



Why Use Merb?

Architecture

Like Ruby on Rails, Merb is an MVC framework. Unlike Rails, Merb is ORM-agnostic, JavaScript library agnostic, and template language agnostic, preferring plugins that add in support for a particular feature to trying to produce a monolithic library with everything in the core. In fact, this is a guiding principle of the project, which has led to third-party support for the ActiveRecord, DataMapper, and Sequel ORMs for Ruby in advance of Merb's 0.4 release.

Merb is designed to be modular from the start. A number of features that are built into Rails, such as form helpers, are available as official plugins to Merb, which forced the project to develop entry points into the core. It's possible to modularize controllers and views through Merb plugins, and both Mailers and Parts (more later) take advantage of this modularity to bake in support for the controller architecture throughout the framework.

Plugins

Plugins in Merb are implemented as simple gems, which are distributed in the svn repository, by third parties, or via the Merb Plugin Nursery on RubyForge. That means that plugins can take advantage, out of the box, for Rubygems' versioning and dependency control. Plugins can either be installed to the system's repository or burned into the gems directory in an application's distribution; Merb applications simply add the **/gems** folder as an alternate repository.

Plugins for Merb already include support for ActiveRecord, DataMapper, and Sequel, with support for SQL sessions, model generation, and database.yml baked in to all three. The Merb svn also already includes a helpers plugin (to add support for Rails-style form helpers).

Controllers

Merb's controllers are made up of two components. First, an AbstractController, which handles layout- and template-finding, instance variable assignment, and before/after filters. Second, a Merb::Controller, which handles request/response semantics. Because the

components are separate, it is possible to inherit from `AbstractController`, which Merb does for Mailers and Parts (again, more on that later).

Controllers also support excellent content-type negotiation. You can specify in your controllers, or in individual actions, what MIME-types should be supported via `provides :xml, :html`. Once that information is provided, the controller has a number of ways to automatically render the appropriate content. If a template called `foo.html.erb` exists, it will automatically be rendered for all content-type `text/html`, and so on.

Additionally, calling `render @object`, will call `@object.to_mime_type` (for instance, `@object.to_xml`). The mime-type chosen in both cases is based upon either the file extension (`foo.html` maps to the `:html` type), or the `Accepts` header (the first acceptable content-type that's also in the `provides` list). If the object doesn't have the appropriate method, render will fall-back to rendering a template (so if the user requests, say, HTML, the lack of `#to_html` on the object will cause the `foo.html.erb` template to be loaded).

Exceptions

Merb also handles exceptions interestingly. Instead of exception raising throwing an error in your application, Merb catches certain types of exceptions and allows you to handle them in a controller/view fashion. For instance, raising `NotFound` will call the `Exception#not_found` action, which you can customize as appropriate. Raising an error in this way will also send the appropriate error-code back to the browser.

All HTTP error codes are defined in Merb as Exception classes, so you can raise `NotAcceptable`, which will call `Exception#not_acceptable`, and return a 406 error to the client.

Mailers

Merb's Mailers are implemented on top of `AbstractController`, so you get all the default controller behavior (including templates, assigns, and before/after filters) for free in the Mailer. But instead of calling `render`, you call `render_mail`, which takes options like: `render_mail :html => :foo, :text => :bar`. A number of options are

supported, including attachments via an **#attach** method, so you can build up your multi-part mails with attachments and site-wide layouts fairly trivially.

Mailers have their own root directory, which contains controller classes inside it, as well as a **views** directory (which contains **layouts**, just like a regular controller), and an optional **helpers** directory. Because **AbstractController** can specify its layout root trivially, it's easy to create new controller types and drop them in.

Mailers are called from a regular controller via **send_mail Klass, :action, options**, where options is a hash of options such as **from, to, subject, and cc**.

Parts

Like Mailers, parts take advantage of the flexibility of the **AbstractController** to allow simple Controller/View delegation. Parts have a directory structure that's identical to the Mailer structure, and you can use them to separate out logic about partials that are used throughout your app.

For instance, you might have a tag-cloud that appears app-wide. You could create a TagCloud part, and have actions in the part set up the controller logic for the template. Like controllers and Mailers, parts can have layouts, templates, and before/after filters. Parts are called via **part TagCloud => :show**. Parts can also be used just to segment out the logic for sections of partials, so you have discrete components instead of one massive controller.

Tests and Specs

Merb is testing-framework-agnostic: you can use Test::Unit, Rspec, or test/spec. All three testing frameworks have built-in support for mock objects that allow you to micro-target your tests exactly as you like. And because Merb is so modular, it's easy to test your controller without a request object at all, if you'd want to.

Generators

Merb has a series of generators that allow you get up and running quickly. The Merb application generator is started via **merb -g app**. It'll build a skeleton app that includes folders for controllers, parts, and Mailers. Building a plugin is easy as well, **merb --**

Why Merb?

generate-plugin merb_plugin_name will produce a very simple plugin skeleton with a few rake tasks to help you deploy the plugin (such as **rake package** and **rake install**).

In addition, Merb has controller generators, which create an empty controller file, a view directory with an empty **index.html.erb**, an empty helper file, and a test file in your chosen spec framework. Merb also has model generators, which are implemented by the ORM plugins, and support a special syntax: **script/generate model foo bar:string baz:integer** will generate a new model that implements those attributes using its own syntax. For instance, ActiveRecord would generate a migration, while DataMapper would generate new model using its property syntax.