The Scheme Pet Store

Ben Simon (benjisimon@gmail.com)

October 10, 2005

1 Introduction

As a programmer who spends a good portion of his time developing web applications, I'm always on the lookout for new frameworks or methodologies. When I saw the announcement for SISCweb [11], I realized that it appeared to solve a number of key web development issues. As the name suggests, SISCweb is a web application framework built on top of the SISC [17] Scheme implementation. Some of the immediate benefits of using SISCweb include the following:

- Allows you to use any Java API or library
- Applications can be hosted by any standard J2EE servlet container
- High quality Scheme implementation which fully implements the latest Scheme standard
- Development is highly interactive allowing the author to make changes to any aspect of the system without needing to restart the server
- Continuation-based development model allows for a novel implementation of web applications

While a variety of frameworks address one or two of the above items, SISCweb is fairly unique in offering all of them. I decided to test this powerful combination of features in the hopes of demonstrating the advantages of SISCweb and Scheme as a web application development environment.

2 Choosing an Application

Choosing an application to demonstrate the capabilities of SISCweb turned out to be relatively easy. SUN essentially set the industry standard for a sample application when they published their Java Pet Store [1]. Since then, others have followed suit, which has led to .Net [3], ColdFusion [8] and Flash versions [4].

One aspect that makes the pet store ideal is its relative complexity. It requires the programmer to account for common web requirements, such as managing server side state and the execution of transactions. These details regularly go into a real life web application, while samples can easily forgo them.

3 Key Principles

I wanted to demonstrate two principles in the design and development of the Scheme Pet Store. First, I wanted to show that the SISCweb model of development could be effective in a real-life web application. Second, I wanted to demonstrate that the Scheme Pet Store could be developed by a multi-disciplinary team via the concept of separation of concerns.

3.1 The SISCweb Way

SISCweb is an application framework that follows in the tradition of continuation-based servers [19, 9, 13, 12]. The emphasis of a continuation-based server is that a developer should be able to author a web application without having to worry about the typical details involved in writing one. For example, rather than providing a new API to access a user's session, it simply obviates the need to think in terms of a session altogether. In general, one does not need to think in terms of pages, but can think in terms of functions. The elegance of SISCweb is not in the constructs that it adds for writing a web application, but in the burden it removes by taking away the need for the programmer to concern himself with web-specific details. Perhaps the real power of SISCweb is not so much in how it fits the needs of the Scheme Pet Store, but in how it can be adapated to fit whatever vision a programmer wishes to impose upon it.

While the theory behind continuation-based servers leads one to believe that this is an ideal framework in which to build web applications, it was my hope that the Scheme Pet Store would concretely confirm this hypothesis.

3.2 Putting the SISCweb Way to Work

Using the SISCweb framework to develop the Scheme Pet Store was relatively straightforward. I found that each concept of the application comfortably mapped to a single function. In general, each of these top-level functions took in the user's current state as well as any arguments that made sense for that particular process. It was also the responsibility of these top-level functions to produce well-formed SXML [14]. As will be discussed below, the SXML returned was not markup oriented, but was domain specific to the function.

With these relatively simple rules in place, SISCweb took care of the details of having the different top-level functions call each other as well as package up arguments and return back the appropriate SXML. One of the joys of SISCweb is that it automatically produces valid URIs which dispatch to the correct functions. I did not need to write my own dispatcher or maintain a global XML file which described how the dispatching would take place. This was yet another example of SISCweb allowing me to think in terms of functions instead of low-level web details. These top-level functions are regular Scheme, so they can invoke other modules as well as arbitrary Java code.

One interesting side effect that I noted from this design was that the top-level functions could be set up to return arbitrary SXML. This SXML could then be used by other functions in a non-web context or simply executed and visually inspected. The end result was that top-level functions could easily be reused for nonweb-related processes.

Consider the example of the `view-shopping-cart` function. This function takes in a single argument (the user's current state) and returns SXML as in the example below:

```
(cart (item (price 3.45) (description "Foo") (quantity 3))
      (item (price 9.99) (description "Bar") (quantity 1)))
```

3.3 SISCweb in Action

To further appreciate how concepts described in section 3.2 actually play out, consider the code below. It compares how a typical JSP [16] style application would dispatch the user to view a specific pet versus how it can be done in SISCweb.

```
JSP:
 <a href='viewPet.do?petId=<%= petId %>'>View <%= petName %></a>
Output:
 <a href='viewPet.do?petId=cat003'>View Fluffy Cat</a>
```

The code below shows an example of how it might be done in SISCweb:

```
SXML:
 `(a (@ (href-p ,(lambda (req) (view-pet pet-id))))
     "View " ,pet-name)
Output:
 <a href='store;61712d693142645f47223d20126b4a0473127911'>View Fluffy
 Cat</a>
```

The above code makes use of a special feature in SISCweb that allows closures to be attached to HTML tags. In the above case, when "View *pet name*" link is clicked, it executes the closure which in turn calls `view-pet`.

SISCweb provides two important benefits that are demonstrated in the code above. First, SISCweb allows you to make use of lexically scoped variables (such as `pet-id`) rather than forcing the user to use `GET` style arguments. This is both a convenience to the programmer and provides a higher degree of security. The security is provided by the fact that the website visitor does not explicitly see the arguments being passed in the links and is unable to perform forced browsing [20].

Secondly, most traditional continuation-based web frameworks have the programmer think in terms of the current-continuation. SISCweb, however, frees the programmer from this and allows him to simply attach arbitrary code where he sees fit. This is a significant improvement as the programmer no longer needs to manually manage multiple continuations. See [13] for further discussion on this topic.

3.4 Separations of Concerns

The second key goal I had for the Scheme Pet Store was demonstrating Scheme's ability to participate in multi-developer projects. I wanted to show that a DBA could develop database queries that an application programmer could use or that a graphic designer could give the application a face lift without affecting the application programmer's logic.

I accomplished this goal by viewing key parts of the system as being split into producer and consumer concerns and allowing them to be separately maintained. For example, a producer concern for the query system may be the actual SQL statement used, while a consumer concern may simply be the ability to execute that query and review the results. Some of the areas that I focused on providing a separation of concerns for include database queries, layout, web forms and navigation.

The sections below describe some of the specific approaches I took in separating concerns. When developing a separation of concerns, I chose to maximize flexibility for the producer and simplicity for the consumer.

The query system provides the basis for a good example. The producer can specify arbitrarily complex queries that take in any number of arguments. The consumer, however, needs to know only the name of the query, what arguments it expects and what the return values will be. The consumer can remain ignorant in the details of the query and the producer can remain oblivious to where the query is used.

While developing the Scheme Pet Store, I played the roles of the application programmer, DBA and designer. However, I still found the separation of concerns mechanism to be quite useful. I could put a simple query in place, make use of it, and then go back and improve it without disturbing the application.

The SISCweb environment itself helps to facilitate the separation of concerns by allowing for the redefinition of code on the fly. This allows any of the involved developers to make changes and immediately see the effects .

4 Examples of Scheme Modules

In this section, I describe some specific modules used in the Scheme Pet Store. This section will demonstrate some of the key decisions I made, as well as give some valuable clues to those who are interested in reading the Scheme Pet Store source code.

One common theme that will quickly become clear is the role pre-exisiting 3rd party libraries played in the development of the Scheme Pet Store. The majority of the difficult work usually involved in developing a web application had already been addressed. For example, Java provides JDBC [15] as a method for accessing a database, and SXML provides a standard for representing XML in Scheme. In many cases these technologies were simply wrapped up in a module to expose them in such a way that the above-mentioned goals could be met. This has the additional benefit of making a large amount of the code in the Scheme Pet Store non-controversial and well tested.

4.1 Layout

I have noted how SISCweb gives programmers the opportunity to develop web applications without having to worry about many of the details of the web. While that is true, one issue that must always be kept in mind is that in the end, a browser needs to render the output of our functions. In this section, I describe how I developed a separations of concerns framework for handling the layout of the Scheme Pet Store.

Scheme's ability to support XML is remarkable. In fact, the ability to easily generate and manipulate XML and HTML is just as impressive as SISCweb's magic handling of function calls. There has been quite a bit of work done in the Scheme community to map XML to Scheme and vice versa. SISCweb made use of the SXML representation which is one of the more common approaches. Consider how the following is represented in both XML and Scheme:

```
<cart count="1">
 <item>
  <name>Cat</name>
  <price>3.94</price>
  </quantity>3</quantity>
 </item>
</cart>
```

```
(cart (@ (count "1"))
      (item
      (name "Cat")
      (price "3.94")
      (quantity "3")))
```

SISCweb extends the standard SXML format by allowing numbers to be valid content. In the Scheme Pet Store this approach has been extended even further, allowing arbitrary Scheme values to appear in the XML tree. Consider how a form might be represented in Scheme versus XML:

```
<form action="something.do">
 <field name="q" size="20"/>
 <submit/>
</form>
```

```
‘(form (@ (action ,(lambda () ...)))
       (field (@ (name "q") (size 20)))
       (submit))
```

In the above case, rather than naming an action associated with the form, the implementation of the action is directly specified as a closure.

In early iterations of the Scheme Pet Store, each function was responsible for outputting well-formed HTML. Initially, this approach held some promise because of the ease in which Scheme allows you to generate HTML. Consider the example below:

```
(page "Search for a Pet"
      `(ul
         ,(map (lambda (found)
                 `(li ,found))
               (search-for-pets query))))
```

In the above case the function `page` takes in two arguments, a title and the body of the page. The second argument to the page function is well-formed HTML, which makes use of quasiquote. This expands to the following HTML:

```
<html>
 <head>
  <title>Search for a Pet</title>
 </head>
 <body>
  <ul>
   <li>Saltwater fish</li>
   <li>Freshwater fish</li>
  </ul>
 </body>
</html>
```

The main issue with this approach is that it doesn't allow for separation of concerns. The application developer, rather than a designer, would be responsible for choosing the HTML tags used to render the page. In an ideal world, the application programmer could return a value from the function which contained the data to be rendered, but would not concern himself with the actual rendering instructions. Using Scheme, I was able to accomplish this. This allows the application programmer to play the role of the consumer while the designer acts as producer.

The package I developed is named `domain-ml`, which stands for domain markup language. The approach I had in mind was that functions would return XML that made sense to themeselves. The results, therefore, would be domain specific. For example, a domain specific representation of the above may be:

```
`(query-results
  ,(map (lambda (found)
          `(row ,found))
        (search-for-pets query)))
```

When a domain-ml fragment needs to be rendered, a lookup function is located and applied. This function produces the appropriate HTML to be sent to the browser. The transformation function is defined separately from its use. Thus, our separation of concerns is accomplished. The application programmer can think in terms of a domain specific markup language while the designer simply needs to construct a function to convert the domain-ml into HTML.

In the Scheme Pet Store, the transformation of domain-ml into HTML is accomplished by using the `sxml-match` package [7]. This package allows the programmer to trivially capture patterns and re-write SXML into a new flavor of SXML. It performs the same function as XSLT but is more brief. The `sxml-match` makes use of the same ellipses construct that `define-syntax` does. Using the ellipses notation one can easily decompose and compose XML as well as match complex patterns.

Consider the following conversion function, which converts our `query-results` into valid HTML.

```
(define (query-results->html doc)
  (sxml-match doc
```

```
   ((query-results (row ,value) ...)
    (page "Query Results"
          `(ul
             (li ,value) ...)))))
```

In the above case `query-results` is converted into a `ul` tag with nested `li` tags.

While I found the use of `sxml-match` to be an example of Scheme almost reading my mind, there may be other programmers who do not like this notation. If that's the case, all one needs to do is to redefine the implementation of the translation functions, an activity that can be done without affecting the functionality of the system.

It is recommended that one become familiar with both SXML and `sxml-match` before attempting to understand the Scheme Pet Store.

4.2 Queries

One of the most common requirements of a web application is to communicate with a relational database. Because the Scheme Pet Store is written in SISC, it can easily take advantage of the JDBC API that is standard with Java. For connection pooling, I made use of the Apache Commons DBCP package [18]. For an actual database implementation, I chose PostgreSQL [5], though I could have chosen any database vendor which provides a JDBC interface.

While JDBC accomplishes the difficult work of communicating with a database implementation, its interface from Scheme is not particularly friendly. SISCweb helps to correct this by putting a Scheme-oriented front end on JDBC by providing a `sql` module. The SISCweb API is a collection of procedures and macros that make using JDBC much simpler.

Much like my original approach to the Pet Store involved having functions work directly with HTML, I also had functions embed SQL directly. It did not take long for me to realize how poor a choice this was, as it was a clear violation of the separation of concerns principle I was striving for.

The approach I settled on is very similar to how I handled the issues of layouts described above. At the highest level, queries are registered by name with the system. The query author is the producer in this situation, and as such is responsible for all the details of the query. This includes the actual SQL statement to use, the specification of the returned columns and a conversion function to turn user supplied arguments into SQL values. The consumer in this scenario needs to know the same information that would be required if he wanted to invoke a function created by the producer. This includes the name of the query, the arguments and the expected return values.

From the consumer's view, the query framework is made up of a series of macros. I chose to use macros, instead of functions, because they allowed me to capture patterns that functions could not. This was very much inspired from SISCweb's query framework which also makes use of macros. Consider the following query:

```
(map-query ctx find-products-by-category-name
           (category)
           (product-id name)
  (display (format "I found: ~a (~a)" name product-id))
  `(row ,product-id ,name))
```

The above code executes the query named `find-products-by-category-name`. The first list after the name is the arguments to invoke the query with. In this case we have a single argument, `category`. The next list is a list of variables which will be bound to result set columns. In this case, the result set has two variables available, `product-id` and `name`. The rest of the statements are simply executed for every row returned in the query, and make use of the variables `product-id` and `name`.

The variable named `ctx` is not specific to the `map-query` macro. Instead, it is used throughout the application to represent the current user's state. The `ctx` can be thought of as a nested set of hashtable-like structures, which can contain such items as the current shopping cart contents and error messages that have been queued up for the user.

Not only does the above query hide the details of the SQL to be run, but it also hides details such as the names of the columns that are returned, and how to convert `category` into an argument to be handed to the query. The `map-query` macro also takes care of destructuring each row's values from a result set into scheme variables, so the programmer doesn't have to. This is not only convenient, but saves the programmer from a class of errors.

Along with the `map-query` function, a `let-query` function is provided. This behaves like a `let` statement, and pulls a single row from the database to initialize a set of variables.

4.3 Forms

HTML Forms were another aspect of the Scheme Pet Store that I felt cried out for a framework. I wanted to make HTML forms a resource that could be used by name, just like a query or a layout. The producer of a form would specify the exact fields, the validation and any conversion functions. The consumer of a form would simply look up the form by name, and provide functions to be invoked when the form succeeds or fails.

Using this strategy, the many details of exactly what fields to collect, how they should be rendered and how they should be validated, can all be handled behind the scenes. Consider how one might use a search form:

```
(use-form ctx 'search
          (lambda (ctx query)
            (run-search query))
          (lambda (ctx query)
            (show-oops-page
               (format "The query: ~a is invalid" query)))
          "")
```

The look and feel, as well as what type of validation will take place, are all details the consumer need not concern himself with. However, the consumer is required to provide a success and failure function. These functions will be invoked with the form values as their arguments, effectively destructuring the form request object into local variables. In other words, the form above had a single input box which was being stored in the `query` parameter. If the form had two arguments, then the success and failure functions would take in two arguments in addition to the context variable.

The above strategy makes use of Continuation Passing Style (CPS). While CPS programming can often be very difficult to follow, I found the above strategy to be quite clean. When compared to a framework such as struts, I find the above to be easier to trace. The basic question, "what happens when I hit this submit button," is answered just by looking at the place the form is used, and not by looking up some name in a lengthy XML configuration file. This provides a degree of locality to forms which makes them easier to reason about.

The CPS approach of handing in both success and failure functions makes it so that you can't cheat by not handling the case of form failure. Below is another example usage of a form:

```
(define (ask-for-birthday ctx month year)
  (use-form ctx 'ask-for-birthday
            (lambda (ctx month year)
               (store-data month year)
               (say-thanks))
            (lambda (ctx month year)
```

```
            (ask-for-birthday ctx month year))
          month year))
```

In this scenario, if we succeed in asking for a birthday, we run a function that stores and continues the transaction. However, if we do not have a valid birth date, then we invoke the `ask-for-birthday` function again. In this case we assume that the validation code has added error messages explaining why the birthday is invalid.

The producer side of the form framework is fairly straightforward. The author of the form makes use of the layout framework described above for rendering purposes.

The validation framework, at the lowest level, makes use of CPS much like the consumer side of the framework does. In general, a validator is handed a success and failure function. If the validation passes, success should be called; if the form fails to validate, the failure function should be invoked.

I implemented a set of high-level validators so that the details of continuations would be hidden. An unexpected benefit of making use of continuations is that I was able to change the behavior of the validators without changing their implementation.

For example, the typical implementation of the failure function is to stop validation and call the form's failure function. I set up a validator named `try-all` that invokes a series of validators, but sends in a different implementation of the failure function. In `try-all`'s case, when the failure-function is invoked, it continues to invoke the rest of its validators. However, it insures that any future call to success or failure will result in the failure of the form. The end result is that all the validators are tried, and may possibly succeed or fail, showing the user a collection of errors messages, not just the first error.

Consider the following validation example:

```
(list (try-all (field-is-a-number? "month" "Birthday Month")
               (field-is-a-number? "year" "Birthday Year"))
      (fields-validate '("month" "year")
                       (lambda (ctx month year)
                         (birthday-sanity-check month year))))
```

In this example, we have provided a list of validators. We have three criteria we would like to check, mainly that the `month` and `year` fields contain numbers and that the function `birthday-sanity-check` passes. I have grouped the first two checks together with the `try-all` so that both checks will be conducted before giving up and failing the form. If the user were to leave the form totally blank, he would see an error about both the month and year not being filled in. The third validator demonstrates the ability to run arbitrary checks for a form.

As mentioned above, the form framework not only validates arguments, but also has the opportunity to convert raw form values into more sophisticated data structures. I made use of this functionality while dealing with a shipping address form. The shipping form has quite a few individual fields, though they can be thought of as one `shipping-address` structure. The form framework took responsibility for collecting all of these individual fields and stored them in the `shipping-address` structure. This simplified the success function for the form because it took in a shipping-address object and processed it, rather than having to deal with many raw values.

5 Gotchas

There are many parts of the Scheme Pet Store which I am quite impressed with. In general, SISC and SISCweb are terrific environments in which to produce applications. However, it is important to outline several areas where I ran into issues. I look forward to getting feedback from the Scheme community on how to address them.

5.1 Performance

I did not delve into the areas of performance and scalability. I have no evidence for how the Scheme Pet Store's performance compares to the Java Pet Store or to any of the other implementations. I hope to explore and understand these metrics at a later time.

5.2 Serializing Records

During the development of the Scheme Pet Store, I ran into a well known bug in SISC that made the serializing of records problematic [6]. As a result, I was unable to take advantage of SISCweb's capability of storing continuation data in a relational database and had to make use of the simple HTTP session style storage. This provided an acceptable workaround solution for the short term. Longer term, I would imagine that either the SISC bug will be fixed or a solution that does not involve SISC records will be crafted.

5.3 Deep-linking

As mentioned above, the Scheme Pet Store is implemented as a single web application. All links and forms have URLs that are nonsensical from the user's perspective. They are simply a resource that SISCweb manages, much like a garbage collector manages memory addresses. This is a nice feature from the programmer's perspective, and improves security to some degree. However, this turns out to be an issue when someone either bookmarks a link or tries to send the link to a friend.

Imagine a scenario where a user is browsing the application and notices a specific pet a friend would be interested in. The user cuts and pastes the URL into a mail note, only to have his friend unable to access the URL because the URL contains information visible only to the sender. This problem is part of the larger issue of deep-linking. This is not only a problem in the SISCweb world, both Flash and many AJAX applications suffer from this drawback [10].

As a stop gap measure, I have taken a similar approach to solving the problem that Google Maps [2] takes. Because their application is AJAX-based, the map you are looking at on the page may not correspond to the URL in the address bar. Their solution was to provide a "link to this page" button. When this button is pressed, the page reloads, and places the URL that currently represents this map in the address bar. In the case of the Scheme Pet Store, pages that can be linked to directly have a "show link to this page" button, that when pressed, shows a full URL to the page. This URL can then be thought of as a direct link to that page.

While I have implemented a workaround, I'm hardly convinced that the deep linking problem is solved. There are still questions that have not been fully explored about bookmarking links and how long a user's state should be available for them to resume.

6 Next Steps

The Scheme Pet Store currently represents a proof of concept. I feel as though it demonstrates some of the areas in which Scheme can shine as a web development environment. In general, I'm very pleased with my experiment and feel as though my hunch that SISCweb is a uniquely powerful tool is accurate.

The next level to take the Scheme Pet Store to is to turn it into a reference application. This was one of SUN's primary goals for producing the Java Pet Store, and would no doubt be of huge benefit to the Scheme community. In order to accomplish this goal, the Scheme and programming languages community need to provide feedback on the Pet Store. What patterns and concepts can be introduced, improved upon, and most importantly, removed?

Once input has been provided, I can imagine developing improved versions of the application, as well as conducting neccessary performance benchmarks.

Please contact me at `benjisimon@gmail.com` or visit `http://schemepetstore.pbwiki.com` if you are interested in learning more about the Scheme Pet Store, have any feedback, or need help running this application.

7 Thanks

This project would not be possible without Alessandro Colomba, the author of SISCweb, or Scott Miller and Matthias Radestock the authors of SISC. Additionally, the sisc-users mailing list provided both design recommendations as well as implementation assistance.

References

[1] Enterprise: Java Pet Store Sample Application. http://java.sun.com/blueprints/code/index.html.

[2] Google Maps. http://maps.google.com.

[3] Microsoft .NET Pet Shop. http://www.gotdotnet.com/team/compare/petshop.aspx.

[4] Pet Market 1.2: Rich Internet Application. http://www.macromedia.com/devnet/blueprint/.

[5] PostgreSQL. http://www.postgresql.org/.

[6] Records deserializing incorrectly (bug 806864). http://sourceforge.net/tracker/index.php?func=detail&aid=806864&group_id=23735&atid=379534.

[7] Jim Bender. sxml-match: Pattern Matching of SXML. http://celtic.benderweb.net/sxml-match/.

[8] Tim Buntel. Another Day, Another Pet Market: Why We Built an HTML Version of the Blueprint Application in ColdFusion MX. http://www.macromedia.com/devnet/coldfusion/articles/petmarket.html.

[9] Mike Burns, Gereg Pettyjohn, and Jay McCarthy. *Web Server Manual*. PLT Scheme, http://download.plt-scheme.org/doc/299.400/html/web-server/.

[10] Christian Cantrell. Deep Linking in Flash and AJAX Applications. http://weblogs.macromedia.com/cantrell/archives/2005/06/deep_linking_in.cfm, June 2005.

[11] Alessandro Colomba. SISCweb. http://siscweb.sourceforge.net.

[12] Chris Double. Modal Web Server Example. http://www.double.co.nz/scheme/modal-web-server.html.

[13] Peter Walton Hopkins. Enabling Complex User Interfaces In Web Applications With send/suspend/dispatch, November 2003.

[14] Oleg Kiselyov. SXML. http://okmij.org/ftp/Scheme/SXML.html.

[15] SUN Microsystems. Java Database Connectivity. http://java.sun.com/products/jdbc/.

[16] SUN Microsystems. JavaServer Pages Technology. http://java.sun.com/products/jsp/.

[17] Scott G. Miller and Mathhias Radestock. SISC, Second Interpreter of Scheme Code. http://sisc.sourceforge.net.

[18] Apache Jakarta Project. Commons DBCP. http://jakarta.apache.org/commons/dbcp/index.html.

[19] Christian Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. Technical Report 7, LIP6, May 2001.

[20] Wikipedia. Forceful Browsing. http://en.wikipedia.org/wiki/Forceful_browsing.