Comparison of Web Frameworks Across Languages

Ian Will

Contents

List of Figures

List of Tables

Listings

Abstract

This paper discusses the roles of the basic components in a web framework and provides an overview of leading frameworks from four popular languages: Ruby on Rails, Django, CakePHP, and Spring Roo. It provides a high level introduction to web framework architectures by evaluating similarities and differences across four popular frameworks with a focus development efficiency and maintainability.

1 Introduction

Over the past decade the web has become a major platform for software products. Rapidly building flexible and efficient web sites and web-based APIs (Application Programming Interfaces) is a common and important development task. Frameworks are a critical underlying feature of almost all web applications. Implementing a web page sockets-to-html from scratch would be a Quixotic undertaking; there are much better things on which to spend developer time. Fortunately all major programming languages have many good web frameworks from which to choose. Frameworks are so important that for many web developers the choice of framework supersedes the choice of programming language!

Some popular frameworks include Ruby on Rails (for the Ruby language), Django (Python), Spring Roo (for Java), Apache Struts (Java), Enterprise JavaBeans (Java), Lift (Scala), Joomla (PHP), ASP.NET (Microsoft .NET), and CakePHP (PHP). There are many more. All frameworks implement the same basic components: routing, HTML generation, database access, automated tests.

The implementation of models, controllers, views, code generation, and routing will be discussed for each framework in turn. Samples will be motivated by a common application developed in each of the frameworks. This sample application is a blog consisting of posts, comments, and tags. Comments and tags are associated with a post. The web application provides views to read posts with associated comments and tags; and forms to create or edit posts, comments, and tags. Data is stored in a persistent data source (all four frameworks use relational databases).

An overview of the development process with each framework will be presented, with code examples drawn from the sample application. The overview will discuss the framework's history, philosophy, and work-flow; and will then present code examples of each major factor–model, view, controller, routing, and code generation.

The conclusion presents a quantitative comparison by presenting counts of source files, generated lines of code, and manually edited lines for an implementation of similar sample applications in each framework. The comparison will also qualitatively comment on the number of and type of errors that were made during development, the ability to comprehend how the framework behaves, and how well the framework encourages good development style.

2 Related Work

Bjorn Buchner et. al. [1] analyzed a number of modern Java-based web frameworks using the Analytic Hierarchy Process (AHP) to quantify trade-offs. Their evaluation criteria included such factors as AJAX support, license, testability, efficiency, documentation, and development effort. Their evaluation method scored Spring and Struts highly, though the metrics were skewed toward a particular client and project. The AHP method of quantifying strengths and weakness to analytically optimize trade-offs is an interesting approach to tackling the dizzying numbers of available frameworks.

Tony Shan and Winnie Hua [2] develop a taxonomy of Java web frameworks with categories for Request-based, Component-based, Hybrid, Meta, and Rich Internet Application (RIA)-based. They characterize Struts, Stripes, and Beehive as Request-based; JSF, Tapestry, and Wicket as Component-based; RIFE as Hybrid; Keel and Spring as Meta; and DWR, Echo2, and JSON-RPC as RIA-based.

L. Stella et. al. compare maintainability of J2EE, .NET, and Ruby on Rails frameworks [3]. They evaluated the frameworks based on modifiability, testability, understandability, and portability. Evaluation was based on change propagation analysis, testing and debugging framework capabilities, modularity (cohesion, coupling, cross-cutting), and lines of code across a small-scale CRUD style web application. They found J2EE more portable, but found Ruby on Rails more modifiable, understandable, and testable.

Matt Raible has a frequently updated presentation [4] comparing web frameworks based on a 20 point comparison matrix including such factors as Developer Productivity, Developer Perception, Learning Curve, Project Health, Templating, AJAX support, Scalability, Testing support, and Job Trends. He presents extensive analysis for most categories, including statistics on release frequency and benchmarks of requests-served-per-second. His ranking places Spring MVC, Google Web Toolkit, and Ruby on Rails as the top three ranked web frameworks with Grails (a JVM based Rails clone) a close fourth.

Hongjun Li compares the RESTfulness of Java-based web frameworks [5], including RESTEasy, Restlet, Struts 2, Grails, Axis2, Certia4, sqlREST, and REST-art. The paper presents an interesting discussion on what it means to be RESTful, and how well Java frameworks claiming to implement REST adhere to those ideals.

3 Ruby on Rails

Ruby on Rails was created by David Heinemeier Hansson while developing Basecamp for 37signals [6]. It was first released in July 2004, later revised to version 2.0 in December 7, 2007. It eventually merged with Merb–a competing Ruby-based web framework–producing Rails 3.0 in August 2010. Rails emphasizes popular agile software-engineering principles including Convention-over-configuration, Don't-Repeat-Yourself (DRY), the Active record pattern, and Model-View-Controller. It defines standard conventions for folder structures, MVC design, and flow of control and provides code and test generators that can automatically create the necessary classes, methods, and tests based on a handful of parameters. Many popular sites have been built using Rails, including Basecamp (the genesis of Rails), Twitter, LivingSocial, Github, and Hulu.

3.1 Rails Design

The Rails design is centered around a Model-View-Controller architectural decomposition. This separates data representation and persistence concerns into model classes, data manipulation and business logic into controller classes, and data presentation and user interface specifications into view classes. Rails uses Ruby classes for models and controllers, and HTML combined with Embedded Ruby for view specifications. Each category has its own subdirectory in under the app folder. Controller classes reside in app/controllers, model classes in app/models, and view files in app/views.

Rails is compatible with a Ruby API for web servers called "Rack". Rack is an API a web server to hand off HTTP messages to Ruby-based web applications. Rack handlers are available for most web servers–including Apache (by enabling mod_rails, mod_rack). Rails comes with the WEBrick web server ready to run for quick development. A WEBrick server running the Rails application can be started with the `rails server` command.

After an HTTP message is handed off to Rails, routes defined in Config/routes.rb are consulted to determine which controller method is called. Mapping to controller request is a function of both HTTP method (GET, POST, PUT, DELETE) and URL. The controller then interacts with the database through model classes. Instances of those model objects are then made available to the rendering code, which executes preprocessor engines on HTML with Embedded Ruby, Sass, and CoffeeScript.

HTTP Method	Path	Controller Action
GET	`posts/`	index
GET	`posts/new/`	new
POST	`posts/`	create
GET	`posts/:id/`	show
GET	`posts/:id/edit`	edit
PUT	`posts/:id/`	update
DELETE	`posts/:id/`	destroy

Table 1: Rails RESTful Routing [8]

The Embedded Ruby preprocessor can use the available model objects to generate HTML based on the current state of the model. It's similar to Java Server Pages, but with Ruby statements embedded in HTML instead of Java. The final output of the preprocessor engines is standard HTML, JavaScript, and CSS, which is returned to the requesting browser for display. Rails also includes support for rendering models as JSON, which is more convenient than XML for processing in JavaScript.

3.2 Rails Routing

Rails is designed to encourage development using REST (REpresentational State Transfer) [7] principles. Following from those recommendations, Rails applications use four HTTP methods (GET, PUT, POST, DELETE) in combination with a particular resource (URL) to specify a unique action. Requests using PUT, POST, and DELETE methods change the state of the application, while requests using GET have no side-effects. This facilitates caching and helps prevent unintended side-effects. Table 1 shows the mapping of HTTP methods and URLs to actions. The controller action column indicates method names on the corresponding controller (`PostsController` in this case).

Rails defines its routes in `config/routes.rb`. The router for the sample application is shown below. In the Rails router, lines beginning with `resources` tell the router to map URLs to controller actions using the pattern shown in Table 1. It's a convenient shorthand notation that saves a lot of manual configuration provided an application uses the Rails recommended routing style.

The `:comments` resource router is nested underneath the `:posts` resource, which matches paths like `posts/:post_id/comments/:id`. The line `root :to => ‘‘home#index’’` is a special case that tells Rails to route the root path to the `index` method in `home_controller.rb`.

```
Blog::Application.routes.draw do
  root :to => "home#index"

  resources :posts do
    resources :comments
  end
end
```

Listing 1: Rails Shorthand Route Definition

The `resources` shorthand is convenient in many cases, but Rails also provides a more flexible process for defining routes. Listing 2 shows the manual method for specifying routes that `resources :posts` would otherwise create.

```
Blog::Application.routes.draw do
  root :to => "home#index"

  match 'posts' => 'posts#index', :via => :get
  match 'posts/:id/new' => 'posts#new', :via => :get
  match 'posts/' => 'posts#create', :via => :post
  match 'posts/:id' => 'posts#show', :via => :get
```

```ruby
  match 'posts/:id/edit' => 'posts#edit', :via => :get
  match 'posts/:id' => 'posts#update', :via => :put
  match 'posts/:id' => 'posts#destroy', :via => :delete
end
```

Listing 2: Rails Manual Route Definition

3.3 Rails Models

Rails model classes extend from `ActiveRecord`. They make extensive use of Ruby helpers, so the resulting model classes are small. The `Post` model from the sample application is shown in Listing 3. These use Ruby labels along with Ruby's built in `attr_accessible` helper to create a model with fields content, name, and title with corresponding getters and setters.

```ruby
class Post < ActiveRecord::Base
  attr_accessible :content, :name, :title, :tags_attributes

  validates :name,  :presence => true
  validates :title, :presence => true,
                    :length => { :minimum => 5 }

  has_many :comments, :dependent => :destroy
  has_many :tags

  accepts_nested_attributes_for :tags, :allow_destroy => :true, :reject_if => proc{ |attrs|
      attrs.all? {|k,v| v.blank? }}
end
```

Listing 3: Rails Post Model

3.4 Rails Controllers

The Rails controller is the plumbing that connects the routing, model, and view. Rails controllers extend from `ApplicationController` and implement one method for each type of action specified in Table 1. If the controller is left blank, its default behavior calls render with a view file that matches the action name (e.g. index on `PostsController` would render views/posts/index.html.erb). More details on the interaction between Rails controllers and views can be found in [9].

A few methods from `PostsController` for posts from the sample application are shown in Listing 4. These methods interact with the `Post` model class to retrieve the current state, and make that data available to the view through controller instance variables. Variables are prefixed with an @ symbol such as `@posts` are class instance variables in Ruby.

```ruby
class PostsController < ApplicationController

  http_basic_authenticate_with :name => "admin", :password => "admin", :except => [:index, :
      show]

  # GET /posts
  # GET /posts.json
  def index
    @posts = Post.all

    respond_to do |format|
      format.html # index.html.erb
      format.json { render json: @posts }
    end
  end

  # GET /posts/new
  # GET /posts/new.json
  def new
    @post = Post.new

    respond_to do |format|
      format.html # new.html.erb
      format.json { render json: @post }
    end
  end

  # GET /posts/1/edit
  def edit
    @post = Post.find(params[:id])
  end

  # DELETE /posts/1
  # DELETE /posts/1.json
  def destroy
    @post = Post.find(params[:id])
    @post.destroy

    respond_to do |format|
      format.html { redirect_to posts_url }
      format.json { head :no_content }
    end
  end
  ...
end
```

Listing 4: Rails Post Controller

3.5 Rails Views

The render stage uses a pipeline of preprocessor engines to dynamically construct HTML. Rails uses file extensions such as .html.erb, to specify which preprocessor engines they will run through and it which order. The most common preprocessor engines used in Rails are .erb (Embedded Ruby), .scss (Sass), and .coffee (CoffeeScript).

Embedded Ruby allows dynamic HTML generation similar to Java Server Pages. Listing 5 shows the Embedded Ruby that renders a list of all posts. Embedded Ruby statements are placed in an HTML file and are designated using `<%=` ... `%>` to mark the beginning and end of an Embedded Ruby statement. Embedded Ruby code has access to the instance variables in the corresponding controller class.

```
<h1>Listing posts</h1>

<table>
  <tr>
    <th>Name</th>
    <th>Title</th>
    <th>Content</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>

<% @posts.each do |post| %>
  <tr>
    <td><%= post.name %></td>
    <td><%= post.title %></td>
    <td><%= post.content %></td>
    <td><%= link_to 'Show', post %></td>
    <td><%= link_to 'Edit', edit_post_path(post) %></td>
    <td><%= link_to 'Destroy', post, method: :delete,
    data: { confirm: 'Are you sure?' } %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New Post', new_post_path %>
```

Listing 5: Rails index.html.erb View

In addition to a flexible and efficient system for defining routes, Rails also provides convenience methods for specifying paths to those routes in Embedded Ruby. The `link_to` function can be used in conjunction with automatically created helper functions to specify paths to other pages in the system without explicitly entering those paths in the HTML. The helper function `new_post_path` corresponds to the route to the `new` action on `PostController`. The last line in Listing 5 inserts the HTML to create a hyper-link to the appropriate path.

3.6 Rails Code Generation

Rails provides many commands to generate models, views, controllers, database migrations, and tests from the command line. For example, the command to generate scaffolding for posts is shown below.

```
rails generate scaffold Post name:string title:string content:text
```

This generates a post model with name, title, and content fields; creates a corresponding controller with methods populated to respond to standard Rails style routes (as shown in Table 1); adds those standard routes to routes.rb; generates HTML views; and creates a database migration–totaling 334 lines of code across 17 files. That's quite a bit of work accomplished with a relatively succinct command. Rails assumes it knows what you want, and does it the Rails way. If you want what Rails thinks you want–namely a user-facing web interface for viewing, creating, and editing model data–it's an incredibly efficient way to work.

4 Django

Django is written in Python. It was developed by Adrian Holovaty and Simon Willson, and Jacob Kaplan-Moss as infrastructure for the Lawrence Journal-World newspaper in 2003 and 2004. Django was open sourced in July 2005–one year after Ruby on Rails. It is regarded as the Python analog to Rails, because it provides a "full stack" solution with a bit more (though minimal) "magic" than other Python based web frameworks. Django adopts similar design philosophy as Rails, including an MVC decomposition, emphasizing the DRY principle, and encouraging conventions over configuration. A notable difference in philosophy is that Django's design ethic attempts to maintain compatibility with generic Python and minimize reliance on framework-specific language features. Rails on the other hand has essentially built a domain specific language on top of Ruby, and relies on many special purpose helper functions.

The Django Book [10] is an excellent introduction to Django and web development in general. Holovaty describes the basic workings of the HTTP request response cycle starting with CGI scripts and builds up to Django from that base. He describes the design considerations and details of each Django component, incrementally building an understanding of Django that likely mirrors how the code was written.

Django's interpretation of Model-View-Controller is similar to Rails. The most notable difference is that Django's controller phase is must simpler than Rails'. Rails uses a controller class with eight methods (index, show, new, edit, create, update, destroy) corresponding to different HTTP methods. Django uses a tuple of regular expression and function pointer, using short circuiting to render the first matching view function.

4.1 Django Design

Django architecture also adheres to the Model View Controller pattern. Its models are defined models.py, its views are functions placed in views.py, and its controller logic is considered to be the URL mapping defined in urls.py. The flow of control when processing a request begins in urls.py where Django determines which function in views.py to call based on the URL. Once a view function is called, it uses model classes to query the database, interacts with request parameters, and returns an `HttpResponse`.

Django uses the Web Server Gateway Interface (WGSI) standard to communicate with a web server. WSGI is a simple standard that's a step up from CGI. It was established to foster broad web server compatibility across Python web frameworks.

4.2 Django Routing

Django's routing is specified as a list of tuples which map regular expressions for URLs to Python methods. These tuples are defined in urls.py. The urls.py file for the sample application is shown in Listing 6. Django's routing is determined solely by the URL–not the HTTP method–so implementing different actions for different HTTP methods must be done within a view function.

The first part of the URL tuple defines a regular expression. The second part defines the path to the view function. The dots separating the segments indicate directories, so `posts/new/` is associated with `'blog.views.post_new`, which maps to `post_new` in blog/views.py.

```
urlpatterns = patterns('',
    # Examples:
    url(r'^$', 'blog.views.home'),
    url(r'^posts/$', 'blog.views.posts_index', name=''),
    url(r'^posts/new/$', 'blog.views.post_new', name=''),
    url(r'^posts/(\d+)/$', 'blog.views.post', name=''),
    url(r'^posts/(\d+)/edit/$', 'blog.views.post_edit', name=''),
    url(r'^posts/(\d+)/delete/$', 'blog.views.post_delete', name=''),
    url(r'^posts/(\d+)/comments/(\d+)/delete/$', 'blog.views.comment_delete', name=''),
    url(r'^posts/(\d+)/comments/new/$', 'blog.views.comment_create', name=''),
    }
```

Listing 6: Django urls.py

4.3 Django Models

By convention, Django models are specified in a single file: models.py. Each model is a minimal Python class. The models.py for the sample application is shown in Listing 7. Models contain just enough information for Django to create and query database tables.

```
from django.db import models

# Create your models here.
class Post(models.Model):
  content = models.CharField(max_length=2000)
  title = models.CharField(max_length=50)
  name = models.CharField(max_length=50)

class Comment(models.Model):
  body = models.CharField(max_length=500)
  commenter = models.CharField(max_length=50)
  post = models.ForeignKey(Post)

class Tag(models.Model):
  name = models.CharField(max_length=50)
  post = models.ForeignKey(Post)
```

Listing 7: Django models.py

4.4 Django Views

The Django controller section was omitted, because Django designates what other frameworks consider controller responsibilities as view responsibilities. Django's "controller" layer is equivalent to what other frameworks consider the routing configuration. This section describes the functionality that Django places in the view layer, but in other frameworks this would cover both controller and view.

Django view functions are the recipients of routing decisions defined in urls.py. A sample view function that generates HTML forms for creating new blog posts is shown in Listing 8. This function is called in response to requests on the `posts/new` URL. It handles both HTTP POST and GET methods differently. When the HTTP message is sent via POST, the function requests use Django's form validation helpers to validate the form data. If valid, the data is inserted into the database. For GET requests, the function returns an unpopulated HTML form.

```
def post_new(request):
  if request.method == 'POST':
    form = PostForm(request.POST)
    if form.is_valid():
      cd = form.cleaned_data
      p=Post.objects.create(title=cd['title'],name=cd['name'],content=cd['content'])
    try:
      if reqeust.POST['new_tag']:
        tag = Tag.objects.create(name=request.POST['new_tag'], post=p)
    except KeyError:
      pass
    return HttpResponseRedirect('/posts/')
  else:
    form = PostForm()
  return render_to_response('post_new.html', {
    'form': form,
    'submit_text': 'Create Post',
    'show_tags': 'true'
    }, context_instance=RequestContext(request))
```

Listing 8: Django View Function

Django provides a number of helper methods that can be used by view functions to load HTML templates from files. These apply Django's built in code templates based on a provided variable map, and return the appropriate `HttpResponse`. One such helper is `render_to_response` shown in Listing 8.

Django provides a built-in templating language for dynamically constructing HTML. It's slightly more limited than the HTML templating in other frameworks. Rails and Spring Roo allow arbitrary function calls back into Ruby or Java code. Django templates only allow calls into Python functions without arguments. This limitation simplifies the implementation and encourages decoupling business logic and HTML representations. The Django recommendation is to do all necessary work in the Python view code to avoid needing to embed compliex logic in HTML templates.

```
<h1>Listing posts</h1>

<table>
  <tr>
    <th>Name</th>
    <th>Title</th>
    <th>Content</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>

{% for post in posts %}
  <tr>
    <td>{{ post.name }}</td>
    <td>{{ post.title }}</td>
    <td>{{ post.content }}</td>
    <td><a href="{% url 'blog.views.post' post.id %}">Show</a></td>
    <td><a href="{% url 'blog.views.post_edit' post.id %}">Edit</a></td>
    <td><a href="{% url 'blog.views.post_delete' post.id %}">Destroy</a></td>
  </tr>
{% empty %}

{% endfor %}
</table>

<br />
```

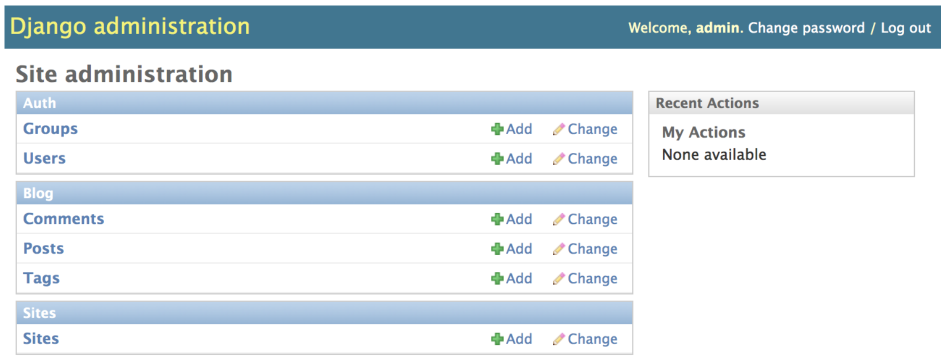

Figure 1: Django Admin Interface

```

<a href="{% url 'blog.views.post_new' %}">New Post</a>
```

Listing 9: Django HTML template

4.5 Django Code Generation

Django's utilities for code generation are more spartan than those of other frameworks. Django will automatically create and maintain database tables from models, but it won't manage database schema migrations. It will create empty starter files and a basic folder structure, but no HTML is generated. Because of this, the sample application required a bit more manual effort in programming the HTML. While Django lacks general-purpose HTML generation, it does have a built in library which generates a web-interface for model administration.

4.6 Django Admin Interface

Django has powerful tools for automatically building a web-based, access controlled, model-administration web interface. These are designed for an internal user (such as newspaper journalists and editors) rather than the general public and are always kept behind a login password. Enabling this admin interface in Django is as simple as uncommenting a few lines in settings.py and urls.py and running `python manage.py syncdb`. The resulting interface for the sample application is shown in Figure 1. This is a powerful and easily-created interface for managing models. However, it's not designed to be customized or converted into a user-facing web site.

5 CakePHP

CakePHP was initially developed as a rapid application framework called Cake by Michal Tatarynowicz in 2005. It was first open-sourced in December 2005 and CakePHP version 1.0 was released in May 2006. The most recent version at the time of writing is 2.2.3. CakePHP is modeled after Ruby on Rails. It provides similar convention-over-configuration model-view-controller architecture for rapid web development in PHP.

5.1 CakePHP Design

CakePHP borrows much of its design ethic from Rails. Each model is represented as a class, defined in the app/Model directory. CakePHP model classes extend a base Model class which provides the bulk of the functionality. Similarly there is one class for each controller, which reside in app/Controller. View logic is stored in CakePHP Template files (.ctp) in app/Views/Model directory (e.g. app/View/Posts for templates related to the Post model). Each view template relates to a particular controller method.

Flow of control behaves similarly as in other frameworks; routing logic determines which controller and method to call, the controller interacts with model classes to query the database, then hands off variables to view templates for rendering.

CakePHP can run on any server that supports PHP. Apache appears to be the most commonly used server. The setup for the sample application was more involved than for the other frameworks. CakePHP requires a running Apache server with PHP and mod_rewrite enabled. Other frameworks provide a lightweight web server and a script to start an instance running the project for testing and development.

The entire CakePHP code base must reside in Apache's web root directory, which presents some configuration and security problems during development. A number of unmentioned post-install steps are required to work out PHP errors–including explicitly setting time zone and giving write permissions on an embedded tmp directory to the www user (on unix based systems).

5.2 CakePHP Routing

CakePHP defines its routing information in app/Config/routes.php. Listing 10 shows a sample CakePHP routing file which redirects the root url to show the index of the posts. The listing also includes a URL that maps a path like `posts/3` to the view function with the id parameter. The default CakePHP route would look like `posts/view/3`. This will call `PostController.view` with id as an argument.

```
Router::connect('/', array('controller' => 'posts', 'action' => 'index') );
Router::connect('/:controller/:id/',
  array('action' => 'view'),
  array('pass' => array('id'))
);
```

Listing 10: CakePHP Routing Definition

5.3 CakePHP Models

CakePHP's approach to database table creation adopts a reverse work-flow from the other three frameworks. Rails, Roo, and Django all allow the programmer to define the model as classes and attributes in code and automatically create the appropriate relational database tables. CakePHP on the other hand will inspect an existing database and infer appropriate attributes for an otherwise empty class. The basic CakePHP tutorial has the reader manually create tables in MySQL before beginning to construct the model classes. The model can then be completely empty (as shown in Listing 11), with all attributes being inferred by database inspection.

```
class Post extends AppModel {
}
```

Listing 11: Empty CakePHP Post Model

CakePHP does provide a work-flow for managing database migrations by allowing the current model to be dumped to a PHP source file that lists all tables and relations. The command to dump the current schema is `app/Console/cake schema generate`. This creates a file in the Schema directory named schema.php which looks like Listing 12. The schema.php file can then be used to re-create the database using the command `app/Console/cake schema create`.

```
class AppSchema extends CakeSchema {

...

  public $posts = array(
    ’id’ => array(’type’ => ’integer’, ’null’ => false, ’default’ => null, ’length’ => 10, ’key’ => ’primary’),
    ’title’ => array(’type’ => ’string’, ’null’ => true, ’default’ => null, ’length’ => 50, ’collate’ => ’utf8_general_ci’, ’charset’ => ’utf8’),
    ’name’ => array(’type’ => ’string’, ’null’ => true, ’default’ => null, ’length’ => 50, ’collate’ => ’utf8_general_ci’, ’charset’ => ’utf8’),
    ’body’ => array(’type’ => ’text’, ’null’ => true, ’default’ => null, ’collate’ => ’utf8_general_ci’, ’charset’ => ’utf8’),
    ’created’ => array(’type’ => ’datetime’, ’null’ => true, ’default’ => null),
    ’modified’ => array(’type’ => ’datetime’, ’null’ => true, ’default’ => null),
    ’indexes’ => array(
      ’PRIMARY’ => array(’column’ => ’id’, ’unique’ => 1)
    ),
    ’tableParameters’ => array(’charset’ => ’utf8’, ’collate’ => ’utf8_general_ci’, ’engine’ => ’InnoDB’)
  );

  ...
```

Listing 12: CakePHP schema.php

Thus a work-flow in which the schema.php file is manually created, and the database tables subsequently created using CakePHP's schema migration scripts appears to be the only mechanism for automated database creation. This still requires the programmer to explicitly specify appropriate foreign keys to capture the relationships between models.

5.4 CakePHP Controllers

As in Rails, CakePHP controllers implement the glue and business logic necessary to pass model information to the views. There is one controller class per model, and routing logic maps URLs to a specific controller class and method.

CakePHP offers controller scaffolding for common database actions, which is available when the `scaffold` variable is declared. This creates index, view, edit, add, and delete actions for the controller. Scaffolded methods will be overridden by class implementations. An example of a CakePHP controller is shown in Listing 13. It uses the default scaffolding, but overrides the view function to redirect HTTP POST messages to the edit command.

```
1  App::uses(’AppController’, ’Controller’);
2
3  class PostsController extends AppController {
4
5    public $scaffold;
6
7    public function view($id = null) {
8          $this->Post->id = $id;
9          $this->set(’post’, $this->Post->read());
10         $this->set(’comments’, $this->Post->Comment->find(’all’));
11         $this->set(’tags’, $this->Post->Tag->find(’all’));
12   }
13   ...
14 }
```

Listing 13: CakePHP Post Controller

5.5 CakePHP Views

CakePHP view code is stored in HTML files with PHP markup with the .ctp suffix. Variables can be made available to the PHP view code by setting the appropriate value on the controller object, as shown on line 12 in Listing 13. An example of the view code that accesses this variable is shown in Listing 14.

```
<div class="posts view">
<h2><?php  echo __('Post'); ?></h2>
  <dl>
    <dt><?php echo __('Title'); ?></dt>
    <dd>
      <?php echo h($post['Post']['title']); ?>
      &nbsp;
    </dd>
    <dt><?php echo __('Name'); ?></dt>
    <dd>
      <?php echo h($post['Post']['name']); ?>
      &nbsp;
    </dd>
    <dt><?php echo __('Body'); ?></dt>
    <dd>
      <?php echo h($post['Post']['body']); ?>
      &nbsp;
    </dd>
  </dl>
</div>
<div class="comments view">
<h2><?php  echo __('Comments'); ?></h2>
  <?php foreach ($comments as $comment): ?>
  <dl>
    <dt><?php echo __('Author'); ?></dt>
    <dd>
      <?php echo h($comment['Comment']['author']); ?>
      &nbsp;
    </dd>
    <dt><?php echo __('Content'); ?></dt>
    <dd>
      <?php echo h($comment['Comment']['content']); ?>
      &nbsp;
    </dd>
  </dl>
  <?php endforeach; ?>
</div>
<div class="tags view">
<h2><?php  echo __('Tags'); ?></h2>
  <?php foreach ($tags as $tag): ?>
      <?php echo h($tag['Tag']['name']); ?>,&nbsp;
  <?php endforeach; ?>
</div>
```

Listing 14: CakePHP Edit View

5.6 CakePHP Code Generation

CakePHP provides a "bake" command to generate code. Running `app/Console/cake bake` launches a menu-based command-line console which has options to generate model, view, and controller files. Each option asks a series of yes/no questions allowing customization generated code including whether tests should be created, whether special admin views are desired, and the types of relations between models (hasOne, hasMany, belongsTo, hasAndBelongsToMany).

Using `bake`, most of the sample blog application was generated automatically. The generated post model is shown in Listing 15. The only line changes required were switching dependent to true so that comments and tags would be deleted when their parent post was deleted. Notice that none of the Post fields are explicitly listed in the PHP model code. These are determined from the database table columns.

```
class Post extends AppModel {

  public $displayField = 'name';

  public $hasMany = array(
    'Comment' => array(
      'className' => 'Comment',
      'foreignKey' => 'post_id',
      'dependent' => true,
    ),
    'Tag' => array(
      'className' => 'Tag',
      'foreignKey' => 'post_id',
      'dependent' => true,
    )
  );

}
```

Listing 15: CakePHP Post Model

6 Spring Roo

Spring Roo emerged on the web development scene much later than Django, CakePHP, or Rails. Its first alpha version was released in April 2009 and version 1.0 was subsequently released in December 2009. The current version (as of Nov 2012) is 1.2.2. It's main goals are to provide rapid development for Java based web applications without introducing a runtime layer and without re-inventing the rich collection of existing Java frameworks for object-relational mapping, persistence, or Web templating. Spring Roo documentation refers to the latter two goals as "Runtime Avoidance" and "Lock-in Avoidance". Thus Spring Roo is designed not to replace existing Java web development components–such as the Java Persistence API (which Hibernate implements), Java Server Pages, Java Servlets, and the Spring Framework. Instead, it provides a convenience layer for defining a model and generating persistence, view, and controller code that is compatible with the selected underlying components and connecting them with Spring.

Avoiding a runtime layer in Spring Roo allows applications to work well with other Java web development tools, it also avoids having that runtime layer become a performance bottleneck–an accusation that Rails has faced. Leveraging existing Java frameworks also helps ease adoption by using tools familiar to Java web developers and side-stepping some of the risk of new, unproven runtime infrastructure.

However these design decisions result in a steeper learning curve and a more complex development experience than more streamlined frameworks. Because so many of the necessary tools for Java web development already exist as mature packages, Roo is a layer of convenience methods for generating code and connecting components. This is still incredibly helpful for speeding development, since Java and XML are verbose languages and the combination of Java frameworks can be dizzying. It's somewhat helpful pedagogically, since Roo provides a working application that draws from the dizzying collection of available Java tools. However, the Roo developer will eventually need to learn all the underlying packages and APIs in order to customize and extend the basic web application that Roo creates.

6.1 Spring Roo Design

In Spring Roo, the Model part of MVC can be used with multiple object-relational mapping APIs, including the Java Persistence API (JPA) or MongoDB. JPA seems to be the more commonly used than MongoDB or other alternatives. JPA is a common API for Java persistence, defined in the javax.persistence package. Roo provides built-in support for Hibernate, DataNucleus, EclipseLink, and Open JPA persistence layers.

Spring Roo is built on top of two primary features–AspectJ and the Spring Framework. Without understanding both of these technologies, it's difficult to really understand how a web application built with Spring Roo works. AspectJ holds scaffolding methods alongside Roo's simple Java model classes. AspectJ methods reside in .aj files alongside .java files, and the methods they define are inserted into the Java class at compile time. This allows the code generated by Spring Roo to remain separate from the main model source file. The Roo console automatically maintains these

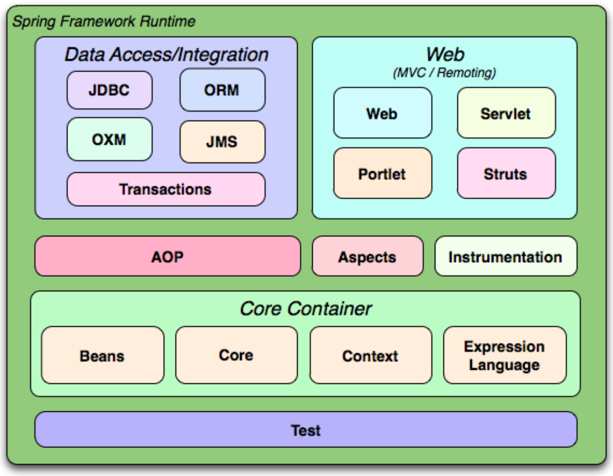

Figure 2: Spring Framework Overview

AspectJ files, adding additional methods when model fields are inserted, and removing AspectJ methods if they are implemented in the Java source.

Spring is a sprawling collection of Java productivity tools that originated around the concept of Inversion of Control (IOC), also known as dependency injection. Spring's IOC allows definition of class dependencies in XML, which dynamically set objects of the appropriate type on dependent objects–like a database adapter that could be mocked-out for tests and backed by a product database when deployed. However, Spring has grown to include libraries and components for assisting Java development in many areas, including tests, database integration, web frameworks, and aspect-oriented programming. Figure 2 shows an overview of the packages available in the Spring Frameworks. Spring Roo is a collection of coding automation tools that generate models, views, and controllers configured to work properly with the Spring Framework.

6.2 Spring Roo Routing

Routes in Spring Roo are defined as annotations in the Controller classes. The annotation that defines URL paths is `@RequestMapping` from the Spring Framework's MVC component. The details of the annotation are described well in the Spring Framework documentation [11]. Table 2 is the URL structure that Spring advertises, however it required some route customization to achieve.

To override the default behavior, one can pull Roo generated AspectJ methods into the Java controller classes and change the `@RequestMapping` annotations. An example of this from the sample

HTTP Method	Path	Action
GET	`posts/`	List members of the collection
POST	`posts/`	Create a new post
GET	`posts/:id/`	Show post with the given id
PUT	`posts/:id/`	Update the post with the given id
DELETE	`posts/:id/`	Delete the post with the given id
GET	`posts/form/`	Form for creating a new post
GET	`posts/:id/form`	Populated form for editing post with the given id

Table 2: Spring Roo Routing [12]

application is shown in Listing 16. This maps the path `/posts/new` to return the JSPX page at WEB-INF/views/posts/create.jspx.

```java
@RequestMapping("/posts")
@Controller
@RooWebScaffold(path = "posts", formBackingObject = Post.class)
public class PostController {

  @RequestMapping(method=RequestMethod.GET, value="/new", produces = "text/html")
    public String createForm(Model uiModel) {
        populateEditForm(uiModel, new Post());
        return "posts/create";
    }
```

Listing 16: Spring Roo Route Customization

6.3 Spring Roo Models

The commands in Listing 20 define a class named Post in the domain sub-package, then define three fields: name, title, and content for that class. The resulting Post.java file is shown in Listing 17.. These are enhanced by five AspectJ files adding implementations of toString, Java Bean, JPA Entity, ActiveRecord, and Configurable interfaces.

```java

@RooJavaBean
@RooToString
@RooJpaActiveRecord
public class Post {

    @NotNull
    @Size(min = 5)
    private String name;

    @NotNull
    @Size(min = 5)
    private String title;

    @NotNull
    @Size(min = 1)
    private String content;

    @ManyToMany(cascade = CascadeType.ALL)
    private Set<Comment> comments = new HashSet<Comment>();

    @ManyToMany(cascade = CascadeType.ALL)
    private Set<Tag> tags = new HashSet<Tag>();
}
```

Listing 17: Spring Roo Post Model

6.4 Spring Roo Controllers

Like other frameworks, Spring Roo uses one controller class per model class. Those controller classes interact with model classes to pass information to the JSPX views. The Roo convention is to place web controllers in a .web sub-package next to the models' .domain package.

A sample of controller code was shown in Listing 16. A snippet from the corresponding AspectJ file is shown in Listing 18. Note the similarity to Java syntax. When compiled, this adds the `show` method to the PostController class, which maps to the path `posts/:id`.

```java
privileged aspect PostController_Roo_Controller {
    ...
    @RequestMapping(value = "/{id}", produces = "text/html")
    public String PostController.show(@PathVariable("id") Long id, Model uiModel) {
        uiModel.addAttribute("post", Post.findPost(id));
        uiModel.addAttribute("itemId", id);
        return "posts/show";
    }
    ...
}
```

Listing 18: AspectJ for PostController

6.5 Spring Roo Views

Spring Roo stores its views in XML compliant Java Server Pages (JSPX). Views draw from the standard JSP tag library as well as tags from the Spring Framework. Roo creates these files when the `web mvc all --package ~.web` command is run. The result is a WEB-INF directory designed to run as a servlet, which uses JSP and Spring Web Framework tags to dynamically create HTML representing the model. An additional Roo-specific tag library was added in Spring Roo 1.1. It reduced the quantity of JSPX scaffolding by 90% [12].

Servlet definitions begin with the web.xml file in the WEB-INF directory. It defines the name of the web application, the Servlet class that handles the request (Roo uses the `DispatcherServlet` class from the Spring framework), and the configuration options for that Servlet (in webmvc-config.xml). Roo uses Spring Framework's TilesConfigurer bean to arrange snippets of JSPX code in a document.

The layouts.xml configuration file defines tile definitions, which map tile names to JSPX files. The defaults.jspx file contains application-wide HTML tags and defines where to pull in other JSPX files based on the supplied attributes. There is one views.xml configuration file for each model class. They are placed in views/<model name>/views.xml. The views.xml file defines which JSPX page to load as the body tile for each URL. The resulting JSPX page relies heavily on JSP and Spring tag libraries and is more difficult to follow than the other HTML templating systems. The JSPX file that shows the content, comments, and tags for a specific post is shown in Listing 19.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:field="urn:jsptagdir:/WEB-INF/tags/form/fields" xmlns:jsp="http://java.sun.com/
    JSP/Page" xmlns:page="urn:jsptagdir:/WEB-INF/tags/form" version="2.0">
    <jsp:directive.page contentType="text/html;charset=UTF-8"/>
    <jsp:output omit-xml-declaration="yes"/>
    <page:show id="ps_com_iancwill_blog_domain_Post" object="${post}" path="/posts" z="
        bMsjvaHSqIKQKUojDd7z06xtAmg=">
        <field:display field="name" id="s_com_iancwill_blog_domain_Post_name" object="${post
            }" z="6z1mK8Y72zz40bCT6Md8v6SkkAg="/>
        <field:display field="title" id="s_com_iancwill_blog_domain_Post_title" object="${
            post}" z="ve0cWdeA2U5EhfMDqKYfl85SJPY="/>
        <field:display field="content" id="s_com_iancwill_blog_domain_Post_content" object="
            ${post}" z="1JRkyhat0ZtnWXzytviYHfvoW7k="/>
    </page:show>
    <page:list id="pl_com_iancwill_blog_domain_Comment" items="${comments}" z="
        dvpn6uWMxvNkdmfhCcyXxcEMS10=">
        <table:table data="${comments}" id="l_com_iancwill_blog_domain_Comment" path="/
            comments" z="QQZac3PqPy/Q7XUnWoUchvGvLYY=">
            <table:column id="c_com_iancwill_blog_domain_Comment_commenter" property="
                commenter" z="05JqnEIg+z8CittVXKWn2kgff68="/>
            <table:column id="c_com_iancwill_blog_domain_Comment_body" property="body" z="Cp
                +6wb5Qb49BFYBUHC+CUop9Tb8="/>
        </table:table>
    </page:list>
    <page:list id="pl_com_iancwill_blog_domain_Tag" items="${tags}" z="u3uVJe9Yhe+
        v0NIcjSMy8qze7EQ=">
        <table:table data="${tags}" id="l_com_iancwill_blog_domain_Tag" path="/tags" z="
            rW2ScFP4xaYE7D1r+u59iBwc4f0=">
            <table:column id="c_com_iancwill_blog_domain_Tag_name" property="name" z="
                AIvLX70Lavrr/0IcaUqjgGHEiV8="/>
        </table:table>
    </page:list>
</div>
```

Listing 19: Spring Roo Post View (show.jspx)

6.6 Spring Roo Code Generation

The Roo experience centers on a Roo console, in which commands are issued to define the model, generate tests, and generate view and controller scaffolding corresponding to the model. The Roo commands aren't as succinct as the Rails generation commands, but they allow a similar level of code generation. While developing the sample application, these were the commands issued to generate the basic model.

```
~ roo>entity jpa --class ~.domain.Post --testAutomatically
~.domain.Post roo>field string --fieldName name --notNull --sizeMin 5
~.domain.Post roo>field string --fieldName title --notNull --sizeMin 5
~.domain.Post roo> field string --fieldName content --notNull --sizeMin 1
```

Listing 20: Spring Roo Model Code Generation

7 Conclusion

The summary of effort and size of the sample application implemented in each framework is listed in Table 7. Django is the most succinct, but offers the least support for generating HTML. It's the easiest to understand, which is aided by excellent documentation written by the original author. Rails is the second-most succinct, and offered a good balance between automation, simplicity, and brevity. Rails also has excellent documentation and a vibrant development community. CakePHP used significantly more lines of code that Rails or Django, which is partly due to the nature of PHP. Development workflow in CakePHP was relatively easy to follow, though Ruby and Python are more powerful and expressive than PHP. Spring Roo generated the most code and provided the most complex configuration. Its complexity is a direct consequence of intentionally avoiding runtime dependence and framework lock-in. Both of those are good goals, but they present significant barriers to a web-developer not already familiar with the Java web of Servlets, JSPs, AspectJ, and Spring.

Rails and Spring Roo both offer full support for REST conventions by allowing controller functions to be mapped to a combination of HTTP method and URL path. CakePHP and Django both require special case handling in the controller or view logic to handle HTTP methods differently. CakePHP presented more barriers when starting out–requiring manual database model definition and a running Apache web server configured for PHP.

Based on this analysis, Rails appears to be generally the easiest and most flexible framework for a new web developer. Django is appealing in its conceptual simplicity, especially if HTML is being supplied from outside the development team. CakePHP offers many of the benefits of Rails to PHP programmers, but doesn't present a compelling reason to select PHP over Ruby of Python. Spring Roo does what it can to lessen the steep learning curve for Java web development, but is limited by existing frameworks and infrastructure. However, it's a good place to start if a Java implementation is required.

Command	Rails	Django	Cake PHP	Spring Roo
Code files	40	10	25	44
Generated Lines of Code	639	231	1564	1908
Edited Lines of Code	43	182	70	22
Total Lines of Code	682	374	1634	1915
Lines generated / Lines Edited	15.8	1.27	22.3	86.7
Config or HTML files	13	8	34	56
Total Lines of config/HTML	154	119	1219	2585
Lines of config/HTML edited	102	133	22	18
Lines of config/HTML generated	103	0	1197	2606

Table 3: Comparison of Size and Effort for the Sample Application

References

[1] B. Buchner, A. Bottcher, and C. Storch, "Evaluation of Java-based open source web frameworks with Ajax support," *Web Systems Evolution (WSE), 2012 14th IEEE International Symposium on*, pp. 45–49, 2012.

[2] T. C. Shan and W. W. Hua, "Taxonomy of Java Web Application Frameworks," in *e-Business Engineering, 2006. ICEBE '06. IEEE International Conference on*, 2006, pp. 378–385.

[3] L. F. F. Stella, S Jarzabek, and B. W. S. E. . W. . t. I. S. o. Wadhwa, "A comparative study of maintainability of web applications on J2EE, .NET and Ruby on Rails," *Web Site Evolution, 2008. WSE 2008. 10th International Symposium on*, 2008.

[4] M. Raible, "Comparing Web Frameworks," pp. 1–50, May 2007.

[5] H. Li, "RESTful Web service frameworks in Java," in *Signal Processing, Communications and Computing (ICSPCC), 2011 IEEE International Conference on*, 2011, pp. 1–4.

[6] (Nov. 2012). Ruby on rails, [Online]. Available: `http://en.wikipedia.org/wiki/Ruby_on_Rails`.

[7] R. T. Fielding and R. N. Taylor, "Principled design of the modern Web architecture," *Transactions on Internet Technology (TOIT*, vol. 2, no. 2, May 2002.

[8] (Nov. 2012). Rails routing from the outside in, [Online]. Available: `http://guides.rubyonrails.org/routing.html`.

[9] (Nov. 2012). Ruby on rails guides: layout and rendering in rails, [Online]. Available: `http://guides.rubyonrails.org/layouts_and_rendering.html`.

[10] (2009). Django book, [Online]. Available: `http://www.djangobook.com`.

[11] (Nov. 2012). Web mvc framework, [Online]. Available: `http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/mvc.html`.

[12] (Nov. 2012). Beginning with roo: the tutorial, [Online]. Available: `http://static.springsource.org/spring-roo/reference/html/beginning.html`.