to map authors to their blog posts. It also defines methods that navigate the join table, as we'll see in a moment.

Of course, the many-to-many relationship goes both ways: `BlogPost` has a many-to-many relationship to `Person`. So add a `many_to_many` call to the definition of `BlogPost` (this, too, must show up before your `og_connect` call):

```
class BlogPost
  many_to_many :authors, Person
end
```

With these relationships in place, it's easy to find blog posts for an author, and authors for a blog post:

```
og_connect

# Retroactively make Bob and Carol the collaborative authors of our
# first blog post.
['Bob', 'Carol'].each do |name|
  p = Person.new
  p.name = name
  p.save
end
Person.find_by_name('Bob').add_post(post)
Person.find_by_name('Carol').add_post(post)

author = Person.first
puts "#{author.name} has made #{author.posts.size} blog post(s)."
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
# Bob has made 1 blog post(s).

puts %{The blog post "#{post.title}" has #{post.authors.size} author(s).}
# The blog post "First post" has 2 author(s).
```

To add an anonymous `BlogPost` on the fly, use the `add_post` method as follows:

```
author.add_post(BlogPost.create_with({
    :title => 'Second post',
    :content => 'We have some cats as well.'
} ))
```

Since `Person posts` returns an array-like object, you can iterate over it to find all the blog posts to which a given user contributed:

```
author.posts.each do |post|
  puts %{#{author.name}'s blog post "#{post.title}" has #{post.comments.size}
comments.}
end

# Bob's blog post "First post" has 2 comments.
# Bob's blog post "Second post" has 0 comments.
```

If you want to delete an object from the database, you can use the `delete` method available to all Og database objects:

```
BlogPost.first.delete
```

Deleting a blog post will automatically remove all the comments associated with that blog post. This automatic deletion (i.e., cascade deletion) is not always a good idea. For instance, we *don't* want the authors of a blog post to be deleted when the post itself is deleted! We can avoid the cascade deletion by passing `false` in as an argument to the `delete` method:

```
BlogPost.first.delete(false)
```

If you want some associated objects (like comments) to get cascade-deleted, and other objects (like authors) to be left alone, the best strategy is to implement the cascade yourself, in post-delete hooks.

See Also

- The Active Record pattern is described in *Patterns of Enterprise Application Architecture* by Martin Fowler (Addison-Wesley)

Recipe 13.13. Building Queries Programmatically

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Problem

You have to write fragments of SQL to pass parameters into an ActiveRecord query. You'd like to dispense with SQL altogether, and represent the query paramaters as a Ruby data structure.

Solution

Here's a simple solution. The method `ActiveRecord::Base.find_by_map` defined below picks up where `find` leaves off. Normally a query is represented by a SQL fragment, passed in as the `:conditions` argument. Here, the `:conditions` argument contains a mapping of database field names to the desired values:

```
require 'cookbook_dbconnect'

class ActiveRecord::Base
  def self.find_by_map(id, args={}.freeze)
    sql = []
    values = []
    args[:conditions].each do |field, value|
      sql << "#{field} = ?"
      values << value
    end if args[:conditions]
    args[:conditions] = [sql.join(' AND '), values]
    find(id, args)
  end
end
```

Here's `find_by_map` in action, using the `BlogPost` class first seen in Recipe 13.11:

```
activerecord_connect

class BlogPost < ActiveRecord::Base
end

BlogPost.create(:title => 'Game Review: Foosball Carnage',
                :content => 'Four stars!')
BlogPost.create(:title => 'Movie Review: Foosball Carnage: The Movie',
                :content => 'Zero stars!')

BlogPost.find_by_map(:first,
                     :conditions => {:title =>
                                     'Game Review: Foosball Carnage' }
                    ).content
# => "Four stars!"
```

Discussion

ActiveRecord saves you from having to write a lot of SQL, but you still have to write out the equivalent of a SQL WHERE clause every time you call `ActiveRecord::Base#find`. The `find_by_map` method lets you define those queries as Ruby hashes.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

But `find_by_map` only lets you run one type of query: the kind where you're restricting fields of the database to specific values. What if you want to do a query that matches a field with the LIKE construct, or combine multiple clauses into a single query with AND or OR?

A hash can only represent a very simple SQL query, but the `Criteria` object, below, can represent almost any WHERE clause. The implementation is more complex but the idea is the same. We define a data structure that can represent the WHERE clause of a SQL query, and a way of converting the data structure into a real WHERE clause.

Here's the basic class. A `Criteria` acts like a hash, except it maps a field name to a value *and* a SQL operator. Instead of mapping `:title` to `'Game Review: Foosball Carnage'`, you can map it to `['%Foosball%', 'LIKE']`. Each `Criteria` object can be chained to other objects as part of an AND or OR clause.

```
class Criteria < Hash
  def initialize(values)
    values.each { |k,v| add(k, *v) }
    @or_criteria = nil
    @and_criteria = nil
  end

  :private
  attr_accessor :or_criteria, :and_criteria

  :public
  def add(field, value, operation='=')
    self[field] = [value, operation]
  end

  def or(criteria)
    c = self
    while c.or_criteria != nil
      break if c == criteria
      c = c.or_criteria
    end

    c.or_criteria = criteria
    return self
  end

  def and(criteria)
    c = self
    while c.and_criteria != nil
      break if c == criteria
      c = c.and_criteria
    end

    c.and_criteria = criteria
    return self
  end
```

This method turns a `Criteria` object, and any other objects to which it's chained, into a SQL string with substitutions, and an array of values to use in the substitutions:

```
class Criteria
  def to_where_clause
    sql = []
    values = []
    each do |field, value|
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
      if value.respond_to? :to_str
        value, operation = value, '='
      else
        value, operation = value[0..1]
      end
      sql << "#{field} #{operation} ?"
      values << value
    end
    sql = '(' + sql.join(' AND ') + ')'

    if or_criteria
      or_where = or_criteria.to_where_clause
      sql = "(#{sql} OR #{or_where.shift})"
      values += or_where
    end

    if and_criteria
      and_where = and_criteria.to_where_clause
      sql = "(#{sql} AND #{and_where.shift})"
      values += and_where
    end
    return values.unshift(sql)
  end
end
```

Now it's simple to write a version of `find` that accepts a `Criteria`:

```
class ActiveRecord::Base
  def self.find_by_criteria(id, criteria, args={}.freeze)
    args = args.dup
    args[:conditions] = criteria.to_where_clause
    find(id, args)
  end
end
```

Here's `Criteria` used to express a complex SQL WHERE clause with a little bit of Ruby code. This query searches the `blog_post` table for reviews of bad movies and good games. The movies and the games must not be about the game of cricket.

```
review = Criteria.new(:title => ['%Review%', 'LIKE'])
bad_movie = Criteria.new(:title => ["%Movie%", 'LIKE'],
                         :content => 'Zero stars!')
good_game = Criteria.new(:title => ['%Game%', 'LIKE'],
                          :content => 'Four stars!')
no_cricket = Criteria.new(:title => ['%Cricket%', 'NOT LIKE'])

review.and(bad_movie.or(good_game)).and(no_cricket)
review.to_where_clause
# => ["((title LIKE ?) AND
#       (((content = ? AND title LIKE ?) OR (content = ? AND title LIKE ?))
#      AND (title NOT LIKE ?)))",
#     "%Review%", "Zero stars!", "%Movie%", "Four stars!", "%Game%",
#     "%Cricket%"]

BlogPost.find_by_criteria(:all, review).each { |post| puts post.title }
# Game Review: Foosball Carnage
# Movie Review: Foosball Carnage: The Movie
```

The technique is a general one. It's easier for a human to construct Ruby data structures than to write valid SQL clauses, so write code to convert the one into the other. You can use this technique wherever any library expects you to write SQL.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

For instance, the `find` method expects SQL fragments representing a query's ORDER BY or GROUP BY clause. You could represent each as an array of fields, and generate the SQL as needed.

```
# Just an idea…
order_by = [[:title, 'ASC']]
```

See Also

- The `Criteria` class is inspired by the one in the Torque ORM library for Java (http://db.apache.org/torque/)

Recipe 13.14. Validating Data with ActiveRecord

Problem

You want to prevent bad data from getting into your ActiveRecord data objects, whether the source of the data is clueless users or buggy code.

Solution

The simplest way is to use the methods defined by the `ActiveRecord::Validations` module. Each of these methods (`validates_length_of`, `validates_presence_of`, and so on) performs one kind of validation. You can use them to declare restrictions on the data in your object's fields.

Let's add some validation code to the `Comment` class for the weblog application first seen in Recipe 13.11. Recall that a `Comment` object has two main fields: the name of the author, and the text of the comment. We'll reject any comment that leaves either field blank. We'll also reject comments that are too long, and comments whose body contains any string from a customizable list of profane words.

```
require 'cookbook_dbconnect'
activerecord_connect

class Comment < ActiveRecord::Base
  @@profanity = %w{trot krip}
  @@no_profanity_re = Regexp.new('^(?!.*(' + @@profanity.join('|') + '))')

  validates_presence_of %w{author}
  validates_length_of :content, :in => 1..200
  validates_format_of :content, :with => @@no_profanity_re,
                      :message => 'contains profanity'
end
```

`Comment` objects that don't fit these criteria won't be saved to the database.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
comment = Comment.create
comment.errors.on 'author'                        # => "can't be blank"
comment.errors['content']
# => "is too short (minimum is 1 characters)"
comment.save                                      # => false

comment = Comment.create(:content => 'x' * 1000)
comment.errors['content']
# => "is too long (maximum is 200 characters)"

comment = Comment.create(:author => 'Alice',
  :content => "About what I'd expect from a trotting krip such as yourself!")
comment.errors.count                              # => 1
comment.errors.each_full { |msg| puts msg }
# Content contains profanity

comment = Comment.create(:author => 'Alice', :content => 'I disagree!')
comment.save                                      # => true
```

Discussion

Every ActiveRecord record has an associated `ActiveRecord::Errors` object, which starts out empty. Before the record is saved to the database, all the predefined restrictions for that class of object are checked. Every problem encountered while applying the restrictions adds an entry to the `Errors` object.

If, at the end of this trial by ordeal, the `Errors` object is still empty, ActiveRecord presumes the data is valid, and saves the object to the database.

ActiveRecord's `Validations` module provides many methods that implement validation rules. Apart from the examples given above, the `validates_numericality_of` method requires an integer value (or a floating-point value if you specify `:integer => false`). The `requires_inclusion_of` method will reject any value not found in a predefined list of acceptable values.

If the predefined validation rules aren't enough for you, you can also write a custom validation rule using `validate_each`. For instance, you might validate URL fields by fetching the URLs and making sure they're valid.

The method `Errors#each_full` prepends each error message with the corresponding field name. This is why the actual error messages look like "is empty" and "contains profanity": so `each_full` will yield "Author is empty" and "Content contains profanity".

ActiveRecord assumes you named your fields so that these messages will be readable. You can customize the messages by passing in keyword arguments like `:message`, but then you'll need to access the messages with `Errors#each` instead of `Errors#each_full`. Here's an alternate implementation of the `Comment` validation rules that customizes the messages:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
require 'cookbook_dbconnect'
activerecord_connect

class Comment < ActiveRecord::Base
  @@profanity = %w{trot krip}
  @@no_profanity_re = Regexp.new('^(?!.*(' + @@profanity.join('|') + '))')

  validates_presence_of %w{author}, :message => 'Please enter your name.'
  validates_length_of :content, :in => 1..200,
              :too_short => 'Please enter a comment.',
                    :too_long => 'Comments are limited to 200 characters.'
  validates_format_of :content, :with => @@no_profanity_re,
                      :message => 'Try to express yourself without profanity.'
end
```

The declarative validation style should be flexible enough for you, but you can do custom validation by defining a `validate` method. Your implementation is responsible for checking the current state of an object, and populating the `Errors` object with any appropriate error messages.

Sometimes new objects have different validation rules from existing objects. You can selectively apply a validation rule by passing it the `:on` option. Pass in `:on => :create`, and the validation rule will only be triggered the first time an object is saved to the database. Pass in `:on => :update`, and the validation rule will be triggered every time *except* the first. You can also define the custom validation methods `validate_on_add` and `validate_on_update` as well as just plain `validate`.

See Also

- Recipe 1.19, "Validating an Email Address"
- Recipe 8.6, "Validating and Modifying Attribute Values"
- The built-in validation methods (http://rubyonrails.org/api/classes/ActiveRecord/Validations/ClassMethods.html)
- Some sample `validate` implementations (http://rubyonrails.org/api/classes/ActiveRecord/Validations.html)
- The `Errors` class defines a few helper methods for doing validation in a `validate` implementation (http://rubyonrails.org/api/classes/ActiveRecord/Errors.html)
- Og defines some declarative validation methods, similar to ActiveRecord's (http://www.nitrohq.com/view/Validation/Og)

Recipe 13.15. Preventing SQL Injection Attacks

Problem

You want to harden your code against SQL injection attacks, whether in DBI or ActiveRecord code.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Solution

With both ActiveRecord and DBI applications, you should create your SQL with question marks where variable interpolations should go. Pass in the variables along with the SQL to `DatabaseHandle#execute`, and the database will make sure the values are properly quoted.

Let's work against a simple database table tracking people's names:

```
use cookbook;

DROP TABLE IF EXISTS names;
CREATE TABLE names (
  first VARCHAR(200),
  last VARCHAR(200)
) ENGINE=InnoDB;

INSERT INTO names values ('Leonard', 'Richardson'),
                         ('Lucas', 'Carlson'),
                         ('Michael', 'Loukides');
```

Here's a simple script that searches against that table. It's been hardened against SQL injection attacks with three techniques:

```
#!/usr/bin/ruby
# no_sql_injection.rb

require 'cookbook_dbconnect'
activerecord_connect
class Name < ActiveRecord::Base; end

print 'Enter a last name to search for: '
search_for = readline.chomp

# Technique 1: use ActiveRecord question marks
conditions = ["last = ?", search_for]

Name.find(:all, :conditions => conditions).each do |r|
  puts %{Matched "#{r.first} #{r.last} with ActiveRecord question marks"}
end

# Technique 2: use ActiveRecord named variables
conditions = ["last = :last", {:last => search_for}]

Name.find(:all, :conditions => conditions).each do |r|
  puts %{Matched "#{r.first} #{r.last}" with ActiveRecord named variables}
end

# Technique 3: use DBI question marks
with_db do |db|
  sql = 'SELECT first, last FROM names WHERE last = ?'

  db.execute(sql, [search_for]).fetch_hash do |r|
    puts %{Matched "#{r['first']} #{r['last']}" with DBI question marks}
  end
end

puts "Done"
```

Here's how this script looks in use:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
$ ruby no_sql_injection.rb
Enter a last name to search for: Richardson
Matched "Leonard Richardson" with ActiveRecord question marks
Matched "Leonard Richardson" with ActiveRecord named variables
Matched "Leonard Richardson" with DBI question marks
Done

# See the Discussion if you're not sure how this attack is supposed to work.
$ ruby no_sql_injection.rb
Enter a last name to search for: " or 1=1
Done
```

Discussion

SQL is a programming lanuage, and running SQL is like calling `eval` on a string of Ruby code. Unless you have complete control over the entire SQL string and all the variables interpolated into it, you need to be very careful. Just one mistake can leave you open to information leakage or database corruption.

Here's a naive version of `sql_injection.rb` that's vulnerable to an injection attack. If you habitually write code like this, you may be in trouble:

```
#!/usr/bin/ruby
# sql_injection.rb
require 'cookbook_dbconnect'

print "Enter a last name to search for: "
search_for = readline.chomp
query = %{select first, last from names where last="#{search_for}"}
puts query if $DEBUG
with_db do |db|
  db.execute(query).fetch_hash do |r|
    puts %{Matched "#{r['first']} #{r['last']}"}
  end
end
```

Looks fine, right?

```
$ ruby -d sql_injection.rb
Enter a last name to search for: Richardson
select first_name, last_name from people where last_name="Richardson"
Matched "Leonard Richardson"
```

Not necessarily. Whatever I type is simply being stuck into a SQL statement. What if I typed as my "query" part of a SQL `WHERE` clause? One that, when combined with the original `WHERE` clause, matched anything and everything?

```
$ ruby -d sql_injection.rb
Enter a last name to search for: " or 1=1
select first_name, last_name from people where last_name="" or 1=1
Matched "Leonard Richardson"
Matched "Lucas Carlson"
Matched "Michael Loukides"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

I can see every name in the table.

This is just one example. SQL injection attacks can also alter or delete data from a database.

The correct version of this program, the one described in the Solution, quotes my attempt at a SQL injection attack. My attack is executed as a normal query: the program looks for people (or robots, I guess) whose last name is the string " or *1=1*. Quoting the data makes the application do what you want it to do every time, no matter what kind of weird data a user can come up with.

DBI will not run two SQL commands in a single `do` or `execute` call, so certain types of SQL injection attacks are impossible with DBI. You can hijack a `SELECT` statement to make it select something else, but unlike with some other systems, you can't make a `SELECT` also do an `UPDATE` or `DELETE`. An attacker can't use SQL injection to drop database tables unless your application already runs a `DROP TABLE` command somewhere.

You don't usually write full-blown SQL statements with ActiveRecord, but you do write conditions: snippets of SQL that get turned into to the `WHERE` clauses of `SELECT` or `UPDATE` statements. Whenever you write SQL, you must take these precautions.

See Also

- "Securing your Rails application" in the Ruby on Rails manual (http://manuals.rubyonrails.com/read/chapter/43)
- The RDoc for the `ActiveRecord::Base` class
- "SQL Injection Attacks by Example" is a readable introduction to this topic (http://www.unixwiz.net/techtips/sql-injection.html)
- "Using the Ruby DBI Module" has a section on quoting (http://www.kitebird.com/articles/ruby-dbi.html#TOC_8)

Recipe 13.16. Using Transactions in ActiveRecord

Problem

You want to perform database operations as a group: if one of the operations fails, it should be as though none of them had ever happened.

Solution

Include `active_record/transactions`, and you'll give each ActiveRecord class a `transaction` method. This method starts a database transaction, runs a code block, then

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

commits the transaction. If the code block throws an exception, the database transaction is rolled back.

Here's some simple initialization code to give ActiveRecord access to the database tables for the weblog system first seen in Recipe 13.11:

```
require 'cookbook_dbconnect'
activerecord_connect # See chapter introduction

class User < ActiveRecord::Base
  has_and_belongs_to_many :blog_posts
end

class BlogPost < ActiveRecord::Base
  has_and_belongs_to_many :authors, :class_name => 'User'
end
```

The `create_from_new_author` method below creates a new entry in the `users` table, then associates it with a new entry in the `blog_posts` table. But there's a 50% chance that an exception will be thrown right after the new author is created. If that happens, the author creation is rolled back: in effect, it never happened.

```
require 'active_record/transactions'

class BlogPost
  def BlogPost.create_from_new_author(author_name, title, content)
    transaction do
      author = User.create(:name => author_name)
      raise 'Random failure!' if rand(2) == 0
      create(:authors => [author], :title => title, :content => content)
    end
  end
end
```

Since the whole operation is enclosed within a `transaction block`, an exception won't leave the database in a state where the author has been created but the blog entry hasn't:

```
BlogPost.create_from_new_author('Carol', 'The End Is Near',
                                'A few more facts of doom…')
# => #<BlogPost:0xb78b7c7c … >

# The method succeeded; Carol's in the database:
User.find(:first, :conditions=>"name='Carol'")
# => #<User:0xb7888ae4 @attributes={"name"=>"Carol", … }>

# Let's do another one…
BlogPost.create_from_new_author('David', 'The End: A Rebuttal',
                                'The end is actually quite far away…')
# RuntimeError: Random failure!

# The method failed; David's not in the database:
User.find(:first, :conditions=>"name='David'")
# => nil
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

You should use database transactions whenever one database operation puts the database into an inconsistent state, and a second operation brings the database back into consistency. All kinds of things can go wrong between the first and second operation. The database server might crash or your application might throw an exception. The Ruby interpreter might decide to stop running your thread for an arbitrarily long time, giving other threads a chance to marvel at the inconsistent state of the database. An inconsistent database can cause problems that are very difficult to debug and fix.

ActiveRecord's transactions piggyback on top of database transactions, so they'll only work if your database supports transactions. Most databases do these days; chances are you won't have trouble unless you're using a MySQL database and not using InnoDB tables. However, most of the open source databases don't support *nested*transactions, so you're limited to one transaction at a time with a given database connection.

In addition to a code block, the `transaction` method can take a number of ActiveRecord objects. These are the objects that participate in the transaction. If the transaction fails, then not only will the database be restored to its previous state, so will the member variables of the objects.

This is useful if you're defining a method that modifies ActiveRecord objects themselves, not just the database representations of those objects. For instance, a shopping cart object might keep a running total that's consulted by the application, but not stored in the database.

See Also

- http://wiki.rubyonrails.com/rails/pages/HowToUseTransactions
- http://rubyonrails.org/api/classes/ActiveRecord/Transactions/ClassMethods.html

Recipe 13.17. Adding Hooks to Table Events

Problem

You want to run some code whenever a database row is added, updated, or deleted. For instance, you might want to send out email whenever a new blog post is created.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Solution

For Og, use the aspect-oriented features of `Glue::Aspect`. You can use its `before` and `after` methods to register code blocks that run before or after any Og method. The methods you're most likely to wrap are `og_insert`, `og_update`, and `og_delete`.

In the following code, I take the `BlogPost` class first defined in Recipe 13.12, and give its `og_insert` method an aspect that sends out email:

```
require 'cookbook_dbconnect'
require 'og'
require 'glue/aspects'

class BlogPost
  property :title, :content, String
  after :on => :og_insert do |post|
    puts %{Sending email notification of new post "#{post.title}"}
    # Actually send the email here…
    end
end

og_connect
post = BlogPost.new
post.title = 'Robots are taking over'
post.content = 'Think about it! When was the last time you saw another human?'
post.save!
# Sending email notification of new post "Robots are taking over"
```

This technique works with ActiveRecord as well (since aspect-oriented programming is a generic technique), but ActiveRecord defines two different approaches: callbacks and the `ActiveRecord::Observer` class.

Any `ActiveRecord::Base` subclass can define a number of callback methods: `before_find`, `after_save`, and so on. These methods run before or after the corresponding ActiveRecord methods. Here's an callback-based ActiveRecord implementation of the Og example, running against the `blog_post`table first defined in Recipe 13.11. *If you ran the previous example in a session, quit it now and start a new session.*

```
require 'cookbook_dbconnect'
activerecord_connect

class BlogPost < ActiveRecord::Base
  def after_create
    puts %{Sending email notification of new blog post "#{title}"}
    # Actually send the email here…
  end
end

post = BlogPost.create(:title => 'Robots: Gentle Yet Misunderstood',
                       :content => 'Popular misconceptions about robERROR 40')
# Sending email notification of new blog post "Robots: Gentle Yet Misunderstood
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

ActiveRecord's callback interface is simple, but it's got a big disadvantage compared to Og's. You can attach multiple aspects to a single method, but you can only define a callback method once.

This makes little difference when you only want the callback method to do one thing. But suppose that in addition to sending email whenever a blog post is created, you also want to notify people of new posts through an instant messenger client, and to regenerate static syndication feeds to reflect the new post.

If you used a callback, you'd have to lump all of that code together in `after_create`. With aspects, each piece of functionality can go into a separate aspect. It's easy to add more, or to disable a single one without affecting the others. Aspects keep auxilliary code from cluttering up your core data classes.

Fortunately, ActiveRecord provides a strategy other than the callback methods. You can define a subclass of `ActiveRecord::Observer`, which implements any of the callback methods, and use the `observe` decorator to attach it to the classes you want to watch. Multiple `Observers` can watch a single class, so you can split up the work.

Here's a third example of the email notification code. *Again, start a new session if you're following this recipe in* `irb`.

```
require 'cookbook_dbconnect'
activerecord_connect

class BlogPost < ActiveRecord::Base
end

class MailObserver < ActiveRecord::Observer
  observe BlogPost
  def after_create(post)
    puts %{Sending email notification of new blog post "#{post.title}"}
    # Actually send the email here.
  end
end
ActiveRecord::Base.observers = MailObserver

post = BlogPost.new(:title => "ERROR 40",
                    :content => "ERROR ERROR ERROR ERROR ERROR")
post.save
# Sending email notification of new blog post "ERROR 40"
```

Note the call to `ActiveRecord::Base.observers=`. Calling this method starts the observer running. You can call `ActiveRecord::Base.observers=` whenever you need to add one or more `Observers`. Despite the implication of the method name, calling it twice won't overwrite one set of observers with another.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

In a Rails application, observers are traditionally started by putting code like the following in the `environment.rb` file:

```
# environment.rb
config.active_record.observers = MailObserver
```

When working with ActiveRecord, if you want to attach an `Observer` to a specific ActiveRecord class, you can name it after that class: for instance, `BlogPostObserver` will automatically observe the `BlogPost` class. Obviously, this only works for a single `Observer`.

See Also

- Recipe 10.15
- ActiveRecord callbacks documentation (http://rubyonrails.org/api/classes/ActiveRecord/Callbacks.html)
- ActiveRecord `Observer` documentation (http://rails.rubyonrails.com/classes/ActiveRecord/Observer.html)
- Og used to define a class called `Og::Observer` that worked like ActiveRecord's `ActiveRecord::Observer`, but it's been deprecated in favor of aspects; some of the documentation for `Og::Observer` is still online, so be careful not to get confused

Recipe 13.18. Adding Taggability with a Database Mixin

Problem

Without writing a lot of code, you want to make one of your database tables "taggable"—make it possible to add short strings describing a particular item in the table.

Solution

Og comes complete with a tagging mixin. Just call is `Taggable` on every class you want to be taggable. Og will create all the necessary tables.

Here's the `BlogPost` class from Recipe 13.12, only this time it's `Taggable`. Og automatically creates a `Tag` class and the necessary database tables:

```
require 'cookbook_dbconnect'
require 'og'
require 'glue/taggable'

class BlogPost
  is Taggable
  property :title, :content, String
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
og_connect

# Now we can play around with tags.
post = BlogPost.new
post.title = 'Some more facts about video games'
post.tag(['editorial', 'games'])

BlogPost.find_with_tags('games').each { |puts| puts post.title }
# Some more facts about video games

Tag.find_by_name('editorial').blog_posts.each { |post| puts post.title }
# Some more facts about video games
```

To get this feature in ActiveRecord, you'll need to install the `acts_as_taggable` gem, and you must create the database tables yourself. Here are the tables necessary to add tags to the ActiveRecord `BlogPost` class (first described in Recipe 13.11): a generic `tags` table and a join table connecting it to `blog_posts`.

```
DROP TABLE IF EXISTS tags;
CREATE TABLE tags (
  id INT(11) NOT NULL AUTO_INCREMENT,
  name VARCHAR(32),
  PRIMARY KEY (id)
) ENGINE=InnoDB;

DROP TABLE IF EXISTS tags_blog_posts;
CREATE TABLE tags_blog_posts (
   tag_id INT(11),
   blog_post_id INT(11)
) ENGINE=InnoDB;
```

Note that the join table violates the normal ActiveRecord naming rule. It's called `tags_blog_posts`, even though alphabetical ordering of its component tables would make it `blog_posts_tags`. ActiveRecord does this so all of your application's `tags_join`tables will show up together in a sorted list. If you want to call the table `blog_posts_tags` instead, you'll need to pass the name as the `:join_table` parameter when you call the `acts_as_taggable` decorator below.

Here's the ActiveRecord code that makes `BlogPost` taggable. *If you ran the previous example, run this one in a new*`irb`*session so that you can define a new*`BlogPost`*class.*

```
require 'cookbook_dbconnect'
require 'taggable'
activerecord_connect

class Tag < ActiveRecord::Base
end

class BlogPost < ActiveRecord::Base
  acts_as_taggable
end

# Now we can play around with tags.
post = BlogPost.create(:title => 'Some more facts about inflation.')
post.tag(['editorial', 'economics'])

BlogPost.find_tagged_with(:any=>'editorial').each { |post| puts post.title }
# Some more facts about inflation.
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

A mixin class like `Enumerable` is an easy way to add a lot of functionality to an existing class without writing much code. Database mixins work the same way: you can add new objects and relationships to your data model without having to write a lot of database code. Of course, you'll still need to decide how to incorporate tags into your user interface.

The Og and ActiveRecord tagging mixins work the same way, although the Og mixin hides the details. In addition to your original database table (the one you want to tag), you need a table that contains tags, and a join table connecting the tags to the tagged. Whether you use Og or ActiveRecord, the database schema looks something like Figure 13-1.

Figure 13-1. BlogPosts are associated with Tags through a join table

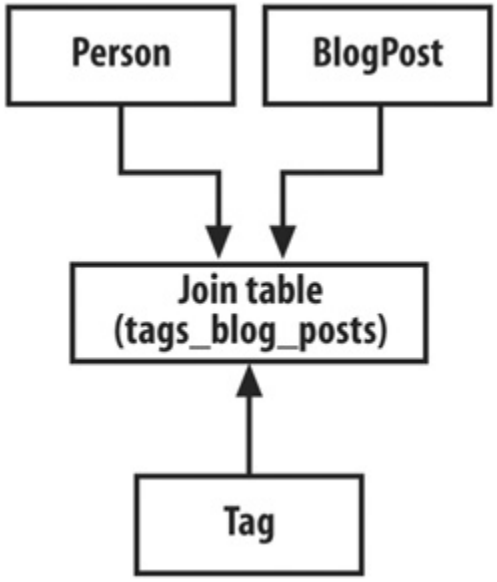
The tagging mixin saves you from having to write code for managing the tag table, and the original table's relationship with it.

But there are two ways to tag something, and we've only covered one. You add tags to `BlogPost` if you want one set of tags for each blog post, probably set by the author of the post. The tags act as canonical categories. What if you want to create a tag system where everyone has their own set of tags for blog posts? Instead of a single system imposed by the authors, every user gets to define a categorization system that makes sense to them.

When you do this, the application doesn't tag a blog post itself. It tags *one person's relationship* to a blog post. The schema looks something like Figure 13-2.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Figure 13-2. When tags are per-user, the join table associates BlogPosts, Tags, and People



Let's implement per-user tagging in ActiveRecord. Instead of making the `tags_blog_posts` table connect a blog post directly to a `tag`, we'll have it connect a tag, a blog post, *and* a person.

```
DROP TABLE IF EXISTS tags_blog_posts;
CREATE TABLE tags_blog_posts (
   tag_id INT(11),
   blog_post_id INT(11),
   created_by_id INT(11)
) ENGINE=InnoDB;
```

Here's the Ruby code. First, some setup we've seen before:

```
require 'cookbook_dbconnect'
require 'taggable'
activerecord_connect

class Tag < ActiveRecord::Base
end

class Person < ActiveRecord::Base
end
```

When each blog post had one set of tags, we called `acts_as_taggable` with no arguments, and the `BlogPost` class was associated directly with the Tag class. This time, we tell `acts_as_taggable` that `BlogPost` objects are associated with `Tag` through the `TagBlogPost` class:

```
# ActiveRecord will automatically define the TagBlogPost class when
# we reference it.
class BlogPost < ActiveRecord::Base
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
  acts_as_taggable :join_class_name => 'TagBlogPost'
end
```

Now we tell `TagBlogPost` that it's associated with the `Person` class: every `TagBlogPost` represents one person's opinions about a single blog post:

```
# Specify that a TagBlogPost is associated with a specific user.
class TagBlogPost
  belongs_to :created_by, :class_name => 'Person',
                          :foreign_key => 'created_by_id'
end
```

Now each `Person` can have their own set of tags on each `BlogPost`:

```
post = BlogPost.create(:title => 'My visit to the steel mill.')
alice = Person.create(:name=>"Alice")
post.tag(['travelogue', 'metal', 'interesting'],
         :attributes => { :created_by => alice })

alices_interests = BlogPost.find_tagged_with(:all => 'interesting',
       :condition => "tags_people.created_by_id = #{alice.id}")
alices_interests.each { |article| puts article.title }
# My visit to the steel mill.
```

Og and ActiveRecord each come with several common mixins. For instance, you can use a mixin to model parent-child relationships between tables (Og is `Hierarchical`, ActiveRecord `acts_as_tree` and `acts_as_nested_set`), or to treat the rows of a table as an ordered lists (Og is `Orderable`, ActiveRecord `acts_as_list`). These can save you a lot of time.

See Also

- The built-in ActiveRecord mixins are all in the `ActiveRecord::Acts` module; see the generated documentation at http://rubyonrails.org/api/
- The `taggable` reference for ActiveRecord (http://taggable.rubyforge.org/)

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01