Web Development with Ruby on Rails

Jeff Cohen and Raghu Betina

Spring 2013

This is an incomplete draft. You may find typos and plenty of omitted material.

These notes summarize of the key concepts covered in this *Web Development* course. See also the *Rails Guides*, the *Rails API*, and the offical online Ruby documentation.

Proficiency in the Ruby language is necessary for mastery of Rails fundamentals. However, simple “prototype” Rails apps can be developed with only basic Ruby skills: simple data structure containers such as arrays and hashes; loops; conditional logic; and method calls.

Rails

Rails is a framework of prewritten code in the Ruby programming language that enables rapid creation of database-backed web applications. Rails adopts a traditional Model-View-Controller architecture, and accelerates development by prescribing a set of conventions that reduce the amount of code that is typically necessary when leveraging a web framework.

The most important practices of modern agile development practices are embraced, namely: unit testing; the DRY, YAGNI, and SRP principles; intention-revealing code; source code control; and iterative development.

Ruby

Ruby is a fully object-oriented, dynamically-typed language. Classes may derive from only one base class, but can “mix in” an unlimited number of *modules,* which are simple method packages.

There are no true primitive types; even numeric literals are objects of a specific numeric class. Classes are considered “open” and can be redefined by user code.

All method invocation is dynamically dispatched, and all method arguments are passed by reference. Ruby objects are garbage-collected, and explicit lifetime management is not supported.

The built-in classes that your should become familar with are *Fixnum, String, Array, Hash,* and *Symbol*.

The *Fixnum* class holds integer values. The upper and lower bound depends on the machine word length; usually this is a 32-bit or 64-bit word. (Integer values larger than that must use the *Bignum* class, which we will rarely use in web development.)

Because all things in Ruby are objects, and all objects can respond to a method named *.class*, the following code is an example of how to verify that an object is of type *Fixnum:*

```
puts 5.class     # displays Fixnum
```

Arrays in Ruby can be quickly constructed using the [ ] object-literal syntax, whereas *Hashes* can be created using the { } syntax. Arrays and hashes are heterogeneous, and can contain elements of mixed types.

Array values are retrieved with zero-based indices:

```
favorites = ["Purple", "Cookies", 7]
```

```
puts favorites[1]   # displays "Cookies"
```

Arrays have lots of cool methods on them. Google for "Ruby Array" and learn the various methods.

```
puts favorites.count   # displays 3
favorites.push "Chicago"
puts favorites.count   # displays 4
```

Hash values are retrieved with their key. Keys, like values, can be of any type:

```
favorites = { "color" => "Purple", :number => 7 }
puts favorites["color"]    # displays "Purple"
```

You can modify values quickly like this:

```
favorites["color"] = "Red"
puts favorites["color"]    # displays "Red"
```

And even add brand-new key-value pairs as well:

```
favorites["sport"] = "Hockey"
```

Finally, *Symbols* are string-like identifers that begin with a colon character instead of surrounded quotations. They are often used as human-readable hash keys, or other situations where a value-agnostic, readable constant identifer is desired.

```
favorites = { :sport => "Hockey", :color => "Blue" }
favorites[:snack] = “Cookies”
```

There is a special, optional notation when specifying hash pairs when the key is a symbol:

```
favorites = { sport: "Hockey", color: "Blue" }
favorites[:snack] = "Cookies"
```

Model-View-Controller Architecture

Large computer programs that are written in an object-oriented language typically choose a specific *architecture*. Unlike procedural languages, object-oriented code tends to be distributed across a multitude of files, classes, and organizational units. Object-oriented programs are written so that a particular objects collaborate together in response to some kind of system event.

In web applications, the typical precipitating event is an incoming HTTP request, received first by a *web server* that can receive network packets over a live internet connection. The web server's responsibility is to hand the request to the application, so that it can be interpreted and transformed into an appropriate HTTP response.

HTTP requests appear (from the point of view of the application) to arrive asynchronously, and therefore an event-driven paradigm is the predominant approach in modern web frameworks. Once the event can be transmitted to a reception point inside the application, the internal structure of the application from that point forward is constrained only by the developer's preferences.
However, the Rails framework takes a further, bold step in dictating the internal structure of every Rails web application. Every Rails application must adopt a traditional *Model-View-Controller*, or MVC, structure.

MVC architectures seek to simplify development by dividing all of an application's objects into one of three primary categories:

- **Models**: objects which represent elements in the application's problem domain, encapsulate business rules or domain logic, and can sometimes also act as a data persistence layer. In Rails, models are responsible for all of these activities.
- **Views**: objects which generate the content of the actual HTTP responses. These are typically objects which can generate HTML, often with added capabilites such as embedding flow control logic and other higher-level functions. In Rails, views are typically HTML templates that can optionally

contain embedded Ruby, though sometimes views are primarily Javascript instead of HTML.

- **Controllers**: objects which serve as the “brains” of each request-response transaction. Controllers are request first-responders, and as such have access to the underlying http request and response streams. Controllers typically contain little domain logic and do not render views, but instead coordinate the overall collaboration between the models and views necessary to achieve a response that is appropriate to the incoming request.

In Rails, a fourth component, *routing,* is considered to be just as fundamental these classic three layers. Rails *routes* describe the complete set of HTTP request pattterns that the application can support. Routes connect a raw HTTP request to a specific method in a specific controller class, and fulfilling the role of the all-important first step in an event-driven HTTP framework.

Models

The primary responsibility of a model is to represent an element of the application domain (sometimes called the problem domain, or just domain). Models also serve as a data persistence layer between the application and a relational database system. Domain logic (sometimes called business logic) generally belong to models as well.

Models are Ruby classes which derive from ActiveRecord::Base, a class provided by Rails (specifically, the *activerecord* gem). ActiveRecord provides automatic object-relational mapping to a large number of relational database systems including SQLite, MySQL, Postgresql, MS SQL Server, Oracle, DB2, and others.

Models are said to be associated to one another when explicit declarations are provided inside the model class definitions and are supported by underlying foreign keys in the appropriate database tables.

Database tables are mapped to Ruby model classes by simple naming conventions. For example, given a Ruby model class named *Product*, automatic mapping to a table named *products* is provided without any configuration or extra specification. Note how the singular form in the class name (“*Product*”) maps to the plural form of the database table (“*products*”). This is one of the most important conventions in Rails applications.

Columns in tables are mapped to methods of a mapped Ruby model instance. For each column in the table, two methods are generated: one to retrieve the field value and one to set the value. These methods are named *column* and *column=*.

Here is an example of a *Product* model, mapped to a table named *products,* that in turn contains columns named *title* and *price*:

```
class Product < ActiveRecord::Base
end

# Insert a new row into the ‘products’ table
p = Product.new
p.title = “Yamaha Piano”
p.price = 2000.00
p.save

# Retrieve the last product in the table
# and display the title
p = Product.last
puts p.title
```

Rails provides a Ruby-based system of managing relational database schema definition through the invention of *database migrations*. A migration is simply a Ruby class that contains methods which, when executed, will issue the appropriate DDL statements to the database to create tables, drop tables, add columns, rename or drop columns, specify database indices, and more.

Models declare their relationships to other models by using a miniature domain-specific language suited to the task. Here is an example of how a product is identified as belonging to one specific category, and how a category can be associated to many products:

```
# A category can be associated to zero or more products.
class Category < ActiveRecord::Base
  has_many :products
end

# The ‘products’ database table contains a foreign-key
# column named ‘category_id’.
class Product < ActiveRecord::Base
  belongs_to :category
end
```

Associations do not invent relationships in the underlying table structures, but rather affirm to the Rails framework that the appropriate foreign keys and relationshsips already exist among the tables that underly the models. Foreign key columns are typically generated by their inclusion in the migration files as needed.

Models can implement business logic by user-defined instance methods, callbacks, and validation rules. ActiveRecord objects inherit a domain-specific language for specifying the callbacks and validation rules that should be automatically invoked by inherited persistence layer mechanisms.

Views

The primary responsibility of a view is to provide an HTML response, or more commonly, a combination of HTML markup augemented by Ruby code to provide conditional logic and data structures that serve the purposes of the view.

Views in Rails are HTML files that can optionally contain embedded Ruby statements to control template logic, evaluate variable expressions, and invoke external Ruby methods. Views typically rely on model instances to provide that data necessary to properly generate a particular web page or web page fragment.

Views, also called view templates, or simply templates, usually have end with the composite file extensions *.html.erb*.

Ruby statements can be embedded inside HTML templates by using one of two notational styles. Both use percent signs to enclose arbitrary Ruby expressions. Prepending an equal sign in front of the Ruby code will emit the result of Ruby expression into the HTML stream.

<% *Ruby code* %>
<%= *Ruby expression inserted into HTML output stream* %>

Here is an example utlizing both techniques. Notice the interleaving of HTML markup and embedded Ruby statements. For purposes of this example, imagine that @products is a Ruby variable that holds an array:

```
<ul>
  <% @products.each do |product| %>
    <li><%= product.name %></li>
  <% end %>
</ul>
```

View *partials* are views whose filename begins with an underscore. Partials are used to DRY up view logic and markup that might otherwise have to be repeated among two or more primary views. Including a partial into a main template is achieved by using the *render* method from inside a main template or another partial:

```
render “product”
```

The above code would find a file named _product.html.erb in the same folder as the containing view and render its markup into the HTML output stream. This is just one possible example of using the render method to reuse partials.

Controllers

The primary responsibility of a Controller is to service incoming HTTP requests and generate the appropriate HTTP responses.

Controllers define one or more instance methods, which are invoked automatically by the Rails framework when certain HTTP requests are received. (The exact rules that govern which HTTP requests are mapped to which controller methods are covered in the section on *Routes*.)

Controllers are Ruby classes which derive (ultimately) from ActionController::Base, a class provided by Rails (specifically the *actionpack* gem). ActionController provides built-in behavior for parsing HTTP requests, rendering HTTP responses, inspecting incoming data, and preparing data structures for use by corresponding view templates.

Controller classes contain instance methods referred to as *actions* or action methods. Actions use models to retrieve or update data held in the database, or perform some other kind of work as appropriate based on the request.

Actions commonly use the inherited *params* method to inspect a hash of incoming request data. The hash may contain querystring data, dynamic URL path segment values, and HTML form submission data.

Actions can also use the inherited *respond_to* method to provide a set of possible responses based on the MIME format specified in the request.

Actions must generate an HTTML response, and can utilize one of four techniques:

1. Explicit generation of an http response, by calling the *render* method and passing a hash of HTTP response headers and body content.
2. Explicit rendering of a view template, by calling the *render* method and passing a relative path or fully-qualified path for a view template filename.

3. Implicit rendering of a view template that can be located automatically by the Rails framework.
4. Explicit generation of a redirect response by using the *redirect_to* method.

Implicit rending is very common and is easily achieved by adhering to Rails file naming conventions, specifically: the base part of the view template filename is identical to the method name; and the view folder containing the template is named according to the controller class name.

Action methods can create instance variables to hold data required by a corresponding view template. Instance variables created in action methods are directly accessible by embedded Ruby contained in the the view template.

Controllers can designate certain methods as *filters* rather than action methods. Filters are callbacks which are automatically invoked before, after, or “around” action methods. Filters have the opportunity to abort (and hence, filter) incoming requests, usually based on user authorization criteria. Filters can also be leveraged to eliminate duplicate code that can accrue across action methods of the same controller.

Routes

The primary responsibility of a Rails route is to connect an incoming HTTP request with a defined controller action method.

Routes are critical because they provide the glue between incoming web requests and all application functionality.

The routes defined by a Rails application constitute the entire set of URLs, or more precisely, the entire set of HTTP method-URL combinations, that can be recognized. Rails reads the route set when the application is first loaded into the app server software (typically, this is started by the *rails server* command in development mode; in production, this is typically an app server daemon such as *Passenger* for Apache).

Each route connects a unique HTTP method + URL pattern combination to a single controller method. The general format of a route is as follows:

[http method] [url pattern] [action method descriptor] [route name]

For example, here are two routes:

```
get “/products”, :controller => ‘products’,
                 :action => ‘index’,
                 :as => ‘products’

get “/products/:id”, :controller => ‘products’,
                     :action => ‘show’,
                     :as => ‘product’
```

In the second case, the :id portion of the string will be interpreted by the routing table as a *dynamic segment* or *route placeholder*. Such placeholders will have their actual values at runtime available for inspection in the params hash, where the hash key is the placeholder identifier specified in the route definition and the hash value is the actual URL value.

Routes are unique combinations of an http method and associated URL pattern. Two routes cannot share the same combination. Routes are ordered in top-down fashion and take precedence over any potentially matching patterns below them.

Routes can have optional segments, placeholder segment constraints, placeholder default values, subdomain constraints, and more.

A *resource* is a specific collection of seven convention-based routes. Taken together, they constitute the basic CRUD actions typically used in many web applications. These seven routes can be specified with a single route definition:

```
resources :products
```

The *rake routes* utility can be used to display a comprehensive listing of all of the interpreted routes defined by the application.

View Helpers

Views can sometimes become an unwieldy mixture of Ruby code and HTML markup. The markup should be the the predominant feature of a view, but Ruby code is often needed for flow control, conditional logic, or to retrieve model data. Complex views can become overrun with too much logic, masking the primary purpose of the view.

View helpers are Ruby methods that can assist in encapsulating business rules or logic decisions, and can be invoked from within a view. Refactoring complicated view templates by using view helpers can result in cleaner code that is more understandable and maintainable. View helpers can generate markup, perform conditional logic, collaborate with models, and even access the underlying HTTP request data.

View helpers are Ruby methods contained in a view helper module. Here's an example:

```
module ApplicationHelper

  def show_shopping_cart(cart)
    content_tag :ul do
      cart.items.each do |item|
         content_tag :li, item.title
      end
    end
  end

end
```

Assets

Web applications typically deliver a payload of “static assets” with each page: images, CSS files, and Javascript code. In medium-to-large scale applications, the delivery of these assets can become a bottleneck in overall page performance in production environments.

Rails has built-in optimizations that help reduce amount of traffic required between the browser and server by providing an *asset pipeline*.

The pipeline is activated only in production environments. The basic idea is that all Javascript files are combined into a single, unified file, which is compressed as much as practically feasible; and the same is done with all of the application’s stylesheets.

The browser is then instructed to retrieve these two compressed files with each page load. Combined with HTTP cache headers, this typically results in just one download of the full set of Javascript and CSS, resulting in slower load time for the very first request, but considerably faster for each subsequent request.

Code Generators

Most of the code the Rails developers write is done by hand. However, Rails provides some amount assistance when you're first writing an important component of your application.

App Skeleton Generator

This is the best way to start a new Rails application from scratch:

```
rails new MyApp
```

This will create a new subdirectory named *MyApp* containing a skeleton Rails app.

Use **rails --help** for a list of options that you can use to customize the code that is initially generated.

Component Generators

These generators aren't intended to generate complete, shippable code; but rather, simply remove some of the tedium when you're starting a new component of your application.

To get a list of all generators:

```
rails generate
```

To get help on a specific generator, provide the generator name followed by the --help option. For example, here's how to get help on how to use the model generator:

```
rails generate model --help
```

You can also abbreviate the word generate with the letter g:

```
rails g model --help
```

Migration Generator

The migration generator is an important one: it generates *database definition files*, which can be executed with the `rake db:migrate` command. Database definition files can be used to add, remove, or modify the tables and columns in your database. Migrations enable the developer to evolve the database scheme in small steps, with the ability to version-control, rollback, and roll forward as needed:

```
rails g migration [MigrationName] [attribute list...]
```

You must provide a migration name, but the attribute list is always optional.

Creating a New Table

For example, here's how to generate a migration for a new *products* table with columns *title, price, and in_stock*:

```
rails g migration CreateProducts title:string price:integer
```

The generator only writes the definition code for you; it *does not* actually execute the defintion.

It saves the code in a file named something like db/migrate/20130401023121_create_products.rb, where the first part of the filename is a date-timestamp. If you open up this file in your editor you will see something like this:

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :title
      t.integer :price
    end
  end
end
```

Remember, your database is *not* changed as a result of running the migration generator. All it did was write this code file. To execute the code, see the `rake db:migrate` command.

Modifying an Existing Table

Sometimes you need to modify an existing table. Migrations help there too. Let's say you have a model named *Book* and you want to add an *editiion* column to it.

```
rails generate migration AddEditionToBook edition:integer
```

Again, open up the generated code and inspect it before you execute it.

Model Generator

```
rails generate model [ModelName] [attribute list...]
```

Generates a model class; a database migration; and skeleton unit tests.

Example:

```
rails generate model book title isbn author_id:integer
```

Controller Generator

```
rails generate controller [ControllerName] [action list...]
```

Use the plural form for the controller name. Generates a controller class, with empty methods for each action, placeholder view templates for each action, and a GET route for each action.

```
rails generate controller Books index show
```

Rails Conventions Summary

1. CamelCase to underscore_case, to connect classes with filenames
2. Plural names for controllers
3. Singular names for models
4. Implicit view rendering
5. Name of controller class
6. Form element names with [] becomes hash in params
7. Filename of view folder
8. Filename of partials (start with underscore)
9. Implicit partial iteration via collection
10. Rendering a partial with object instead of filename
11. Named route methods (_url and _path)
12. form_for expects named routes
13. Primary key is “id”
14. Foreign keys; *other_model*_id
15. resources :things
16. Route abbreviated syntax (products#index)
17. Overall MVC folder structure