Functional Programming & Web Frameworks

Independent Study Report

Justin Cady

February 22, 2011

This report summarizes an independent study on web frameworks by Justin Cady, advised by Professor Matthew Fluet. The study examined several frameworks, evaluating each and examining it with regards to any functional programming either present or possible. Professor Fluet and I met weekly to review the topics and to outline the deliverables of each week.

1 Django

Django is a web framework written in Python. It is an open-source project maintained by the Django software group. For this study, Python 2.6.1 and Django 1.2.3 were the versions used.

1.1 Framework Philosophy

Django applications are built using the model-template-view pattern (traditionally model-view-controller), with the emphasis on starting with well-defined models. The data design is what drives Django, exemplified by the fact that changing the model definitions midway through development is not possible without external tools. The model system is a simple object-oriented approach to database schema. It eliminates the complexities of writing SQL, and makes transitioning to different databases simple because of this abstraction layer. Complex database designs relating objects through join tables are easily definable in pure Python code. This example model definition shows standard fields and a relationship definition:

```
class Note(models.Model):
    text = models.TextField()
    title = models.CharField(max_length=100,unique=True)
    contributors = models.ManyToManyField(User, through='Contribution')
```

The view layer is where the logic of the application resides. Since the code is also pure Python, any Python libraries or constructs can be used. Each view typically concludes by rendering a given dictionary (hash) with a template in an HTTP response.

Django uses a domain specific template language for the templating layer. It became clear after research that this template language was run as Python. At a page's request time, a dictionary is supplied to a template. The Python objects in that dictionary are used within the template's executable code sections. The templating language is one of Django's best features. It uses minimal syntax, cleanly integrates with HTML, CSS, and Javascript, and has a useful inheritance system. The ease of use was impressive for such a powerful piece of the framework.

This example shows a potential base template that would be extended by other templates:

```
{% block header %}
<!doctype html>
```

```
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>Jot - Django</title>
  <meta name="description" content="An implementation of Jot in Django" />
  <meta name="author" content="Justin Cady" />
  <link rel="stylesheet" href="/static/style.css" type="text/css" />
  {% block scripts %}
  {% endblock scripts %}
</head>
<body>
{% endblock header %}

{% block content %}
{% endblock content %}

{% block footer %}
</body>
</html>
{% endblock footer %}
```

A child template extend this class, overwriting blocks as it needs. It also can iterate over objects given to the template, as well as some other basic data checking:

```
{% extends "base.html" %}

{% block content %}
<div id="profile_container" class="group">
    <h2>{{ user.username }}</h2>

    <div id="photo_container" class="group">
        <div id="bio_picture" style="background:url({{ profile.photo.url }});"></div>
        <div id="bio"><p>{{ profile.bio }}</p></div>
    </div>

    <div id="photo album" class="group">
        {% for photo in photos %}
            <img src="{{ photo.url }}" alt="{{ photo.description }}" />
        {% endfor %}
    </div>

</div>
{% endblock content %}
```

Django's URL routing is done via regular expressions that are explicitly connected to views. This is very simple to grasp, but perhaps could be optionally automated in the future. That would allow developers to avoid parameter matching mistakes and spelling errors, both of which are easy to do in its current state.

1.2 Project

The project built using Django was a note sharing application named Jot. Jot employed many of the features possible with Django, including user authentication, an administrative database backend, and form generation and validation. Users could log in and create notes they stored on a virtual desktop. They could share these notes with other Jot users, who would see the notes in their inbox upon their next login. Users also could manage their own profile which allowed them to upload a photo, and save a short biography.

1.3 Functional Programming Demonstrated

Since the server code is written in pure Python, any of Python's functional constructs (list comprehensions, filter, map, lambdas...) could be used. Querying the database uses method chaining to continually filter the dataset. Effectively this practice is repeated function application. Also, I found that list comprehensions became commonly used in retrieving a list of data from the database. Example:

```
def get_mutual_friends(user1, user2):
    one_friends = get_friends(user1)
    two_friends = get_friends(user2)
    return [friend for friend in one_friends if friend in two_friends]
```

1.4 Strengths & Weaknesses

The official documentation is superb. The template system packs a lot of power and is a pleasure to use, due to its simplicity. I also enjoyed the necessity of explicit connections, rather than the alternative of the framework automatically connecting the pieces of the application stack. Conversely, the inability to modify models without low-level database hacking is a major downside.

2 Ruby on Rails

Ruby on Rails is arguably the most popular web framework today. It is developed and maintained by 37Signals. For this study, Rails 3.0.3 and Ruby 1.9.2 were the versions used.

2.1 Framework Philosophy

The Ruby on Rails ideology is convention over configuration. This is clearly demonstrated when developing Rails applications, because so much code is automatically generated based on "convention." The idea of sensible defaults may work well once a developer is familiar with the framework, but as a new user I found it confusing. Django seems to expect the developer to be more explicit, whereas Rails expects developers to rely more on magic. However, I do feel that continued exposure to the framework would ease some of these concerns.

Interacting with the models in Rails is done using an object-oriented approach that is very similar to Django. There seem to be more convenience model methods available in Rails, such as ".first" or ".last". By using the "rails generate" command, models are automatically created in the application code. The "generate" command also creates routes, controllers, and views based on these models for basic CRUD capability. Ruby on Rails uses a model definition system in pure Ruby. It separates the database schema from the individual object models and their validations. This is different from Django, where all of the model information is in a single file. An example of a Rails model schema:

```
# ...
create_table "notes", :force => true do |t|
  t.string   "title"
  t.text     "text"
  t.datetime "created_at"
  t.datetime "updated_at"
end
# ...
```

And the corresponding model file:

```ruby
class Note < ActiveRecord::Base
  has_many :contributions
  has_many :users, :through => :contributions
end
```

Controllers use the convention over configuration policy as well, providing a global parameters array, and connecting controllers to views based on filename alone. Controllers have the powerful ability to define different response types that render a given model. The same controller can be defined to respond to *.html*, *.xml*, and *.json* requests, for example:

```ruby
def get_desk
    user = User.where(:username => session[:username]).first
    @notes =
    Note.find(:all,
              :joins => :contributions,
              :conditions => {:contributions => {:on_desk => true, :user_id => user.id}})
    respond_to do |format|
        format.html  { render :json => @notes }
        format.xml { render :xml => @notes }
    end
end
```

Variables in the controller prefixed with "@" are provided to their corresponding template. Rails' template language allows pure Ruby (unlike the Django template language) inside of the view files. Files containing embedded Ruby code are suffixed with the .erb extension. Example:

```
<div id="profile_container" class="group">
    <h2><%= @username %></h2>

    <div id="photo_container" class="group">
        <div id="bio_picture" style="background:url(<%= @username %>.jpg);"></div>
        <div id="bio"><p><%= @user.bio %></p></div>
    </div>
    <br/>
</div>
```

Unsurprisingly, connecting views to controllers is done through automatically generated routes (though they can be overridden).

2.2 Project

The Jot project was again used in Ruby on Rails, so that the two similar frameworks could be compared based on the same requirements.

2.3 Functional Programming Demonstrated

Similar to Django, Rails allows any Ruby construct within controllers (Rails also allows it in views). This means that any functional piece of Ruby can be used in these areas. Ruby has lambda expressions and blocks, among other functional tools like "map."

2.4 Strengths & Weaknesses

I found the official documentation to be overwhelming and had trouble finding a quality resource that let me learn Rails quickly. Even after going through tutorials and documentation, Rails seems intimidatingly large. Even using a simple extension (user authentication with Clearance) took days of research that was ultimately unfruitful. A simple application generates dozens upon dozens of

files, and a boilerplate kit available for download required well over 30 external dependencies. Rails certainly has enough features for developers, but it seems to try to do too much.

3 Flapjax

Flapjax is an implementation of functional reactive programming in Javascript. Version 2.1 was used for this project.

3.1 Framework Philosophy

Flapjax is built around the idea of behaviors, continuous values changing over time. The parallel to behaviors are the discrete value streams known as event streams. In addition to the premier behavior features, Flapjax also provides some useful helper functions such as map, fold, and filter. Flapjax has other functional constructs such as lifting, which is the ability to lift functions over behaviors. Example:

```
function loader() {
  //store a timer behavior in a variable
  var timeB = timerB(100);
  //lift this anonymous function over the timer behavior
  var secondsB = liftB(function (v) { return Math.floor(v / 1000); }, timeB);
  //update the page to reflect this behavior's changing value
  insertDomB(secondsB, 'timer');
}
//Example courtesy the Flapjax documentation
```

Flapjax provides some valuable insight into how functional programming can be used in creative ways on the web. The complex concept of functional reactive programming does not at first thought seem like a natural fit with Javascript, but Flapjax provides many real-world examples that prove otherwise. Their main philosophy centers around rethinking the use of callbacks, a common idiom in Javascript web programming, using FRP.

I found it interesting that Flapjax is essentially a complete paradigm shift from traditional Javascript programming. While a large majority of web applications that utilize Javascript are designed around callbacks, Flapjax presents a completely new take in the area. I think this shows that creating new ideas is still beneficial, and can open ways of thinking about a problem previously unused within a given domain.

3.2 Project

The project was a sudoku puzzle webpage that updates the user's progress in real time. Behaviors are applied to each cell in the sudoku puzzle, notifying the user if a cell's value is invalid within its row, column, or group. Additionally, the rows, columns, and groups status are logically anded together, which results in the user being notified of a win when all three collections are filled and valid. This project demonstrated a fresh take (FRP) on the classic problem of sudoku solving.

3.3 Functional Programming Demonstrated

Flapjax is functional at its very core. The concepts of behaviors using functional reactive programming, lifting functions, and mapping event streams are all functional. Javascript also uses first class functions, making the sudoku implementation partially functional as well.

3.4 Strengths & Weaknesses

The documentation is well-organized, and contains plenty of examples. Flapjax is a great demonstration of truly functional concepts applied to the web development space. It has the flexibility of either being compiled with HTML, or included as a library on a webpage. However, its scope is small enough that it likely would not be able to be used in an industry setting. Paired with other technologies though, it performs responsively enough to be used in real-world environments.

4 Snap

Snap is a web framework written in Haskell. Version 0.3.0 was used for this study.

4.1 Framework Philosophy

Snap offers a different perspective on web programming, as it is based on monadic programming. Developers are given monads to control the application state, and supply the templates with data. Snap also aims to benefit developers in performance. Even in its early stages, the Snap server has excelled speed-wise.

Of course, some of this may be due to the fact that Snap currently has no concept of data modeling, and therefore no database latency to worry about. These features are planned and will come with time, but Snap is still very early-stage software.

Snap is being developed in conjunction with the Heist templating language, a separate project that is integrated into Snap. Heist presents some templating features that are both powerful and functional in nature. It allows data substitution via XML tags that can be bound to values within the controller code.

```
<bind tag="longname">
    Einstein, Feynman, Heisenberg, and Newton Research Corporation
    Ltd.<sup>TM</sup>
  </bind>
<p>
    We at <longname/> have research expertise in many areas of physics.
    Employment at <longname/> carries significant prestige. The rigorous
    hiring process developed by <longname/> is leading the industry.
</p>
<!-- Example courtesy the Heist documentation -->
```

The system also provides splicing: the ability to bind XML tags to functions written in Haskell. One could create a factorial function, and then bind it to the tag "¡factorial¿" so that anywhere in the template, the value inside the tag will be run by the function:

```
...
<factorial>5</factorial> <!-- renders as 120 -->
...
```

Unfortunately, without database information readily available to be processed with this advanced feature, it is hard to get a grasp on how powerful this feature truly is. Regardless, it does provide an interesting functional take on templating languages.

4.2 Project

The Snap project involved a simple user profiling system in which users (stored in memory) each had their own profile page. The page was dynamically generated using splicing and template binding

with information about each user. Due to some development difficulty the project was not fully realized, but its basic concepts still provided plenty of information about the framework and its capability.

4.3 Functional Programming Demonstrated

Being completely written in Haskell, Snap is functional from the ground up. Its use of monads is a very functional take on web application development, and the splicing in templates would be hard to imagine outside a functional context.

4.4 Strengths & Weaknesses

Snap is very well-documented, but its problem is that there is not too much to document at this point. Many features that would be givens in other frameworks are not present as of this writing, but will surely come in the future. The Heist templating language is what I consider the most innovative and powerful feature of Snap to date. The performance metrics are also impressive, but it will be interesting to see what numbers the framework can produce when it becomes more fully-featured.

5 Conclusion

This independent study was an extremely informative research project. The time spent carefully examining the aforementioned frameworks produced a wide range of perspectives on web application development. As expected, there is a striking difference between traditional or popular web programming and some of the new ideas to come from the functional programming community.

The surprising part is that there is also a lot in common. All types of frameworks are reaching for the same goals: rapid development of web applications that cover common needs. These common needs include user accounts and authentication, security measures, database interactions, URL routing, and a separation of concerns for the various parts of the application, among other features.

There are many functional ideas that translate well to the web. In both Ruby on Rails and Django, the filter-chaining on the database was functional in spirit, simply putting a more common syntax over the relational algebra. Both frameworks also benefitted from list comprehensions in gathering data to be presented back to the user.

In Flapjax, functional reactive programming was used to define behaviors and essentially abstract away the controller glue code to create dynamic applications. Flapjax also took advantage of Javascript's first class functions by allowing developers to lift them over behaviors. The Snap framework used monads to process application state, and allowed the binding of tags to functions for even more dynamic templating.

It is clear that functional programming can be used for web application development, but more importantly, can be used to make web application development *easier* on developers. The best example of this during this study was Flapjax, which redefined the callback paradigm and eliminated an entire layer of code while providing the same functionality as typical Javascript code would have allowed. The other frameworks demonstrated their strengths, but showed some spots where functional concepts could improve the total package. The challenge for the functional community will be to continue to innovate in this space, while keeping things accessible to current web developers.

A Course Hours

The study was agreed to take 8 to 12 hours per week before it began. On average I spent about 8 hours on research weeks, and closer to 12 or more hours on project weeks. Averaging these two, the course took approximately 10 hours per week.