

Ruby On Rails - <http://www.rubyonrails.org/>

DRY & Database Features - <http://oomoo.wordpress.com/2008/03/05/a-model-is-a-class/>

Fixture - <http://manuals.rubyonrails.com/read/chapter/26>

Partials - /app/views/tblweb_part/_tblweb_parts_list.html.erb

Parts of a View - <http://oomoo.wordpress.com/2008/03/06/a-view-in-pieces/>

Rails "Packages"

ActiveRecord

Object Relational Mapping (ORM) to represent data (tables) as objects with associated methods for data manipulation.

(<http://oomoo.wordpress.com/2008/03/05/a-model-is-a-class/>)

ActiveResource

It lets you declare and consume web-services using an ActiveRecord-like interface.

ActionPack

Splits the response to a web request into a controller part (performing the logic) and a view part (rendering a template). This two-step approach is known as an action

(<http://ap.rubyonrails.com/>)

ActiveSupport

A collection of various utility classes and standard library extensions that are found useful for Rails.

(<http://api.rubyonrails.org/classes/ActiveSupport/BufferedLogger.html>)

ActionMailer

Framework for handling sending/receiving e-mail.

(<http://wiki.rubyonrails.org/rails/pages/ActionMailer>)

MVC: Models - Views - Controllers

The MVC architecture is a set of design patterns that allows the separation between data models, user interfaces, and the control logics of the application. To make it clear, these three components are described as:

- *The model*, this layer contains codes that operate on the application data. Any actions that wanted to be executed on the raw data must go through this layer. Definitions of how the application work with data (commonly retrieve, create, update, or delete) are written here.
- *The view*, this is the presentation layer. It defines how your pages should look like to the user, how the application presents data, or how a user can submit certain instructions to be executed by the application.
- *The controller*, this component acts as the orchestrator of the application. It controls the flow of the program. It receives user commands, processes it, and then contacts the model, and finally instructs the view to display appropriately to the user.

Rails and MVC

Rails implements the concepts of MVC quite straightforward. The most visible implementation is perhaps the way Rails structures the application files and directories. One of the directories Rails creates is the `app/` directory; this is where most of the application stuff will go. It contains subdirectories in which developers put their models, views, and controllers of the application.

- `controllers/`
Contains Ruby files of the application's controllers.
- `helpers/`
This directory contains helper files that are usually related to presentation logic. Developers can put application-wide helpers or controller-specific helpers in this directory.
- `models/`
The model definitions are saved here, each data structure in the application represented as a Ruby class saved in a file. Developers would modify the auto-generated models to meet their application specific requirement in these files.
- `views/`
By default, each controller assumed to have its own presentation layer. Thus under the `views/` directory, there are subdirectories named accordingly to controllers that has been created. These subdirectories contain the views for a number of methods defined in the controller class. However, developers can change this default behavior by applying certain command in the controller. The view files are usually special HTML files (`.rhtml`) that can contain embedded Ruby code in it.

Underneath, Rails also has different modules that represent the three MVC components. These modules are:

- **Models: ActiveRecord**
ActiveRecord is designed to handle all data-related processes in the application. This module contains an Object/Relationship Mapping (ORM) library that can be used to abstract how data is actually accessed in the lower level. ORM library maps data stored in a database to a class in an application. The rest of the application components will remain agnostic to the database. This loosely-coupled design will in turn increase the scalability of the application.
- **Controllers: ActionController**
ActionController is the component that handles user requests and facilitates communication between models and the views. It is responsible for, among other things: routing external requests into internal actions (think friendly URLs), managing Rails caching features, managing helper modules, managing user sessions, etc.
- **Views: ActionView**
This library is responsible for creating either a part or all of a page to be displayed to users. Rails uses template system to generate dynamic content of views. There are three templating schemes known in Rails:
 - `.rhtml`, generates HTML views to users.
 - `.rxml`, lets developers construct XML documents using Ruby code, used widely to generate feeds in Rails-powered blogging engines.
 - `.rjs`, allows developers to create dynamic JavaScript codes, suitable for creating an AJAX interface.

Note that both ActionController and ActionView libraries are packaged together in one module called ActionPack module.

Conventions over Configurations

The concept conventions over configurations comes from the fact that Rails assumes a number of defaults. For instance is Rails naming conventions regarding database tables and the generated models. When a developer execute the generate model command `BlogComment` for instance, Rails will look into the database for table named `blog_comments` (note that Rails also aware of plurals), investigate it, and then generate the ORM class for that particular table.

Similarly, if we have a controller named `Archive`, then Rails assumes:

- The class is called `ArchiveController`, saved in the file `archive_controller.rb` within the `app/controllers/` directory.
- That there is exist a helper module called `ArchiveHelper` in the file `archive_helper.rb` within the `app/helpers/` directory.
- The templates for the views of this controller reside in the `app/views/archive/` directory.

This conventions over configurations principle prevent developers from making excessive configuration files, which could easily grow and hard-to-maintain as the project becomes larger. However, there are some configuration files Rails needed in order to run. These configurations are mainly about basic application parameters, such us runtime environment parameters and database connection configurations.

DRY: Don't Repeat Yourself

DRY (Don't Repeat Yourself) sounds quite self-explanatory, developers are encouraged not to express a piece of information/knowledge/code repeatedly in more than one place. DRY principle can found anywhere within a Rails application.

Rails view component supports partial template mechanism in order for developers defined a part of the overall page layout just once, to be included many times within different page layouts.

Another excellent example is the migration feature. When a developer for one reason needs to change the database structure, all he or she has to do is modify or create a new migration script and execute it using the `rake::migrate` tool, the changes reflect not only in the database but also in the model definition.

This principle will lead to an easy-to-maintain application, where changes in the application can be done effortlessly.

Rails also comes with a bunch of useful generators to provide developers the skeleton of the application, saving them a lot of time and in the same time giving a way to further develop it cleanly.