This course is based on RubyLearning's "Introduction to Sinatra" book. Do consider buying this eBook to support the time and energy put into the various courses by RubyLearning's mentors and to help RubyLearning bring to you relevant eBooks and courses related to the Ruby programming language.

A Simple CRUD app with Sinatra

We shall create a simple CRUD application using Sinatra, ActiveRecord and SQLite3. Remember that Sinatra is ORM-agnostic.

What is CRUD?

Explaining CRUD is beyond the scope of this tutorial. However, you can refresh your knowledge of CRUD - http://en.wikipedia.org/wiki/Create,_read,_update_and_delete.

YAML and Object Serialization

YAML

Refer: http://www.yaml.org/

(Ain't Markup Language) is a human friendly data serialization standard for all program ming languages. YAML is part of standard Ruby. YAML is human-readable and human-writable. You can write YAML files in a text editor and load them into Ruby as objects.

Object Serialization

Refer: http://rubylearning.com/satishtalim/object_serialization.html

Let's take the same example, namely:

```
# p051gamecharacters.rb
class GameCharacter
  def initialize(power, type, weapons)
    @power = power
    @type = type
    @weapons = weapons
  end
  attr_reader :power, :type, :weapons
end
```

The following program creates an object of the above class and then uses **YAML.dump** to save a serialized version (file **gc**) of it to the disk.

Next we use **YAML.load** to read it in.

Here's the code:

```
require 'yaml'
require 'p051gamecharacters'
gc = GameCharacter.new(120, 'Magician', ['spells', 'invisibility'])
open("gc", "w") { |f| YAML.dump(gc, f) }
data = open("gc") { |f| YAML.load(f) }
puts data.power.to_s + ' ' + data.type + ' '
data.weapons.each do |w|
  puts w + ' '
end
```

The contents of the file **gc** are:

```
--
- !ruby/object:GameCharacter
power: 120
type: Magician
weapons:
- spells
- invisibility
```

The documentation of YAML is http://www.yaml.org/YAML_for_ruby.html.

ORM and ActiveRecord

ORM (http://en.wikipedia.org/wiki/Object-relational_mapping), stands for object-relational mapping. ORM libraries map database tables to classes. If a database has a *table called orders*, our program will have a *class named Order*. Rows in this table correspond to objects of the class - a particular order is represented as an object of class Order. Within that object, attributes are used to get and set the individual columns. Our Order object has methods to get and set the amount, the sales tax, and so on.

Therefore an ORM layer maps tables to classes, rows to objects, and columns to attributes of those objects. Class methods are used to perform table-level operations, and instance methods perform operations on the individual rows. **ActiveRecord is an ORM**. ActiveRecord relieves us of the hassles of dealing with the underlying database, leaving us free to work on business logic. ActiveRecord relies on convention over configuration. Wherever possible, ActiveRecord guesses the correct configuration by reflecting against the data schema. When you do need a specific override, you specify the override directly in your class.

Also, ActiveRecord assumes that:

- database table names, like variable names, have lowercase letters and underscores between the words.
- table names are always plural.
- files are named in lowercase with underscores.

Ruby programs can interact with databases like SQLite via adapters in ActiveRecord. An adapter can be a pure Ruby implementation or a hybrid Ruby/C extension. From your Ruby code, you need to specify only the name of the adapter you want to use; ActiveRecord will provide the Ruby bridge code and worry about loading the native extension (if necessary).

ActiveRecord assumes that every table it handles uses a primary key of type integer called **id** internally. **ActiveRecord** uses the value in this column to keep track of the data it has loaded from the database and to link between data in different tables.