



## Table of Contents

1. Background (p. 1)
2. Introduction (p. 1)
3. An App in 30 Seconds (p. 3)
4. A Walking Tour of Miso (p. 4)
5. Further Reading and Credits (p. 10)

### 1. Background

Miso was created by a couple Ruby developers for a project that, due to a client requirements, had to be completed in Java. After a couple hours of searching, we were unable to find a lightweight web framework with the simplicity of Sinatra or Merb. The most popular alternatives seemed laden with complex configurations, IDE dependencies / complex build processes, and required mountains of code to accomplish simple things.

Miso is also about helping to bridge the gap between the Java and Ruby communities. Both are good, stable languages with independent strengths. Java's static typing and verbose syntax enable an intelligent compiler to produce lightning-fast bytecode on a smart VM. Ruby's flexibility and expressive syntax empower developers to create simple DSLs for complex tasks. We hope that Miso can help Ruby developers to understand the speed and power of Java, and that it might show Java developers a simpler way to get things done. Regardless of your background, miso is designed to get you started, then get out of your way.

Finally, Miso is nearing completion. Our aim is to take care of the very basics – and nothing more. With the sparseness of a sand garden, you're welcome to build what you like. Be sure to read the Miso code, though. Good documentation is helpful, but nothing can replace reading the code itself.

### 2. Introduction

#### Code Less –

Miso's simple. No XML configuration files, no `AbstractQueryProcessorManagerFactoryFactories`, no complicated build process (just one command), and deployments are a snap. Superlatives are annoying, but it's arguably the simplest, most flexible Java MVC framework possible. An entire app with basic CRUD is 275 lines of code (plus your templates).

#### Do More –

That 275 lines of code includes all of Miso itself, and is entirely generated for you as a basic scaffold. Just pull it out of the box, type one command, and you've got a sample app ready to rock.

Throw it all away and write your own stuff from scratch if you like. Like Ruby microframeworks such as Sinatra or Merb, Miso gets out of your way and lets you write your app your way, with almost no overhead.



## It's Zippy –

Serving up a high-traffic site or high-performance API that demands dynamic requests and fast response times? Light-speed JSON output comes built into Miso. Rendering a short list view of objects pulled on-demand from a database fires requests out with a round-trip response time of just **1.25 milliseconds** each.

## Short Stack –

Of course, we couldn't do all of this with such little code. Miso uses a couple lightweight, well-tested libraries to help out with the heavy lifting. ActiveObjects is a simple ORM that provides object-mapping and associations (one-to-many, many-to-many, and polymorphic), JSON-Simple provides JSON output, and a couple pieces from Apache Commons saved us from reinventing the wheel. That's pretty much it.

## What's in the Box –

- A basic model/view/controller framework that provides parameter-based URL routing and class + method-binding.
- A simple but powerful ORM compatible with MySQL, Postgres, MS SQL Server 2005/2007, Oracle, HSQLDB, and Derby. Miso is bundled DBPool for database connection pooling.
- Code generation for basic Ruby on Rails-like scaffolds. Create the classic “Recipe Book” app in 30 seconds.
- Automatic API generation. The “index” and “show” actions come with JSON output built-in. By appending `?format=json` to a request, you'll get back a JSON representation of the data.
- That's pretty much it. Miso lets you forget about the boring stuff and concentrate on writing your app. It's more fun when stuff gets out of your way.
- Miso is bundled with an unmodified Jetty 6.1.22 for quick building and testing, but you can deploy to any servlet-based Java web platform.

## Dependencies –

Gross!

Seriously, though - we're working with Miso on Mac OS X 10.6 (Snow Leopard). You'll need Java 6 (installed by default), and MySQL (or another supported database, with a simple change in the code). The code generators (`script/generate`) are powered by Ruby, which is also installed by default on Macs. While Miso should run fine on any POSIX-based platform with similar resources available, there is nothing inherently platform-specific about the code produced. The build script is a Bash shell script, which should run fine on a Mac, Linux, or potentially in Cygwin, but writing a Windows batch file port should be trivial.



## 3. An App in 30 Seconds

1. Download Miso: <http://www.github.com/cscotta/miso>
2. This how-to assumes you're running MySQL with a root user using a blank password.
3. Unpack the archive, hop into Terminal, and type this:

```
$ script/generate scaffold Puppy name breed owner cuteness

MISO GENERATOR 2000 =====
=====

Creating model Puppy with columns: name breed owner cuteness

Generating Controller...
Generating Model Definition...
Generating Model Access Layer...
Generating Index Template...
Generating Show Template...
Generating Add Template...
Generating Edit Template...
Generating SQL...

Please execute the following SQL before starting the app:

CREATE DATABASE IF NOT EXISTS miso;
USE miso;
CREATE TABLE `puppy` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) DEFAULT NULL,
  `breed` varchar(255) DEFAULT NULL,
  `owner` varchar(255) DEFAULT NULL,
  `cuteness` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

Add the following line to app/Application.java in the 'Import Controllers' section:
import controllers.PuppyController;
```

4. Create the database using the SQL query that appears in the output.
5. Open up app/Application.java in your favorite text editor and add this line at the end of the "Import Controllers" section.

```
import controllers.PuppyController;
```

6. Type `script/build`, then browse to <http://localhost:8080/miso/app?controller=puppy&action=index>. Append `&format=json` to this URL to see a JSON version. To stop the server, type `script/stop`.



## 4. A Walking Tour of Miso

### How it Works –

Miso is designed much like Ruby web microframeworks such as Sinatra and Merb. The core ideas are simplicity and consistency - a lightweight framework that takes care of a few basic things and provides a structure for your app, then fades immediately into the background and lets you do your thing. If you're not familiar with the model / view / controller concept, consider reading the first portion of this article by Kalid Azad: <http://betterexplained.com/articles/intermediate-rails-understanding-models-views-and-controllers/>

Here is the relevant portion of it. Kalid has kindly offered to allow noncommercial republishing of it:

- The browser makes a request, such as <http://mysite.com/video/show/15>
- The web server (mongrel, WEBrick, etc.) receives the request. It uses routes to find out which controller to use: the default route pattern is “/controller/action/id” as defined in config/routes.rb. In our case, it's the “video” controller, method “show”, id “15”. The web server then uses the dispatcher to create a new controller, call the action and pass the parameters.
- **Controllers** do the work of parsing user requests, data submissions, cookies, sessions and the “browser stuff”. They're the pointy-haired manager that orders employees around. The best controller is Dilbert-esque: It gives orders without knowing (or caring) how it gets done. In our case, the show method in the video controller knows it needs to lookup a video. It asks the model to get video 15, and will eventually display it to the user.
- **Models** are Ruby [*ed: here, Java*] classes. They talk to the database, store and validate data, perform the business logic and otherwise do the heavy lifting. They're the chubby guy in the back room crunching the numbers. In this case, the model retrieves video 15 from the database.
- **Views** are what the user sees: HTML, CSS, XML, Javascript, JSON. They're the sales rep putting up flyers and collecting surveys, at the manager's direction. Views are merely puppets reading what the controller gives them. They don't know what happens in the back room. In our example, the controller gives video 15 to the “show” view. The show view generates the HTML: divs, tables, text, descriptions, footers, etc.
- The controller returns the response body (HTML, XML, etc.) & metadata (caching headers, redirects) to the server. The server combines the raw data into a proper HTTP response and sends it to the user.

It's more fun to imagine a story with “fat model, skinny controller” instead of a sterile “3-tiered architecture”. Models do the grunt work, views are the happy face, and controllers are the masterminds behind it all.

The structure of a Miso app is divided into a few folders:

Your application's code is located in the app/ folder. Inside app/ is Application.java, along with folders for your controller, models, and views. The “miso” folder contains two support files which handle request processing and rendering. You can ignore them.



# Miso

A simple web framework.

The simplicity of Ruby meets the power of Java.

Name	Date Modified	Size	Kind
app	Today, 11:28 PM	--	Folder
Application.java	Yesterday, 7:47 PM	4 KB	Java source
controllers	Today, 11:26 PM	--	Folder
PersonController.java	Today, 9:17 AM	4 KB	Java source
models	Today, 9:17 AM	--	Folder
Person.java	Today, 9:17 AM	4 KB	Java source
PersonModel.java	Today, 9:17 AM	4 KB	Java source
nacho	Today, 9:16 AM	--	Folder
Controller.java	Yesterday, 2:57 PM	4 KB	Java source
DispatchedRequest.java	Yesterday, 7:31 PM	4 KB	Java source
views	Today, 9:17 AM	--	Folder
404.jsp	Yesterday, 1:25 PM	4 KB	Java Server Page
500.jsp	Yesterday, 1:25 PM	4 KB	Java Server Page
includes	Yesterday, 1:25 PM	--	Folder
person	Today, 9:17 AM	--	Folder
lib	Today, 9:16 AM	--	Folder
public	Yesterday, 1:25 PM	--	Folder
css	Yesterday, 1:25 PM	--	Folder
index.html	Yesterday, 1:25 PM	4 KB	HTML document
script	Today, 9:16 AM	--	Folder
web.xml	Yesterday, 1:25 PM	4 KB	XML document

## Application.java

Application.java is the entry point to your application. It provides a servlet and accepts HTTP GET and POST requests on behalf of your application.

These requests are routed to your controllers by a method called `dispatchRequest`. `dispatchRequest` takes the request, parses the *controller* and *action* parameters, and uses reflection to discover, load, and invoke the right action in the right controller. If you're coming from the Ruby/Rails world, this is the equivalent of calling `constantize(params[:controller]).send(params[:action].to_sym)`. It's a simple way of binding requests to controllers and actions.

You'll only need to edit this file when adding new controllers. Just add a line such as `import miso.WhateverController;` in the `Import Controllers` section to ensure that your new controller is loaded along with the application.

## Controllers Folder

The controllers folder contains each of your controller files. These files should be named in the format of `[Model]Controller.java`, where `[Model]` is the capitalized, singular name of the data model you're creating. *Rails developers, note that in Miso there is no attempt to bind models, controllers,*



*and tables together using pluralization, which is somewhat unreliable and difficult to accomplish efficiently. They are simply independent Java classes.*

## A Sample Controller

Each controller defines a class that extends Miso's *Controller* class, opens a connection to the data access layer, and defines methods for each controller action/view pair you'd like in your application. Typically, these are defined in terms of RESTful operations. By default, Miso generates the following actions: `show`, `index`, `edit`, `add`, `create`, `update`, `destroy`.

Our loyalty in designing Miso has always been to clean controllers. Here's what a sample controller action looks like:

```
public class PuppyController extends Controller {
    static PuppyModel model = new PuppyModel();

    public void show(DispatchedRequest req) {
        Puppy puppy = model.get((String) req.getParameter("id"));
        req.setAttribute("puppy", puppy);

        if (req.getFormat().equals("json")) { renderString(model.toJSONString(puppy), req); }
        else { render("show", req); }
    }
}
```

The `show` action begins by fetching a `Puppy` object from the database by the ID passed in a request parameter (`?id=6`). This object is then made available to the view by attaching it to the `puppy` object by setting it as an attribute on the request.

If a `format` parameter is passed (`?format=json`), the action converts the `Puppy` object to JSON and renders that as the output. Otherwise, it renders the "show" template.

This convention should seem familiar to developers who have worked with Ruby frameworks such as Rails or Sinatra.

## Views

Views in Miso are standard JSP templates. Ideally, the view contains no significant logic and makes use only of data objects passed to it by the controller. A sample view for the `Puppy` example above might look like this:

```
<%@ page import="models.Puppy" %>
<% Puppy puppy = (Puppy) request.getAttribute("puppy"); %>

<jsp:include page="../includes/header.html" />
```



```
<h1>View a Puppy</h1>
<p>Here is all of the info we have about this puppy.
<a href="/java/app?controller=puppy&action=add">Add another puppy</a>, or
<a href="/java/app?controller=puppy&action=index">list all of them</a>.</p>

<p><strong>First:</strong> <%= puppy.getFirst() %></p>
<p><strong>Middle:</strong> <%= puppy.getMiddle() %></p>
<p><strong>Last:</strong> <%= puppy.getLast() %></p>
<p><strong>Email:</strong> <%= puppy.getEmail() %></p>

<form method="post" action="/java/app">
  <input type="hidden" name="id" value="<%= puppy.getID() %>">
  <input type="hidden" name="controller" value="puppy">
  <input type="hidden" name="action" value="destroy">
  <input type="submit" value="Delete" />
</form>

<jsp:include page="../../includes/footer.html" />
```

Note that the view imports the Model classes for each type of data it's displaying, then fetches the data objects passed to it from the controller. Database columns on these objects are rendered by calling `<%= object.getColumnName() %>`. There's not much to it.

## Models

Models are broken up into two components: the model definition, and the model's business logic / data access layer. Think of the model definition as providing getter and setter methods (in Ruby, `attr_accessors`) for each property you're mapping to a database column.

### Model Definition

A model definition in `ActiveObjects` looks something like this:

```
package models;

import models.*;
import net.java.ao.*;

public interface Puppy extends Entity {
  public String getFirst();
  public void setFirst(String first);

  public String getMiddle();
  public void setMiddle(String middle);

  public String getLast();
  public void setLast(String last);

  public String getEmail();
  public void setEmail(String email);
}
```



We merely define a list of `getColumnName()` and `setColumnName()` methods for each column in the data model. Make sure the types match - for example, specifying a `String` in the setter method indicates that you're mapping to a `varchar` or long text column in your database.

See the `ActiveObjects` documentation for more information on database mapping, including all type information as well as how to map associations across objects (one-to-many, many-to-many, and polymorphic): <https://activeobjects.dev.java.net/>

Recommended first reading is a concise six-page introduction to the ORM: <https://activeobjects.dev.java.net/ActiveObjects.pdf>

Once you dig in, be sure to check out `ActiveObjects`' `JavaDoc` here: <https://activeobjects.dev.java.net/api/overview-summary.html>

## Model Access Layers

Once you've defined your model, we need to implement our data access and business logic. This is separated from the model definition in a file named `PuppyModel.java` (where "Puppy" is your model name). While it would be possible to implement this logic in a controller, the controller would quickly become fat, wet, and soggy. This is undesirable.

By default, Miso generates `list`, `get`, `create`, `update`, and `destroy` methods for each of your models, as well as `toJSONString()` and `listToJSONString()` methods to render individual objects and collections out as JSON.

You'll likely want to implement additional query methods specific to the business logic of your application (specifically, those which require advanced `WHERE` clauses, ordering, and limits).

If your business logic is simple, your model access layer makes a cozy home for it. However, if you've got a lot to do, it might be worth breaking out into a separate class to better encapsulate the query methods and your logic rather than cramming everything together. That's up to you!

A sample query method might look like this (wrapped in its `Model` class for clarity):

```
public class PuppyModel {

    static String url = "jdbc:mysql://localhost/miso";
    static EntityManager em = new EntityManager(url, "root", "");

    // Find a single puppy
    public static Puppy get(String strId) {
        int id = new Integer(strId).intValue();
        return em.get(Puppy.class, id);
    }
}
```



This method lets you to fetch a Puppy object from the database by its ID.

## 5. Further Reading and Credits

### ActiveObjects

For information on implementing advanced queries, see the ActiveObjects documentation:

<https://activeobjects.dev.java.net/>

Recommended first reading is a concise six-page introduction to the ORM:

<https://activeobjects.dev.java.net/ActiveObjects.pdf>

Once you dig in, be sure to check out ActiveObjects' JavaDoc here:

<https://activeobjects.dev.java.net/api/overview-summary.html>

For information about the `HttpServletRequest` and `HttpServletResponse` objects wrapped up inside `DynamicRequest`, see the JavaDocs for each:

### HttpServletRequest:

[http://java.sun.com/j2ee/sdk\\_1.3/techdocs/api/javax/servlet/http/HttpServletRequest.html](http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/http/HttpServletRequest.html)

### HttpServletResponse:

[http://java.sun.com/j2ee/sdk\\_1.3/techdocs/api/javax/servlet/http/HttpServletResponse.html](http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/http/HttpServletResponse.html)

### JSP Templating Syntax:

<http://java.sun.com/products/jsp/syntax/1.2/syntaxref12.html>

### JSON-Simple Site and Documentation:

<http://code.google.com/p/json-simple/>

*Thanks to Fenchurch on Flickr for publishing the photo of Miso soup used in the header of this document with a Creative Commons license. Find it here:*

<http://www.flickr.com/photos/fenchurch/894515/>