Functional Programming Applied to Web Development Templates

MS Project Report

Justin Cady

Rochester Institute of Technology

Chair: Professor Matthew Fluet
Reader: Professor James Heliotis
Observer: Professor Dan Bogaard

May 31, 2011

1 Project Description

In most web applications, the *model-view-controller (MVC)* design pattern is used to separate concerns of a system. However, the *view* layer, also referred to as the *template* layer, can often become tightly coupled to the controller. This lack of separation makes templates less flexible and reusable, and puts more responsibility in the controller's hands than what is ideal.

In fact, templates are often completely powerless, giving developers no option but to tailor them more and more directly to their respective controllers. A better solution is to give the template layer more control over the data it presents and limit the controller's influence on presentation. This document presents the *jtc* template engine, which uses elements of functional programming to empower templates by treating templates as functions. The background, process, and results regarding the development of *jtc* are discussed below.

2 Background

2.1 Web Development

As the web application field continues to grow, more developers are creating products delivered through the browser. But, web development is a discipline that involves a multitude of different technologies. A typical application involves:

- A content server, such as IIS, Apache or Nginx.
- A database, such as MySQL, PostgreSQL or Microsoft SQL Server.
- A server-side language, commonly Java, ASP.NET, Perl, PHP, Python, or Ruby.
- A client-side language, almost exclusively JavaScript.
- A client-server communication layer, typically AJAX (Asynchronous JavaScript and XML).
- A markup language, such as HTML, XHTML, or XML.
- A markup styling document written in CSS.

Among these pieces of an application, development is divided across upwards of five programming languages. There have been frameworks that attempted to unify this process into a single language, such as the Links framework [1]. But, these projects have not seen mainstream adoption, in part due to their intimidating learning curve.

2.2 Web Frameworks

Web frameworks are toolsets that expedite the development process of web applications across all of the aforementioned pieces of a web application. Common tasks such as user authentication, data model definition, database object-relational mapping, and HTML templating are built into most web frameworks. By using frameworks, developers can focus on building their unique functionality rather than once again coding the same tasks common across a majority of web applications.

Frameworks can also provide stability. Any problems or major bugs with a framework are likely to be discovered by the community before individual developers would run into them. The communities built around popular web frameworks can be an essential reference as well, lending advice to newcomers and providing a large knowledge base for advanced support.

2.2.1 Example Web Frameworks

Ruby on Rails Ruby on Rails (RoR) is a web framework written in Ruby [2]. The RoR ideology is "convention over configuration." This is evident as much of the code for RoR applications is automatically generated. As mentioned, a large portion of web application structure is often repeatable, therefore providing some boilerplate to developers is helpful. Once developers fully learn and understand the conventions, projects can be up and running extremely quickly. This philosophy is partly what has made RoR so heavily used.

Django Django is a web framework written in Python [3]. The philosophy of Django is to start from the bottom by defining the data model, then build the rest of the application on top of that foundation. Whereas Ruby on Rails favors automatically generating much of the code, Django either provides functionality with includable modules, or asks the developer to connect the application layers manually.

Snap Snap [4] is a recently created Haskell web framework that applies heavy FP thought to web development. Snap uses monads to control the state of the application on the server and deliver that data to the user through templates. In its current state, Snap is extremely limited. It does not support database connectivity or model definitions. But, it is an example of a purely functional perspective on web applications.

CodeIgniter There are quite literally hundreds of web frameworks written in PHP. The language was designed for the web, so this number is expected. One of the prominent PHP web frameworks is CodeIgniter [5]. CodeIgniter, like the other frameworks mentioned, uses the MVC design pattern. It is designed for performance, even on servers with limited resources.

The main benefit that CodeIgniter and frameworks of its kind hold over competitors is ease of deployment. Because of PHP's long history with the web, nearly every server available supports some form of PHP. Because of that fact, PHP-based frameworks typically require minimal configuration to run on a live server. Other languages, such as Ruby or Python, require much more meticulous deployment strategies for them to run in a server environment.

2.3 Functional Programming

Functional programming (FP) is a paradigm based on the lambda calculus in which programs compute results through the application of functions rather than the state of the system. Functional programming languages lack mutable data structures and common iteration structures such as `for` or `while`—iteration is done through recursion. This structure provides many features that are typically unavailable outside of FP, though some elements of FP have made their way into imperative languages.

2.3.1 Selected FP Concepts

Functional programming is an enormous discipline that cannot be fully covered within this document. But some selected concepts are now explained as they are leveraged by the *jtc* framework.

First-class functions
Functions themselves can be stored in variables, passed as arguments to other functions, and returned as results from functions.

Currying

Using first-class functions, an n-parameter function can be broken down into a series of *1*-parameter functions.

Closures

Rather than returning a function as simply a function pointer to a subroutine, a function is *closed over* its free variables, retaining access to its original scope for its lifetime.

2.3.2 Functional Programming and Web Development

For most developers, FP is more difficult to learn than imperative programming. However, it gives developers the benefits of terse code, precise expressiveness, and it adapts well to concurrent programming. FP is gaining popularity in some web applications, but is still relatively uncommon in the area.

However, FP is an ideal paradigm for web development. HTTP is a stateless protocol; state is instead simulated through the use of cookies or HTML5 *localStorage* on the client and sessions on the server. As a user interacts with a web application, each action on the user's part could be thought of as executing a function from HTTP request to HTTP response. And with an application developed using FP, the code could exactly match that natural thought process.

2.3.3 Related Functional Programming Frameworks

Though FP frameworks are not as commonly used in web development, many do exist in addition to the aforementioned Snap framework. Links [1], Ocsigen [6], and Lift [7] are examples of existing frameworks, and even PLT Scheme [8] has been used to create a complete web application. These frameworks contain various functional ideas that are highly beneficial to web frameworks:

Static Typing A problem exists within web development concerning data validity. Data transfer between the client and the server is done through HTTP requests [9], typically *GET* or *POST* requests. This data is transferred in raw string form, which makes the process of validating data types more cumbersome. The problem is exacerbated by the fact that most web server environments are built with dynamically typed languages. Couple all of that with another set of data types defined in the database schema, and the difficulty exists across all three layers of the application.

There are a variety of solutions to this problem. Data can be transferred using either JSON or XML formats to more clearly describe what is being passed to and from the server. And as always in web development, server-side validation is absolutely essential to provide feedback about mistaken input to the user and protect against attacks.

Some functional web frameworks use their language's static type system to accomplish typing of web applications. Ocsigen [6] uses the type system of its language, OCaml, as a method to verify the types of HTML form submissions and HTML output validity. The authors of Ocsigen cite static typing as a major benefit of their system because HTML forms can be statically checked against the server-side code accepting them.

The Links [1] framework is built upon the Links language that was developed for the project. Since each layer of the application is written in the same language and compiled into the necessary HTML, JavaScript, and SQL, a static check similar to Ocsigen can be performed. More approaches akin to this design have been used by Lift [7] and SMLServer [10].

Continuation-based Sessions Another FP element that has demonstrated usefulness in web development is the continuation. Balat states the following two ways to think about functional web development [6]:

- "... viewing the sending of a Web page by the server as a function call, (a question asked of the user) that will return for example the values of a form or a link pressed."
- "... viewing the click on a link or form by the user as a remote function call. Each link or form of the page corresponds to a continuation, and the user chooses the continuation he wants by clicking on the page."

With the second opinion in mind, the Ocisgen framework uses continuations on the server for user interaction. Balat gives the following example which illustrates how continuations are useful:

> As an example, consider a web page that asks for a number from the user. The number is sent to the server, and then a second page is generated, proposing to enter a second number to be added to the first one. Then a third page is displayed, showing the result of the calculus. A first way of implementing this site consists in putting the first number in the second page, as a hidden form field or in the URL of the third page. Thus, both numbers will be transmitted to the third page. But what if the result depends on much more complicated data, that you cant put easily in a hidden field or in the URL? With continuations, the solution is very clear. When receiving the first parameter, the server creates dynamically a continuation corresponding to adding this number. Sending the second number will activate this continuation.

Both the PLT Scheme web framework [8] and the Links framework [1] also leverage continuations, albeit slightly differently, for their user interactions.

2.4 Problems Addressed by this Project

Both the fields of web development and functional programming are vast, and the problems of web development frameworks are too numerous to address in one single project. The scope of the *jtc* project is building a template engine that aims to solve the following problems using FP concepts:

2.4.1 Multitude of Technologies

As discussed in Section 2.1, the development of web applications involves a large amount of technologies and different programming languages. A goal of the *jtc* project is keeping as much of the framework as possible in the same language—JavaScript. The reason for the selection of JavaScript is discussed in Section 4.2.1.

2.4.2 Simple, Flat Data Presentation

Many web frameworks are limited when transferring data between the layers of the application. If the controller of an application is only able to pass a hash table of string values, it inevitably becomes tightly coupled to the view. In this case, the controller must know explicitly what data the view needs, and both sides need to have knowledge of the key for the data in question.

A goal of *jtc* is reducing the coupling between the controller and the view as much as possible. This is done by placing some of the controller's common responsibilities on the view, and by passing rich data structures between the layers instead of a table of strings.

2.4.3 Simplistic Templates

Templates that simply spit out tags wrapped around specified data from the controller are extremely limited. With a system that gives the view no power, the design and presentation of the website becomes reliant on the controller. A better separation sets up the controller as the provider of a general dataset, with the view then free to choose which specific pieces of the dataset it needs and how to display them. The *jtc* template engine was built with this philosophy in mind.

3 Related Work in Template Engines

A typical interaction with a web application flows as follows:

- A user clicks on a link or submits a form, causing the browser to send an HTTP request to the server
- The server receives the request, and performs some server-side computations based on the data received in the request
- The server sends an HTTP response back to the user, commonly containing a body of HTML, XML, or JSON

The template engine, or *view* layer, is the piece of the application stack that assembles the HTML result to be delivered via the HTTP response to the user. The template engine is executed server-side. It gathers data that was sent from the user (request URL, cookie, form entries) along with any necessary data from a database, and uses that data in forming some response in memory. The response is returned to the user in one of the formats mentioned above, most commonly HTML. The browser receives the HTML response and processes its contents, rendering the viewable webpage to the user.

Many web frameworks include template engines that each present a slightly different take in designing the HTML result. Some splice HTML with source code, while others are purely code or purely HTML. A select few are briefly demonstrated below as examples of what has already been done in this space.

3.1 Ruby on Rails

Following the "convention over configuration" philosophy of RoR, the template file to be used for a certain URL is selected by naming schemes. Filenames must match the names of the controller set to render them back to the user (this default behavior can be overridden) [11].

RoR templates themselves are `.erb` files, which stands for "**e**mbedded **rub**y." As the name suggests, template files allow the insertion of pure Ruby code inside HTML, for example:

```
<h1>Ruby Template</h1>
<!-- A Ruby statement -->
<% number = 42 %>
<!-- An expression to be inserted into the HTML (note the equals sign) -->
<p>The number is <%= number %></p>
```

The `.erb` files live on the server within the RoR directory structure. When the user requests a URL, RoR loads the `.erb` file that corresponds to that URL. It then processes the embedded

Ruby within that file, and returns the HTML result to the user. As mentioned, formats other than HTML can be returned, such as JSON or XML. For all formats, the result is sent to the user within an HTTP response.

To aid in the writing of templates, RoR provides helper functions to supply stylesheet and JavaScript includes [11]. For example:

```
<html>
<head>
  <title>Page Title</title>
  <!--
    The line below loads the necessary <link> tags to
    references the stylesheets provided.
  -->
  <%= stylesheet_link_tag 'layout', 'typography' %>
  <!--
    The line below loads all of the JavaScript files in
    public/javascripts/ in the project directory.
  -->
  <%= javascript_include_tag :all %>
</head>
```

Templates in RoR can use the `yield` statement to allow a template to identify where content should be inserted into the HTML. For example, consider the following two files [11]:

Yielding Template

```
<html>
  <head>
  <%= yield :head %>
  </head>
  <body>
  <%= yield %>
  </body>
</html>
```

Template Used by Yield

```
<% content_for :head do %>
  <title>A simple page</title>
<% end %>

<p>Hello, Rails!</p>
```

Through convention of RoR file structure in the project, the first file could be set to `yield` to the second file for its content where specified by the `yield` statements. The resulting output of the above example [11]:

```
<html>
  <head>
  <title>A simple page</title>
  </head>
  <body>
  <p>Hello, Rails!</p>
  </body>
</html>
```

Data is provided to the templates by prepending an `@` to the beginning of variables inside controllers [12]. Doing this makes those instance variables available inside the corresponding templates. This behavior is one example of Rails performing what it calls "magic" for developers.

3.2 Django

The Django framework [3] refers to its separation of concerns differently than MVC, though the design is nearly identical. Instead of the traditional terminology, Django applications are built using a *model-template-view* design. This translates directly to MVC, with *templates* being the traditional *views* and *views* being the traditional *controllers*. The remainder of this section will use the *model-template-view* terminology for clarity.

3.2.1 Basic Structure

Specifying which template to use for a given URL in Django is done within the URL configuration file. Each URL is explicitly assigned to a certain view. The view receives data from the URL router, performs any necessary computations, then renders a template with any data the template needs to an HTTP response. An example URL configuration:

```
# A URL route table with a single entry
# This maps a regular expression to a Django view
urlpatterns = patterns('',
    ('/factory/widgets/','factory.assembly_line.get_widgets')
)
```

And the view that it corresponds to, which renders a template by passing it a Python dictionary of the data to be displayed in the template:

```
def get_widgets(request):
  if request.user.is_authenticated():
    widgets = Widget.objects.all()
    render_to_response('widgets_template.html',{
        my_widgets : widgets
    })
  else:
    return HttpResponseRedirect('/index/')
```

Lastly, the corresponding template:

```
<html>
  <head>
  <title>My Widgets</title>
  </head>
  <body>
    <dl>
    {% for widget in my_widgets %}
      <div class="widget">
        <dt>{{ widget.name }}</dt>
        <dd>{{ widget.function }}</dd>
      </div>
    {% endfor %}
    </dl>
  </body>
</html>
```

The simple template example above introduces the Django template language, which is a separate entity from Python. Statements such as iteration structures or conditionals are contained within `{% %}` blocks. Expressions to be directly substituted into the HTML are contained within `{{ }}` blocks.

3.2.2 Template Inheritance

Templates have the ability to extend other templates through inheritance. A parent template creates blocks that can be used or overridden by its template children. For example, the following could be a parent template:

```
<html>
  <head>
  <title>A simple page</title>
  </head>
  <body>
    {% block body %}
      <h1>Default content</h1>
    {% endblock body %}
  </body>
</html>
```

And a child template could implement it as so:

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

{% block body %}
  <h2>New content</h2>
{% endblock body %}
```

The `body` block defined in the parent can be optionally overridden in the child template. All the other HTML content from the parent template would remain exactly as originally written. Only the `body` block would be altered for the output of the child template.

3.3 Heist

Heist is the template engine bundled with Snap (Section 2.2.1), created by the same developers of the Snap framework [4]. Heist provides a functional take on template languages. XML tags in a template file can be bound to data values or bound to Haskell functions on the server (this process is called splicing). For example [4]:

```
<bind tag="longname">
  Einstein, Feynman, Heisenberg, and Newton
  Research Corporation Ltd.<sup>TM</sup>
</bind>
<!-- The following will render with the above text substituted into it -->
<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>
```

Above, the `longname` tag is bound to the string contained within it. Within a template, there could also exist a tag such as the following:

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

And the following Haskell function would exist on the server:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

The `factorial` tag is bound to the factorial function on the server, and its content is treated as input to that function. The function result will be substituted in place of the template's `factorial` tag.

In addition to substitution and splicing, Heist templates can be applied within each other across files. For example, imagine the following code for a navigation menu within the file *nav.tpl* [4]:

```
<ul>
  <li><a href="/">Home</a></li>
  <li><a href="/faq">FAQ</a></li>
  <li><a href="/contact">Contact</a></li>
</ul>
```

This code is going to repeated on many pages across the site. Within one of the templates for a page, the navigation menu above can be included as follows [4]:

```
<html>
  <head>
    <title>Home Page</title>
  </head>
  <body>
    <h1>Home Page</h1>
    <!-- The navigation menu is inserted here -->
    <apply template="nav"/>
    <p>Welcome to our home page</p>
  </body>
</html>
```

The difference between the simple substitution is that in this case, the common HTML is stored in an external file, and referenced by all files that need to include it. This eliminates the need to duplicate code across a number of template files.

3.4 Ocsigen

The Ocsigen framework provides two different approaches to generating XHTML documents. The first is an implementation using a set of OCaml functions corresponding to XHTML tags. An example as presented by Balat [6]:

```
(html
  (head (title (pcdata "")) [])
  (body [h1 [pcdata "Hello"]]))
```

An advantage of this approach over others presented in this document is that the nesting of functions can ensure document validity. Ocsigen does include another option for templates which looks more like the examples given for other template engines. This example, also as presented by Balat [6]:

```
(fun s ->
  << <html>
       <head><title></title></head>
       <body><h1>$str:s$</h1></body>
       </html> >>)
```

Much like the other template syntaxes, this alternate Ocsigen functionality intermixes its language code (OCaml in this case) with familiar HTML code. Of course, the benefits of the previous approach are lost here, therefore Balat recommends the first method unless converting previously used templates into Ocsigen templates is a concern [6].

3.4.1 Quasiquote

This approach is reminiscent of the Scheme construct, `quasiquote` [13]. `Quasiquote` allows the mixing of literal statements with statements that need to be evaluated, just as the above Ocisgen syntax allows the splicing of OCaml code with HTML tags. An example of `quasiquote`:

```
; Standard syntax
(quasiquote (apple (unquote banana) orange))

; Equivalent compressed syntax
`(apple ,banana orange)
```

Both the above statements are equivalent. In each case, the result will be a list containing the literal expression `apple`, followed by the *value* of the expression `banana`, followed by the literal expression `orange`.

3.5 Links

Because Links is a web programming language that merges the layers of *MVC* into one piece, it behaves quite differently than the other examples presented. The functionality of the application is intermixed with the template code. Functions can contain program logic, HTML, or both. The following is an example of a function containing interspersed HTML [1]:

```
fun format(defs) client {
  <#>
    <h3>Click a definition to edit it</h3>
    for (var def <- defs)
      <span class="def" id="def:def.id">(formatDef(def))</span>
  </#>
}
```

The above function takes a list of definitions as its argument, and outputs an HTML fragment with the `h3` tag and a collection of `span` tags. (It is of note that the code given in this example executes on the client side.) This is just one example of the power of the Links language. Being that Links is a language and does not explicitly hold a separate template engine, it is difficult to draw further parallels between Links and other framework template layers.

3.6 Mustache

The Mustache template engine is a simplified take on generating HTML from templates [14]. Being a template engine only, Mustache is closely related to *jtc* in scope. Mustache is available in a variety of languages; the JavaScript variant is the one being specifically referenced here. The author refers to Mustache templates as "logic-less" templates because the system does not provide any `for`, `while`, or `if` statements. Instead, Mustache templates provide looping and conditionals in different ways. An example Mustache template [15]:

```
Hello {{name}}
You have just won ${{value}}!
{{#in_ca}}
Well, ${{taxed_value}}, after taxes.
{{/in_ca}}
```

To render their results, Mustache templates are given a hash of values. Continuing the example, a hash given to the above template [15]:

```
{
  "name": "Justin",
  "value": 10000,
  "taxed_value": 10000 - (10000 * 0.4),
  "in_ca": true
}
```

The above hash is a JavaScript object with key-value pairs. When the Mustache engine renders the template with the above data, the following is the resulting output [15]:

```
Hello Justin
You have just won $10000!
Well, $6000.0, after taxes.
```

3.6.1 Sections

As previously mentioned, Mustache is capable of provided the desired results of loops and conditionals using a different syntax. *Sections* are a Mustache construct that render a block repeatedly for each element in a list structure. Consider the following template which contains a section, denoted by the # [15]:

```
{{#repo}}
  <b>{{name}}</b>
{{/repo}}
```

A data hash corresponding to the template [15]:

```
{
  "repo": [
    { "name": "Michelle" },
    { "name": "Monica" },
    { "name": "Melanie" },
  ]
}
```

And the resulting output [15]:

```
<b>Michelle</b>
<b>Monica</b>
<b>Melanie</b>
```

At its heart, this feature is the FP `concatMap` function, taking a list of values and returning an accumulated string of the values with the template applied to them. Additionally, since functions are first-class in JavaScript, they can be passed to templates in Mustache. Functions can then receive the text of the template block, as shown in the following example [15]:

```
// Section
{{#wrapped}}
  {{name}} is awesome.
{{/wrapped}}

// Object
{
  "name": "Kris",
  "wrapped": function() {
    return function(text) {
      return "<b>" + render(text) + "</b>"
    }
  }
}

<!-- Output -->
<b>Kris is awesome.</b>
```

4 Design

The research conducted for the background information and related work was essential in the design of the *jtc* template engine. The ideas and decisions regarding the development and refinement of the template engine are discussed in this section.

4.1 Initial Designs

The development process began with the idea of applying FP ideas to the template engine space. To that end, the concept of higher-order templates was developed. The goal was to create a template engine in which templates could take other templates as parameters, much like higher-order functions are passed as parameters in FP. Two examples that came out of the design process are presented below.

4.1.1 XML Templates

The earliest design of *jtc* was an XML-based template engine. The idea was that template files would be written in a custom XML document that also contained the HTML to be given back to the user. A special `templates` tag would exist at the top of a given document, defining the reusable "template functions" that could be called within the document. The following example shows an early idea of what this could look like.

```
<!doctype html>
<html lang="en">
  <templates>
    <template name="showName" args="showFirst, showLast, firstName, lastName">
      <apply template="showFirst">firstName</apply>
      <apply template="showLast">lastName</apply>
    </template>

    <template name="showBold" args="data">
      <b>data</b>
    </template>

    <!--
      This template will return the first and last names surrounded by bold tags.
      It takes templates as arguments, illustrating the concept
      of higher order templates.
    -->
    <template name="showBoldName" args="firstName , lastName">
      <apply template="showName">showBold, showBold, firstName, lastName</apply>
    </template>
  </templates>

  <head>
    <meta charset=" utf 8 " />
    <link rel="stylesheet" href="styles.css"/>
    <script type="text/javascript" src="scripts.js"></script>
    <title>Test Page</title>
  </head>
  <body>
    <h1>Pittsburgh Sports Page</h1>
    <div id="about">A Pittsburgh sports blog</div>
    <div id="athletes">
      <!-- Will be rendered as: <b>Sidney</b> <b>Crosby</b> -->
      <showBoldName args="Sidney, Crosby"/>
    </div>
    <div id="footer">Copyright 2011</div>
  </body>
</html>
```

The example above illustrates the first iteration of designing higher-order templates. The template function `showName` takes as its arguments two other templates and two string values. The template `showBoldName` passes a template function to put its argument in bold to `showName`, where the final result would return a first and last name surrounded by bold tags.

Implementation was tentatively planned to use the XSLT transformation language. The syntax has some issues, and was not developed beyond this stage. Readability and the density of the syntax were two primary concerns.

4.1.2 Pure JavaScript Templates

The second design for *jtc* was similar in spirit to Ocsigen templates (Section 3.4), in which each HTML tag would be mapped to a corresponding function. The language JavaScript was chosen for its close relationship to web development. In this design, templates would be created through a series of nested functions, as demonstrated in the following example. The example, analogous to the XML-based example above, is commented to explain each piece of this design:

```
function showBold(data){
  return "<b>" + data + "</b>"
}

function showName(showFirst, showLast, firstName, lastName){
  // showFirst and showLast are function parameters
  // in this example, both are given the showBold function
  showFirst(firstName)
  showLast(lastName)
}

function showBoldName(firstName, lastName){
  return showName(showBold, showBold, firstName, lastName)
}

doctype()

// an empty call to html() would include these attributes,
// but attributes can be explicitly set using hashes, as so:
html({
  'xmlns':'http://www.w3.org/1999/xhtml',
  'lang':'en'
})
// also note that html just got closed, but we see that nesting
// tags by nesting function is the later convention. however,
// the html node never has anything except for head and body.
// with that in mind, the three "essential" tags have been separated,
// though they technically are nested within <html>

head(
  meta({'charset':'utf-8'})
  style('styles.css')
  script('scripts.js')
  title("Test Page")
)

body(
  h1('Pittsburgh Sports Page')
  // div can take a normal hash as other functions, but for
  // single attributes the (key, value) syntax is allowed
  div('id','about').("A Pittsburgh sports blog")

  div('id','athletes').(
    showBoldName("Sidney", "Crosby")
  )

  div('id','footer').(
    p('Copyright 2011')
  )
)
```

Ultimately it was decided that this template style was undesirable for a number of reasons. While the function application approach does provide benefits such as the ability to check document validity, it does obscure from the developer what the HTML looks like during development. One advantage of the XML or HTML approach provided in Section 4.1.1 is that the template code can be easily scanned to see what exactly the HTML result will look like. The pure JavaScript design as given above was not developed any further.

4.2 Final Design and Rationale

The final template design chosen combined elements of the early designs, as well as some ideas put forth by the related work in the space. Further design decisions were made with respect to the project goals (Section 2.4):

Multitude of technologies
JavaScript is used across the entire project. This unifies the technologies as much as possible, and provides the FP capabilities necessary for *jtc*.

Simple, Flat Data Presentation
With JavaScript functioning as both *controller* and *view*, its first-class functions and closures enable the passing of rich data structures to *jtc*.

Simplistic Templates
The empowerment of *jtc* through the data it is given and *jtc's* concept of higher-order templates make *jtc* templates much more capable that simple HTML "fill-in-the-blank" structures. It also decouples the *view* from the *controller* to better separate concerns.

It should naturally follow that *jtc* was named for its composition: ***J**avascript **T**emplates and **C**losures.*

4.2.1 Chosen Technologies

JavaScript As stated, the chosen language to be used with *jtc* is JavaScript. JavaScript is the *lingua franca* among web developers, always in use regardless of the choice of server-side platform. Though the language is primarily used for dynamic HTML, it has been utilized for many more programming tasks.

JavaScript provides first-class functions and closures, both FP necessities for *jtc*. JavaScript is also an ideal choice for the target audience. The use of the aforementioned FP elements is common enough that many developers write code leveraging these traits without even realizing it. JavaScript's event-driven architecture built around callback functions requires the use of first-class functions, creating a large group of programmers who could learn FP concepts through knowledge indirectly instilled via the language. JavaScript also provides the opportunity to unify the framework by writing both client and server-side code in the same language, using the tools outlined later in this document.

There are some complete web frameworks available in JavaScript, including Geddy [16]. But Geddy is designed to be similar to Ruby on Rails or Django, not an FP-inspired template engine.

Node.js Node.js (Node) [17, 18] is asynchronous, event-driven, server-side JavaScript framework that runs on Google's V8 JavaScript engine. It uses callbacks to achieve a completely non-blocking system of IO, letting it scale for high-performance servers on a single thread.

Node is used for the server of the project framework. Its event-driven architecture uses JavaScript's functional capabilities (anonymous functions, closures) for its implementation. The concept of the event-driven server was also used in the Ocsigen [6, 19] and Lift [7] frameworks.

Node provides very little "out-of-the-box" beyond its basic server scripting ability, which was desirable for this project. For more features, Node integrates with "modules" which can extend the functionality of Node.

Database A new object-relational mapper was not a goal of this project, but a database is desirable to illustrate some capabilities of the framework and template engine. Two database systems were used during this project:

MySQL and Sequelize The relational database MySQL was chosen to serve as the model layer of the project. The Node module Sequelize [22] was used to integrate MySQL into the project. Sequelize was far enough along in development to allow the relation tables and querying of related objects. Sequelize provides an object-relational mapper for MySQL, allowing JavaScript objects to represent objects stored in the database. For example, a definition of an object [22]:

```
var Project = sequelize.define('Project', {
  title: Sequelize.STRING,
  description: Sequelize.TEXT
})
```

The above code defines a new model, `Project`. This model has two attributes, with `title` mapping to a `VARCHAR` in MySQL, and the `description` mapping to MySQL `TEXT` [22]. Upon each restart of the Node server, Sequelize will connect to the database and attempt to create the necessary tables for defined objects (should those tables not already exist). Objects can be created and updated within JavaScript using this object mapping. Additionally, relations can be defined between objects, with Sequelize creating the necessary join tables in the MySQL database.

Querying an object or group of objects and retrieving them in JavaScript form was a great benefit of the Sequelize module. Just like Node, Sequelize uses asynchronous code with callback functions to query the database, as shown below:

```
Project.find(
  {title: 'My First Project'},
  {fetchAssociations:true},
  function(project){
    //the project retrieved from the database
    //is available as a JavaScript object here
  }
)
```

By giving the `find` function the object containing the `fetchAssociations` setting, the returned JavaScript object includes references to the model's related objects from the database.

Note: *Some of the functionality of Sequelize changed during the development of this project, as Sequelize is in active development. For reference, the project uses version 0.4.3 of the module, and the examples in this document represent the capabilities of that version.*

4.3 Architecture

The following explains the architecture of the project framework:

- A user sends an HTTP request to Node
- Node receives the request, and gathers relevant data from the database
- Node passes data and closures to the *jtc* template engine, specifying a template to render
- The *jtc* template engine processes the specified template with the given data, returned the rendered HTML result to Node
- Node returns the HTML result to the user in an HTTP response

4.4 Functional Templates

The idea and use of functional templates is the core of this project. Writing templates in the *jtc* template language is built around four major components: *at-blocks*, *h-blocks*, *lookups*, and *includes*. Each plays a role in allowing developers to write templates in this style. Each component is conceptually expanded upon from a user's perspective below. Technical implementation of each component is discussed in Section 5.

4.4.1 At-blocks and H-blocks

At a basic level, *at-blocks* are blocks of code within a template file that allow JavaScript code within their bounds. This JavaScript is not executed client-side, instead the code within an *at-block* is executed server-side by the template engine before the HTML result is delivered to the user. *At-blocks* are denoted using the following syntax, giving them their name:

```
@{
  //code within an at-block
}@
```

Code within an *at-block* must be either valid JavaScript code, an *h-block*, a *lookup*, or an *include* (the latter three are discussed below). The implementation dictates that the last JavaScript expression of an *at-block* will be returned in string form as HTML to the user. Thus, by convention, two *at-blocks* appear in each template file: one for declarations and definitions, and one for output. The definition block executes statements, including variable assignment and function declaration. The output block contains a JavaScript expression (as opposed to a statement) which will be returned to the user as HTML. A primitive example:

```
@{
  function hello_world(){
    return #{
      <html>
        <h1>Hello world</h1>
      </html>
    }#;
  }
}@

@{
  hello_world()
}@
```

The above example introduces the basic concept of an *h-block*, denoted by `#{ }#`. *H-blocks* are HTML fragments in *jtc* that live within the JavaScript code of *at-blocks*. The idea is analogous to Ocsigen's embedded HTML syntax (Section 3.4) and Scheme's `quasiquote` (Section 3.4.1). Their power comes from the fact that they are a treatment of HTML as a first-class value. The syntax of these HTML blocks is similar to Links (Section 3.5). They can be assigned to variables, returned from functions—for all intents and purposes they can be treated as JavaScript strings. *H-blocks* can contain *at-blocks* as well, which allows them to hold more dynamic content, as demonstrated below:

```
@{
  function hello( name ){
    return #{
      <html>
        <h1>Hello, @{ name }@!</h1>
      </html>
    }#;
  }
}@

@{
  hello( "Jon" )
}@
```

However, it should be noted that *at-blocks* within *h-blocks* are restricted to JavaScript expressions, and not statements. This is not viewed as a limitation—the primary use case for nested *at-blocks* is inserting dynamic content in place of the *at-block*. And for the case of conditionals, the expression form is available through JavaScript's ternary operator. For example,

```
#{
  <h1>User Profile</h1>
  @{ (user.logged_in) ? showControls() : #{ <p>Logged out</p> }# }@
}#
```

More information about why expressions are used is detailed in Section 5.3.1.

4.4.2 Lookups

Data that can be used in both *at-blocks* and *h-blocks* can be delivered to *jtc* from Node. Any relevant data is passed in as a JavaScript object, which effectively functions as a hash. The object contains string values that map to either other strings, objects returned from Sequelize queries, or even JavaScript functions. This object is made available globally within the template.

Accessing the data provided to *jtc* by Node is done via *lookups*. Lookups occur when the `@!name` syntax is read. The example below demonstrates lookup syntax:

```
@{
  function hello(){
    return #{
      <html>
        <h1>Hello, @{ @!name }@!</h1>
      </html>
    }#;
  }
}@

@{
  hello()
}@
```

In this instance of the `hello` function, the *h-block* will query the object passed in from Node and retrieve the `name` field. The `@!name` syntax is valid both inside *at-blocks* and *h-blocks*.

4.4.3 Includes

With the ability to declare template functions within *jtc* comes the natural desire to include these functions on different pages of a web application. Files can be included across an entire site using the `include` function:

```
@{
  @include("common.js");

  // template specific functions...
}@
```

The `include` function as seen above would insert all of the declarations in "common.js" located in the template directory exactly as if they appeared at the line of the `include` call. The "common.js" file should be a single *at-block*, containing any common variables or functions needed for inclusion. Using this technique, any common functions or data that are useful to more than one page can be shared across several templates.

Note that the last defined function will be the one that is executed. When a function is defined and a file that defines a function with the same name is imported later, the imported function will be used. Between local and imported functions, whichever is defined last will be used.

4.4.4 FP Concepts Now Available

With the major tools of *jtc* in place, it is appropriate to demonstrate what the sum of these parts makes possible in *jtc*.

Higher-order Templates *H-blocks* provide the ability to break up HTML templates into functions, and interact with these templates just as if they were JavaScript functions. Higher-order templates, templates that take other templates as parameters, are possible using *h-blocks*. Consider a basic HTML template that will be the same for each page of a website, only substituting in the "content area" of each page:

```
@{
var my_content = #{
  <h1>Welcome to the site</h1>
  <a href="/login/">Log in here</a>
}#;

function page( content ){
  return #{
    <html>
    <head>
      <!-- Relevant stylesheets and meta tags -->
    </head>
    <body>
      <div id="content">
        @{ content }@
      </div>
    </body>
    </html>
  }#;
}
}@

@{
  page( my_content )
}@
```

The above example could be made more useful if the `page` function was included in every template, and the templates would each define their own content. The major takeaway is that the template function `page` is receiving an *h-block* as its argument.

concatMap An even more powerful example of the capabilities of higher-order templates is presented by *jtc's* `concatMap` function. The `concatMap` function takes a template and a JavaScript array as its arguments. It then applies the given template to each element in the list, accumulating the HTML string result to be returned. It is analogous to its namesake, the FP higher-order function `concatMap`. The `concatMap` function introduces the ability to take higher-order templates to another, more useful level. Consider a template function such as this:

```
@{
function artistTemplate(albumTemplate){
  return function(artist){
    return #{
      <div class="artist">
        <h2>@{artist.name}@ (@{artist.albums.length}@ albums)</h2>
      </div>
      <div id="all_albums">
        @{ concatMap(albumTemplate, artist.albums) }@
      </div>
    }#;
  }
}

var body = #{
  <h1>All Artists</h1>
  <div id="artists">
    // Apply the curried function to the artists made
    // available by Node
    @{ concatMap(artistTemplate(album_grid), @!artists) }@
  </div>
}#;
}@

@{
  page( body )
}@
```

The above example actually demonstrates two key abilities of *jtc*. First, the use of the `concatMap` function. It receives as its first argument a template function to display information about a musical album on a webpage. It can be assumed that the database defines many musical artists and albums. Given any one of those albums in the form of a JavaScript object, the `albumTemplate` function would display information about that album such as its cover image, liner notes, or its track listing in HTML form.

The second ability demonstrated is template currying. The `artistTemplate` function takes the aforementioned album template as its argument, and returns a function that expects an artist as a parameter. The returned function, given an artist, would itself return information about the artist followed by a collection of all of that artist's albums rendered by the original album template argument. The `body` variable uses `concatMap` once again to run this process for every artist provided to the template by Node.

The FP concept of function currying is therefore possible in *jtc* HTML templates. It is made technically possible by JavaScript's anonymous functions and closures.

Note: *A user guide in documentation form is available as an appendix at the end of document. It details the mentioned functions, and some of the utility functions to simplify template design that are not covered here.*

4.5 Limitations

There are some recognized limitations of *jtc*. Each limitation and its cause is discussed below.

4.5.1 XHTML Well-Formedness and Validity

Because of the way the output string is accumulated, there is no way to guarantee the resulting output is a valid or well-formed document. The responsibility falls on the end user to ensure that the structure of the document is correct, and that the document conforms to the schema named by its `doctype`.

4.5.2 Error Handling

The *jtc* system's error handling is quite limited in its current state, though it will attempt to gracefully handle `eval` exceptions. The syntax errors that are caught are invalid JavaScript, not invalid *jtc* syntax. The ability to give syntactical and semantic errors regarding *jtc* constructs (*at-blocks*, *h-blocks*, *lookups*, or *includes*) would be ideal. More information is available in Section 6.5.2.

4.5.3 Eval and H-block Expansion

There are some limitations presented by the use of `eval` with regards to *h-block* expansion. The implementation of *h-blocks* is detailed in Section 5.3.1, with a specific limitation on JavaScript statements detailed in Section 5.4.3. The use of `eval` and its limitations are detailed in Section 5.3.2.

4.5.4 Lack of Namespaces

As noted in Section 4.4.3, there is no concept of namespaces between local template functions and imported functions. Functions with the same name can redefine each other, as each is evaluated in the same environment. This could potentially be remedied by re-implementing `include` to use JavaScript's prototype inheritance to separate local and included functions. Though most use cases should not run into this issue, it remains a documented limitation.

4.6 Example Application

An example application was developed to showcase the concepts of *jtc*, and how it could be used to develop a web application. The general goals of the example were to:

- Provide a visually appealing demonstration of a working project using *jtc*
- Integrate with a database to show the way *jtc* flexibly displays data in different ways
- Use functions both passed from Node and existing in *jtc* template code

The example that fulfills those goals is Clef. Clef is a simple music application that allows users to display their purchased music albums. In terms of data relationships, users are connected to the albums they own and other users with whom they are friends. Albums in turn are connected to their own tracks and related to artists. The database schema was defined by Sequelize, and the data was inserted manually. Because the purpose of the example is to showcase templates, users are not provided the ability to alter their data beyond several available preferences. Data exists in the database solely for demonstration. A variety of features are demonstrated within Clef.

Including Common Functions Clef uses a single file, "common.js", to hold all of its functions and variables that are used across the site. The base HTML for the site, including a `doctype`, `head`, and `body` containing a simple menu will be used on every page. For that reason a `base` function is defined with takes three arguments: a page title, any additional HTML to insert into the `head`, and any additional HTML to insert into the `body`'s content section. This function is called on every page with the appropriate arguments to generate that page's HTML result. A typical Clef template is structured similar to the following:

```
@{
  @include("clef/common.js");

  function template_function(){
    //Define function specific to this template
  }

  var body = #{
    //Define HTML specific to this template, using the above function
  }#;
}@
@{
  //Use the base function to output HTML
  base(@!user.username,null,body)
}@
```

Other included functions include a menu function that displays the available menu options relative to a user's authentication state, a function to display albums in a grid that is used in many pages, and an array of functions for sorting albums.

Template Currying The currying example provided in Section 4.4.4 is taken from Clef. It shows on a small scale the power of higher-order templates. The idea is taken a step further in the template for a user's profile page:

```
<h2>Albums</h2>
<div id="albums" class="group">
@{
  concatMap( (@!user.albums.length > 8) ? compressed_grid : album_grid,
    @!user.albums.sort(asf[@!prefs.album_sort]))
}@
</div>
```

This use of `concatMap` uses JavaScript's ternary operator to use the `compressed_grid` if the user owns more than eight albums, or the standard `album_grid` function otherwise. In this case the sorting of the albums is provided by the user's album sorting preference. That preference is an index which is used to retrieve a sorting function available in the included template functions from "common.js".

Function Location The above idea of the preference holding an index into an array of functions living in the *view* is an example of the flexibility of *jtc*. The system allows functions in either the *view* or *controller*. Instead of storing the album sorting functions within a template, they could be stored in the *controller* (Node) and passed to *jtc*. This is done for the page that displays all the albums that Clef has in its database:

```
server.get("/albums/([0-2])?$",function(req,res,number){
    if(number === undefined){
        number = 0;
    }
    Album.findAll({fetchAssociations:true},function(albums){
        res.write( jtc.render("clef/albums.html",{
            cookies:get_cookies(req),
            albums:albums,
            sort_function:album_sorts[number],
            sort_index:parseInt(number),
            get_album_length: get_album_length
        }));
        res.end();
    });
});
```

The server will listen for HTTP GET requests on "/albums/", and will grab the number after the slash (invalid numbers will result in a 404 error). That number corresponds to an array of sort functions on the server (`album_sorts`), and the function at the given index is passed to *jtc* in order to sort the albums. Typically, as seen in Django, RoR, or Snap for example, functions must be integrated with the *controller* code. With *jtc*, presentational functions can be embedded directly within template code. Functions can exist in either application layer depending on specific need.

Block Depth The ability to nest *at-blocks* and *h-blocks* provides more presentational power to the view. Consider the above example which displays all available albums. If the albums are sorted by time length, it would be useful to display the length of each album. This is achieved in the album template with a few nested blocks:

```
@{
(@!sort_index === 2) ?
    #{<br/> @{ (@!get_album_length(album)/60).toFixed(2) }@ min}# : ''
}@
```

First the template checks if the length sort function is being applied (it has the index "2"). If it is, an HTML break tag is inserted, followed by another *at-block* to use JavaScript to perform math on the number of seconds of the album (done by the `get_album_length` function, also passed from Node). The inner *at-block* closes, and the word "min" is inserted after that calculation of minutes is done. Of course, if the albums are not being sorted by length, an empty string is returned, signifying nothing being added to the document.

The ability to embed the power of a full programming language into the template is much different than Django, which uses a special Django template language as opposed to Python (Section 6.3). This ability is similar to RoR, which does allow the full embedding of the Ruby language in its templates. In both RoR and *jtc*, developers are encouraged to utilize the power of this feature while still maintaining a separation of concerns in the code.

The full Clef example's source is made available with this document for further examination of *jtc's* abilities.

5 Implementation

This section describes the implementation of the *jtc* template engine and the framework surrounding the template engine. Components of the framework considered orthogonal to *jtc* were implemented either using external modules, or created to provide the minimum functionality to demonstrate concepts of *jtc*. Each component of the framework is described in detail below.

5.1 URL Routing

The nature of web applications requires any framework to hold some form of a URL router. This piece of any given framework receives the request for a certain URL from the client. The router directs the client's request to the appropriate *controller* to render the corresponding *template*. An example from Django was presented in Section 3.2.1.

For this project, the open source *node-router* project [23] is used as the URL router. This Node module allows URL requests received by Node to be processed, calling the *jtc* template engine with the appropriate template files to create its HTML result. The code that uses *node-router* is within the Node server. An example to illustrate:

```
var node_router = require('node-router');
var server = node_router.getServer();
var jtc = require('jtc');
jtc.template_dir('templates/');

//...

// User profile
server.get("/user/([a-zA-z0-9]+)/?$", function(request, response, username){
  User.find({username: username},{fetchAssociations: true}, function(user){
    response.write( jtc.render("clef/profile.html", {
      cookies: get_cookies(req),
      user:    user
    }));
    response.end();
  });
});
```

The declarations at the top of the example code import *node-router* and *jtc*, making both modules available to Node. The call to `server.get` sets up the server to listen for HTTP GET requests at the specified URL, a regular expression. Because parentheses surround the portion of the string after "user/", that value is passed in as a parameter (`username`) to the callback function.

Within this callback function, the User object (mapped to a MySQL table through Sequelize) is used to query for the given username. More details on the Sequelize query are explained in Section 4.2.1. The final code executed will be the HTTP response object being written to with the contents of *jtc's* render function. The `response.end` function signifies that no more data will be written to the HTTP response, and it can be returned in full to the user. Again, the *node-router* module only provides the ability to match regular expressions to the specified callback functions. The other functionality in this example is provided by Node, Sequelize, and *jtc*.

5.2 Data Simulation

Web applications must connect to a database to store all of their information. Most modern frameworks leverage an object-relational mapper to make working with the database more natural in the application codebase. Building a new ORM was not a goal of this project, but to demonstrate some of its capability data was needed. The Sequelize module was used as an ORM for Node, and data was entered manually into a MySQL database to provide a test set for the project.

The user data in the database was partly used for authentication, to simulate different users logging in and out of the example site. The authentication system was implemented using very simplistic cookies set via HTTP headers. This practice is in no way secure or practical for production systems, but was useful for demonstration purposes of this project. Implementing this simplistic authentication was satisfactory to simulate user access and avoid the introduction of another external dependency in addition to Sequelize.

5.3 Template Engine

Of course, the *jtc* engine is the primary piece of the framework. This section details how the *jtc* template engine is constructed.

5.3.1 The Language

When Node calls `jtc.render`, it passes the `render` function two arguments—a template filename, and a JavaScript hash. `Render` then loads the template file specified from disk, makes the JavaScript hash available in the engine's execution environment, and begins parsing the template.

Grammar The grammar is relatively straightforward. The symbol *AB* represents *at-blocks*, *HB* represents *h-blocks*, *LK* represents *lookups*, and *INC* represents *includes*:

$S \leftarrow \circ\ S$
$S \leftarrow @\{AB\}@\ S$
$AB \leftarrow \circ\ AB$
$AB \leftarrow @\{AB\}@\ AB$
$AB \leftarrow \#\{HB\}\#\ AB$
$AB \leftarrow LK\ HB$
$AB \leftarrow INC\ HB$
$AB \leftarrow \epsilon$
$HB \leftarrow \circ\ HB$
$HB \leftarrow @\{AB\}@\ HB$
$HB \leftarrow LK\ HB$
$HB \leftarrow \epsilon$
$LK \leftarrow @!JSIDENTIFIER$
$INC \leftarrow @\mathit{include}(FILENAME)$

Parsing Template files are parsed with each character as a token. The recursive descent parser continues to descend at each level of *at-blocks* or *h-blocks*, and within appropriate blocks descends into *lookups* or *includes* (or sub-blocks).

To accomplish this task, the parser is divided into several major parse levels. The topmost parse function (`parse`) contains an index and the template file being parsed, which all the functions access. Because no backtracking is necessary, the index need not be passed to each parse level and can remain a global index. The topmost function starts the index at zero, and enters the first parse level, `outerParse`.

The `outerParse` function iterates over the incoming tokens, only looking for the beginning of an *at-block*. Otherwise, it is simply echoes each token to the output string that will be sent back to the user as HTML on parse completion. Should it find the start of an *at-block* (`@{`), the parser descends into the `atParse` function.

Once within the `atParse` function, there are five possible routes the parser can take, depending on what it finds:

Start of Another At-block
: Recursively call itself and append the result to the current `atParse`'s result

Start of an H-block
: Call the `hParse` function and append the result to `atParse`'s result

A Lookup
: Call the `lookup` function and append the result to `atParse`'s result

An Include
: Call the `include` function and append the result to `atParse`'s result

End of the At-block
: Return the result to the caller

The tokens that do not trigger any of the above outcomes are continually appended to the result to be returned. Each time that `atParse` returns to `outerParse`, `outerParse` uses JavaScript's `eval` function to execute the JavaScript code returned by `atParse`. `Eval` is explained in Section 5.3.2, but ultimately this means that the string that `atParse` returns must be valid JavaScript code. Since *at-blocks* contain valid JavaScript, their content can be accumulated normally. There are measures taken to handle comment-only *at-blocks* and ignore their output. Measures also must be taken to create valid JavaScript from the other calls from within `atParse`.

The `includeParse` function is invoked when `atParse` reads `@include(`. After being called, `includeParse` reads until it hits a closing parenthesis, then loads the filename between both parentheses from the template directory. The contents of that template are inserted directly in place of the `@include(...)` call as if they were in the calling document. When `includeParse` returns, `atParse` begins parsing again at the start of the include, so its contents are immediately processed in the order expected from the user.

The `lookup` function is invoked when the parser reads the `@!` tokens. After the bang token, `lookup` greedily grabs as many valid JavaScript variable name tokens as it can (alphanumeric, underscores, and dashes). Periods are also read, as JavaScript syntax allows object properties to be accessed via those characters. The lookup is returned as a string which access the environment array that was provided from Node, such as the lookup "@!artist.albums" producing the valid JavaScript result "jtcEnv.artist.albums".

The `hParse` function is invoked after `atParse` reads the beginning of an *h-block* (`#{`). There are three possible routes `hParse` can take:

Start of an At-block
: Call the `atParse` function and append the result to `hParse`'s result

A Lookup
: Call the `lookup` function and append the result to `hParse`'s result

End of the H-block
Return the result to the caller

As with `atParse`, tokens that do not trigger any of the above outcomes are continually appended to the result to be returned. The string that is accumulated from the content of the *h-block* and the values from any `lookup` or `atParse` calls is returned to the `hParse` caller. However, since the caller is known to always be `atParse`, the returned string must be valid JavaScript so that `atParse` can safely accumulate it. Since *h-blocks* contain HTML fragments and not valid JavaScript, their content is escaped and transformed into a parenthesized string concatenation.

The `hParse` function begins its string accumulation with `(unescape(`, then continues to parse over its contents, escaping each character with the `escape` function. This prevents whatever was inside the string from potentially getting evaluated as code; an unintentional injection of statements would be possible otherwise. When it does finally return to `atParse` and is eventually executed by `outerParse` with `eval`, the call to `unescape` will occur. The result will be the original HTML string as entered by the user. To maintain this string, the parenthesis to close the original `unescape` is appended before the result from a `lookup` call, with the result preceding another `(unescape(`. This is better explained through the line from the actual *jtc* source code:

```
token = '") + (' + lookup + ') + (unescape("';
```

A similar step is taken when accumulating the returned value from the inner `atParse`, but `atParse` is a slightly different case. Because *at-blocks* contain JavaScript, it is possible that their result could return `undefined` if the block only contains void JavaScript expressions. For that reason, a rather rudimentary trick is applied—the result is passed to an anonymous function. The function will return an empty string if the result is undefined, or the string of the result if it is defined. This trick is commonly used in the client-side JavaScript world to establish a scope to protect code from other included libraries, and to get around some behaviors in incompatible browsers. Its use for this project is detailed in Section 5.4.3.

5.3.2 Eval

The *jtc* template engine is built around the capabilities of JavaScript's `eval` function. `Eval` takes a string as its argument and executes that string as JavaScript code within the current lexical environment. If the string is a JavaScript expression, it returns the result of that expression. If the string is a series of statements, it will return the value of the last statement (or `undefined` if there is no return value) [24, p. 641-642]. This behavior is why convention dictates two *at-blocks* in template files for clarity, as detailed in Section 4.4.1.

The code evaluated by the `eval` function is added to the executing environment and remains available for the duration of the caller's lexical scope. This behavior exists as the default in the Google Chrome V8 JavaScript engine. Though `eval` can be executed at a global level by the use of `eval.call` [25], that functionality was not needed for the purposes of this project.

`Eval` provides the only way to dynamically modify the execution environment in JavaScript, as is true in most other dynamic programming languages. It is for this reason that its functionality was leveraged in this project. The ability to use JavaScript essentially as a JavaScript interpreter proved very useful to this project's goals.

The `eval` function is very rarely recommended for use in client-side JavaScript because of the security risks. The possible execution of arbitrary code could open up significant holes for malicious scripts if not used properly. In this server-context where the only code execution is controlled by the developer, it may be considered generally safer. However, insertion of user-submitted data that is not cleaned or valid into the environment could still pose a potential risk. Though data validation was not part of this project, the author recognizes that in a production environment certain measures for safety of this system would have to be in place.

5.3.3 Utility Functions

The *jtc* template engine includes several small utility functions to speed up some of the common tasks in web development.

styles([stylesheets])
: The `styles` function takes one or more strings corresponding to stylesheets ("layout.css" for example), and returns the HTML tags to include those stylesheets on the page.

scripts([scripts])
: Like its sibling `styles`, `scripts` takes one or more strings corresponding to JavaScript files ("jQuery.js" for example), and returns the HTML tags to include those scripts on the page.

concatMap(template_function, array)
: As demonstrated in Section 4.4.4, `concatMap` takes a template function and an array, applying the template function to each array element and accumulating the result to be inserted into the HTML of the page.

Because these functions are defined within the same evaluation context as the code being executed through `eval`, the dynamically evaluated code has access to these functions.

5.4 Problems During Development

5.4.1 MongoDB and Mongoose

MongoDB [20] is a document-oriented database that was originally chosen as the database for this project. MongoDB's documents are stored in a JSON format. With JavaScript being leveraged so heavily, this database seemed to be a natural fit. Connection to the database was integrated using the Mongoose [21] Node module.

Since MongoDB is a document-oriented database and not a relational database, relations are done through the DBRef standard [20]. DBRef objects allow documents to reference each other in a fashion somewhat analogous to joining tables in a relational database. As the example web application was being developed, the limitations of Mongoose were reached when it was discovered that the Mongoose module did not natively support DBRef. For this reason, Mongoose and MongoDB were abandoned. But, the time spent on research and integration of these technologies was significant and therefore, though abandoned, the technologies are mentioned here.

5.4.2 Sequelize

Sequelize solved one limitation of Mongoose as detailed in Section 4.2.1, but it did introduce a minor inconvenience of its own. Sequelize allows the retrieval of associated objects through its object-relational mapper, through the following syntax:

```
// 1. Query with fetchAssociations set to true
Artist.find(
  {title: 'Radiohead'},
  {fetchAssociations:true},
  function(artist){
    // 2. Use the fetchedAssociations property
    // to access the related objects
    var albums = artist.fetchedAssociations.albums;
  }
)
```

In the above example, imagine albums were related to *tracks*; a given album's tracks would not be accessible via `artist.fetchedAssociations.albums[0].tracks`. The issue is that there is no way to fetch related objects' related objects, better stated as the transitive closure of an object's associations. While this did not limit the framework's functionality in any way, it would make working with some related data easier.

5.4.3 Statements in H-blocks

An *at-block* that is nested within an *h-block* is must be a JavaScript expression due to the way *h-blocks* are concatenated together. Characters such as the `var` keyword or the use of semicolons break this concatenation process, as demonstrated below:

```
// The h-block
#{
  <h1>Hello, @{ var x = getName(); }@</h1>
}#

// The resulting concatenation (though the HTML would be escaped)
"<h1>Hello, " + var x = getName(); + "</h1>"
```

The above concatenation is clearly invalid. However, the requirement of expressions in nested *at-blocks* is not limiting, as their use case is precisely for inserting the values of JavaScript expressions into HTML. A similar problem must be dealt with when nested *at-blocks*, though they are in expression form, return `undefined`. This could be caused by void functions, for example. To overcome this issue, all nested *at-block* results are surrounded with a wrapper function, as so:

```
(function( z ) {
  if( z ){
    return z;
  }else{
    return "";
  }
})( atString )
```

The anonymous function takes the result of the *at-block* as its parameter, and it is not `null` or `undefined`, returns it. Otherwise, it returns the empty string. This simple trick works for void expressions, but does not work for the aforementioned problem with statements in nested *at-blocks*. For that reason, only expressions are allowed inside non-top-level *at-blocks*.

5.4.4 Delaying Database Queries through Synchronization

The capabilities of the framework allow for functions to be passed from Node into *jtc*. One idea for a practical use of this functionality was to pass closures containing database queries into *jtc*, allowing the template to query the database if needed. This approach would allow Node, the controller, to pass these closures to *jtc* for any template—only the templates that needed to query the database would execute the function.

However, this task was ill-conceived based upon the entire system. With Node being a single-threaded server, it is completely tied to the idea of asynchronous function calls. The Sequelize module (and all major Node modules) are in line with this philosophy, and perform all of their computation through this asynchronous model. The model does not mix well with return values however.

The concept of calling a function that queries the database and waiting for its return value is practically impossible in the world of Node. Attempts were made to work around the issue in many forms, but the constant fighting with the Node model did not yield any useful results. Even an attempt at tricking the asynchronous functions into returning through the use of a busy wait was proven impossible because Node's single thread blocks on the busy wait's loop. Since this feature was non-essential to the project, it was abandoned after repeated attempts to implement it.

6 Results

This section discusses the project results and other information gathered as a result of this project.

6.1 Benefits

The benefits of the framework have been explored and demonstrated throughout this entire document. The *jtc* template engine integrated with Node provides developers a new perspective on templating, and what is possible with some FP concepts applied to the domain. In a team setting, the flexibility of providing functions that either come from the controller *or* the view would be a great asset. The function could live on either side depending on the team to which it is more closely related (presentational or developmental), while not requiring the team to isolate these functions to one area.

For example, consider a social networking site where the creators want to implement a friend sorting feature. The design team could simply ask the development team to create a user preference for friend sorting, assigning each preference to a key. Then the design team can write all the simple sorting functions themselves, and use the appropriate sort function based on the key passed to the *view* from the *controller*. The development team can add the simple preference, and pass to the *view* the user preferences and a friend iterator that accepts a sort function. This separation of concerns and responsibilities is desirable in a web development environment.

The project also illustrates how unifying web development through a single language can be beneficial. By using JavaScript across each facet of the application, the project allows rich data structures to be transferred seamlessly between the *controller* and *view*. This is made simple because the template language (code within *at-blocks* and *h-blocks*) knows the host language (JavaScript).

The *jtc* template engine demonstrates the concept of higher-order templates, and their power in writing the *view* layer. The use of higher-order JavaScript functions as higher-order templates also continues the aforementioned ideas of unifying with the host language.

6.2 Node.js and Functional Programming

At the start of this project, Node was selected as the server to be used because:

1. It is a relatively new technology, and researching a newer topic is a more appealing premise than researching an aging one.
2. It is server-side JavaScript, allowing the unification of a single language.
3. It uses JavaScript closures and anonymous functions, which tied into the FP goals of the project.

While the first two benefits certainly ring true even at the end of the project, the initial perception of Node as demonstrating some FP concepts may have been a bit misguided. The feeling was that since JavaScript is a Scheme-like language (as noted by Crockford [26]), and Node is able to use those features for its goals, Node must be FP. After working with the project, that assumption now seems less clear. In fact, it calls into debate whether *using* FP features in non-functional languages *is* functional programming. That discussion is worthy of an entire paper in itself.

Node is certainly at least related to FP because of its reliance on anonymous functions and closures. But the creators of Node made their goals clear: to experiment with the strategies of browser JavaScript on the server, hiding the event loop from the user [18]. The event-driven server is closely patterned after simple event callbacks in JavaScript such as `mouseover` or `onclick`. This event-driven style has been used in other FP web frameworks, as discussed in Section 4.2.1.

Node is built around the concept of completely non-blocking IO through the use of asynchronous programming. Another piece of information gathered from project work is that the fields of functional and asynchronous programming are completely independent of each other. Asynchronous programming can be done without FP, and FP can be written without being asynchronous. This seems obvious in retrospect, but was made more clear with each week in the project timeline.

6.3 Comparison within Field

Many template systems are limited to only inserting string data into an HTML file. This is partly due to the handling of HTTP GET and POST data being string based. It also is in some cases a language constraint, because the ability to pass closures between application layers is impossible in some languages. *Jtc* is able to overcome this simplistic structure because of JavaScript, allowing richer data to flow between *view* and *controller*.

In a larger sense, the *view* layer is almost an afterthought in terms of computation. For example, the Django template language is intentionally limited to force developers to constrain their code to the Django design pattern. While a separation of concerns is certainly a good design principle, this system does misplace some responsibility of presentation into the *controller* (known as the *view* in Django). The *jtc* approach is to give that flexibility to the developers, who then have the ability to distribute functionality to the area in which is fits more appropriately.

The power and reusability that these FP concepts grant is hard to achieve naturally in the related template engines. The curried template function in Section 4.4.4 would be difficult to replicate. In the Django template language, one could imagine something like the following:

```
<h1>All Artists</h1>
<div id="artists">
{% for artist in artists %}
  <div class="artist">
    <h2>{{ artist.name }} ({{ artist.albums.count }} albums)</h2>
  </div>

  <div id="all_albums">
    {% for album in albums %}
    <div class="album">
      <a href="/music/{{ album.artist_name }}/{{ album.title }}">
        <img src="/img/albums/{{ album.cover_url }}"/>
      </a>
      <h5><a href="/music/{{ album.artist_name }}">
        {{ album.artist_name }}</a></h5>
      <p class="small">{{ album.title }}</p>
    </div>
    {% endfor %}
  </div>
{% endfor %}
</div>
```

The above example works to display all the albums for each artist, displaying the album using the innermost HTML code. But, any template that uses this code will have to have this code copied into it, or inherit from some template that defines it. That kind of HTML fragment inheritance does not fall within Django's typical structure. And should the code need to be rewritten to change the HTML for displaying an album, it is much less trivial than simply changing a function argument. In the *jtc* example only the included album function used as the parameter needs to be updated, though this is partly a demonstration of inheritance strategy and not FP.

Rather, the FP capabilities are demonstrated in other scenarios. If the way to display an album is based on a user preference, the above Django code would become an unreadable mess of `if-else` statements. In *jtc*, the controller can simply pass the necessary template function to the view, and that function will be given to `concatMap`. Similar opportunities present themselves with passing sorting functions. Other common iteration structures in HTML generation can be more cleanly expressed with FP as the web is full of list data. Being able to succinctly and flexibly process that data is a major benefit of *jtc*.

6.4 Achievement of Proposal Goals

The proposal for this project stated the following goals. Each will be addressed in relation to the final result.

1. Create a template processing engine that simplifies client-side development through FP and boilerplate code generation. The highly expressive template language will allow developers to tersely express common web actions and automatically create AJAX code where appropriate. The *jtc* template engine provides the FP client-side development environment. The utility functions and the ability to reuse templates provide the boilerplate code. "Highly-expressive" and "terse" are relative terms and may have been poor word choices in the original proposal, but the *jtc* template language is concise and designed for simplicity.

The generation of AJAX code was a part of the proposal that was not implemented in the final version. Because of the myriad of ways to implement and design AJAX code, it was decided that automatically generating it was not in developers' best interest. Additionally questions arise about choosing a "sacred" JavaScript library that the project would use, as opposed to leaving the decision to developers. However, the example does demonstrate how AJAX code can be embedded in templates just as with other template engines.

2. Maintain accessibility and ease of use to both experienced and inexperienced FP developers. The aim is to create a system that can be effectively used by web developers regardless of their prior FP experience. This goal was the one of the primary motivations to use JavaScript for the framework language. JavaScript's ubiquity among web developers means that an engine like *jtc* should have a relatively low learning curve for the target audience. In this regard it is similar in spirit to Node. And as stated, no FP background is required to use *jtc*, though it does use FP concepts.

3. Demonstrate that functional programming is an ideal choice for web development by solving common web problems using FP concepts. *Jtc* attempts to present FP concepts as useful to template engines. The research presented in this document also demonstrates FP as useful to web development outside of template engines.

Additionally the proposal noted that the following pieces of a web framework would be delivered, with the delivered result in parentheses:

- A web server (*Node*)
- Server-side scripting for application logic (*Node*)
- Client-side scripting for dynamic interaction (*See Section 6.4*)
- A template engine for dynamically generated content (*jtc*)

6.5 Further Work

Though the concepts have been demonstrated, there are many features that could benefit the *jtc* template engine to prepare it for production environments.

6.5.1 Improve the Parser

The *jtc* parser went through several iterations, but always was focused on the most simple solution possible so that the project concepts could be demonstrated. Improvements could be made such as using a formal parser-generator to process the input.

6.5.2 Error Handling

Related to the parser, currently error handling is minimal. `Eval` exceptions are tolerated, resulting in no output for a given *at-block* being appended to the HTML response. But if the *at-block* that throws the exception is a top-level *at-block*, the page will not likely be able to display anything. Errors are logged to Node, but a more robust error checking system would benefit future versions of the project. Perhaps the HTML output could be replaced by parser errors during development. A syntax highlighter for popular editors would also help limit some simple errors during development.

6.5.3 Performance

Performance under high server load was not a goal of this project, so its ability to scale to large systems is untested. The parser improvements and allowing the accumulated HTML result to append by larger tokens than just characters would likely benefit performance.

References

[1] E. Cooper, S. Lindley, P. Wadler, and J. Yallop, "Links: Web programming without tiers," in *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*, FMCO'06, (Berlin, Heidelberg), pp. 266–296, Springer-Verlag, 2007.

[2] D. Heinemeier Hansson, "Ruby on rails." http://rubyonrails.org/, 2011.

[3] D. S. Foundation, "Django: The web framework for perfectionists with deadlines." http://www.djangoproject.com/, 2011.

[4] G. Collins, D. Beardsley, S.-y. Guo, J. Sanders, C. Howells, S. O'Brien, O. Ataman, and C. Smith, "Snap framework documentation." http://snapframework.com/docs/.

[5] E. Inc., "Codeigniter - open source php web application framework." http://codeigniter.com/.

[6] V. Balat, "Ocsigen: Typing web interaction with objective caml," in *Proceedings of the 2006 workshop on ML*, ML '06, (New York, NY, USA), pp. 84–94, ACM, 2006.

[7] D. Ghosh and S. Vinoski, "Scala and lift: Functional recipes for the web," *IEEE Internet Computing*, vol. 13, pp. 88–92, May 2009.

[8] N. Welsh and D. Gurnell, "Experience report: scheme in commercial web application development," *SIGPLAN Not.*, vol. 42, pp. 153–156, October 2007.

[9] W3C, "Http - hypertext transfer protocol overview." http://www.w3.org/Protocols/.

[10] M. Elsman and K. F. Larsen, "Typing xhtml web applications in ml," in *In International Symposium on Practical Aspects of Declarative Languages (PADL04), volume 3057 of LNCS*, pp. 224–238, Springer-Verlag, 2004.

[11] M. Gunderloy, M. Lindsaar, and J. Iniesta, "Ruby on rails guides: Layouts and rendering in rails." http://guides.rubyonrails.org/layouts_and_rendering.html, April 2010.

[12] G. Pollack, "An introduction to rails." http://railsforzombies.org/, 2010.

[13] R. K. Dybvig, "The scheme programming language, third edition," 2003.

[14] C. Wanstrath, "mustache." http://mustache.github.com/.

[15] C. Wanstrath, "mustache(5) – logic-less templates.." http://mustache.github.com/mustache.5.html, April 2010.

[16] M. Eernisse, "geddy: A modular, full-service web framework for node.js.." http://geddyjs.org/.

[17] S. Tilkov and S. Vinoski, "Node.js: Using javascript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, pp. 80–83, November 2010.

[18] R. Dahl, “Node.js: Evented i/o for javascript.” http://nodejs.org/.

[19] V. Balat, J. Vouillon, and B. Yakobowski, “Experience report: ocsigen, a web programming framework,” in *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP ’09, (New York, NY, USA), pp. 311–316, ACM, 2009.

[20] 10gen Inc., “Mongodb.” http://www.mongodb.org/.

[21] G. Rauch, N. White, B. Noguchi, and A. Heckmann, “Learnboost/mongoose.” https://github.com/LearnBoost/mongoose.

[22] S. Depold, “Sequelize: A mysql object-relational-mapper for nodejs.” http://www.sequelizejs.com/, 2011.

[23] T. Caswell, “node-router.” https://github.com/creationix/node-router, August 2010.

[24] D. Flanagan, “Javascript: The definitive guide, fifth edition,” August 2006.

[25] M. D. Network, “call - mozilla developer center docs.” https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Function/call, March 2011.

[26] D. Crockford, “The little javascripter.” http://javascript.crockford.com/little.html.

A Walkthrough and Documentation

This appendix is designed as an informal documentation of this project through the building of a simple application that uses the *jtc* template engine. All pieces of software are included with this document. A syntax reference is provided in Section A.9.

A.1 Setup

The first step is to install Node.js (version 0.4.3). Unarchive `node-v0.4.3.tar.gz` and follow the instructions in `README.md`.

All of our work will be done within the `example` folder, where the required modules are included in the `node_modules` subfolder. If you run into any problems or would rather just read along, the `completed_example` folder has a working version of the project this appendix describes.

A.2 First Steps

The `example` folder should contain:

- `server.js`
- `templates/`
  - `homepage.html`

The `server.js` file already contains the necessary import statements for a *jtc* project. It should look like this:

```
var node_router = require('node-router');
var server = node_router.getServer();
var jtc = require('jtc');
jtc.template_dir('templates/');
```

A.3 Adding a Route

The first step is to connect a URL to some HTML we want to display. To map a URL to a template, we first create the appropriate function in `server.js`. Typical HTTP methods (GET, POST, PUT, DELETE) can be used. For this example, GET is used. Add the following simple route to the server file after the pre-existing declarations:

```
server.get('/homepage/?$', function(request, response){
  response.write( jtc.render('homepage.html', {
    title : 'Student homepage',
    name : 'Justin',
    bio : 'A CS student at RIT'
  }));
  response.end();
});
```

The last piece of this initial route setup is to start the server listening on a given port. Add this line at the bottom of `server.js` to have the server listen on port `8080`:

```
server.listen(8080,'127.0.0.1');
```

The behavior is now defined. But, since the template does not exist, it is not yet ready to be used.

A.4 Building a Template

The `homepage.html` template must be edited so that the above route can render HTML back to the user. Edit `homepage.html` to the following:

```
@{
function body(){
  return #{
  <html>
    <head>
      <title>@!title</title>
    </head>
    <body>
      <h1>@!name</h1>
      <h2>@!bio</h2>
    </body>
  </html>
  }#;
}
}@

@{
  body()
}@
```

Notice the use of two *at-blocks*, one for declaration and one expression to be rendered to the user. In the first, the `body` function is defined. It returns an *h-block* that will be the HTML of the page we are rendering. Inside the `title`, `h1`, and `h2` tags are *lookups*. These *lookups* refer to the information that was passed to the template by the route we created earlier. They are invoked using the *@!key* syntax, as shown above.

To see this page in action, open a shell and navigate to the directory that contains `server.js`. Enter `node server.js` to start the node server with our server file, then point your browser to `http://localhost:8080/homepage/`. The page is the HTML result with the data from Node inserted.

Note: *Other URLs, such as the visiting the basic `http://localhost:8080/` page, will return an HTTP 404 error.*

A.5 Adding a Function

One of the unique capabilities of *jtc* is the ability to pass functions to the template engine. We'll add a function to our route to compute the time since this student began studies at his college. Rewrite the route so that it reads as follows:

```
server.get('/homepage/?$', function(request, response){
  response.write( jtc.render('homepage.html', {
    title : 'Student homepage',
    name : 'Justin Cady',
    bio : 'A CS student at RIT',
    time_in_college : function(){
      var rit_start = new Date(2006,8,4);
      var today = new Date();
      var one_day_ms = 1000*60*60*24;
      return ((today.getTime() - rit_start.getTime())/one_day_ms);
    }
  }));
  response.end();
});
```

This function creates a Javascript `Date` object representing the student's first day at RIT, and computes the difference in days from that date until the current date. Of course, to display this information the template must be updated as well. Add this snippet right below the `h2` tag in `homepage.html`:

```
<p>I have been at RIT for @{ @!time_in_college() }@ days</p>
```

Inside the *h-block*, Javascript can be executed by created an inner *at-block*. Here, we are opening a block to execute Javascript, and calling the function that was passed over through a *lookup*. Restart the Node server by entering `CTRL-C` at the shell prompt, then re-executing the `node server.js` command. Any time that we change `server.js`, we must remember to restart the server.

A.6 Another Function

Our first function was passed from Node to *jtc*, but functions can exist solely within *jtc* templates as well. We will add a template function to display our student's favorite programming languages. Let's first add the languages to our route, rewriting it so that it now looks like this:

```
server.get('/homepage/?$', function(request, response){
  response.write( jtc.render('homepage.html', {
    title : 'Student homepage',
    name : 'Justin Cady',
    bio : 'A CS student at RIT',
    time_in_college : function(){
      var rit_start = new Date(2006,8,4);
      var today = new Date();
      var one_day_ms = 1000*60*60*24;
      return ((today.getTime() - rit_start.getTime())/one_day_ms);
    },
    languages: [
      'Javascript',
      'Python',
      'Objective-C',
      'Haskell',
      'SML'
    ]
  }));
  response.end();
});
```

With the languages array added to the information our template is receiving, we must edit the template to display this information. Since this is a collection of data that we want to display similarly in the template, it is an ideal situation to use the `concatMap` function of *jtc*. The `concatMap` function takes a template function and a list of elements, such as our language array, as arguments. Edit `homepage.html` to look like this:

```
@{
function language_template(lang){
  return #{
    <li>@{ lang }@</li>
  }#;
}

function body(){
  return #{
  <html>
    <head>
      <title>@!title</title>
    </head>
    <body>
      <h1>@!name</h1>
      <h2>@!bio</h2>
      <p>I have been at RIT for @{ @!time_in_college() }@ days</p>
      <h3>My favorite programming languages</h3>
      <ul>
      @{ concatMap(language_template, @!languages) }@
      </ul>
    </body>
  </html>
  }#;
}
}@

@{
  body()
}@
```

The function `language_template` is a template function that takes a language string as an argument, and renders it as a `li` tag. Note that this function only exists in the template itself. This function is used by `concatMap` inside the `ul` to render the list of the favorite programming language strings. Restart the Node server and verify this new data is displayed.

A.7 Expanding the Language Function

As a small example of the abilities that *jtc* provides, we will now add some random colorization to the list of languages. The `language_template` function will access a random color from an array. To accomplish this, edit the top of the template as follows:

```
@{
var colors = [
  'red',
  'blue',
  'green',
  'purple',
  'orange'
];

function language_template(lang){
  return #{
    <li style="color:@{ colors[Math.floor(Math.random()*5)] }@;">@{ lang }@</li>
  }#;
}
...
```

The function now uses Javascript's `Math.random` function to randomly create an integer from `0` to `4`, and insert the corresponding color as the color of the list item. There is no need to restart the server, as we have only changed the template file. Reload the page to see these changes. As this demonstrates, any Javascript code can be used in these template functions.

A.8 Going Further

This tutorial was designed to get new developers started with the basic principles of *jtc*. Using the concepts above, powerful templates can be designed and utilized. For more information, review the *Clef* source code included with this document.

A.9 Syntax Reference

At-block

`@{ ... }@`
Contains Javascript functions and statements. A template file that is going to render HTML should contain two `at-blocks`—one for declarations and one containing an expression for output. A template file used for includes need only contain one `at-block`.

H-block

`#{ ... }#`
Contains HTML spliced with `at-blocks` or `lookups`. A `h-block` can be assigned to a variable, used as a parameter, or returned from a function.

Lookup

`@!...`
A lookup is used to insert data that was passed in via the *jtc* hash. The string following the bang will be used as the key into the hash, and its value will be inserted in place of the `lookup`.

Include

`@include( ... )`
Load an external template file into another template file. The string parameter should be a filename relative to the *jtc* template directory. That file will be inserted at the exact point of the include statement.