Nevow: A Web Application Construction Kit

Summary

Nevow is a web application construction kit, based on the ideas developed in the Twisted Woven package. Its main focus is on separating the HTML template from both the business logic and the display logic, while allowing the programmer to write pure Python code as much as possible. It has various parts which can be used individually or as a whole, integrated web solution:

- XHTML templates: contain no programming logic, only nodes tagged with special nevow attributes
- data/render methods: simplified MVC
- stan: An s-expression-like syntax for expressing xml in pure python
- formless: For describing the types of objects which may be passed to methods of your classes, validating and coercing string input from either web or command-line sources, and calling your methods automatically once validation passes
- freeform: For rendering web forms based on formless type descriptions, accepting form posts and passing them to formless validators, and rendering error forms in the event validation fails
- livepage: Cross-browser JavaScript glue for sending client side events to the server and server side events to the client after the page has loaded, without causing the entire page to refresh

Dependencies

Before you begin, make sure you acquire the dependencies:

- http://twistedmatrix.com/products/download
- http://nevow.com/nevow-0.1.tar.gz

Then get the examples:

- http://nevow.com/Nevow2004Examples.zip

You might also be interested in this tutorial in presentation format (PDF):

- http://nevow.com/Nevow2004.pdf

Why Nevow?

Nevow was created to help the author build several very large, complicated, web applications. It builds on many ideas the author was exposed to while using various other web frameworks, and attempts to borrow useful concepts from each while still remaining lightweight and as unobtrusive as possible.

Nevow's templating semantics use ideas from various other templating languages, including Quixote, ZPT, and XMLC (from the Enhydra project). It allows for the use of strict, validating XHTML templates tagged with special attributes called 'directives', similar to ZPT.

It allows programmers to render HTML views in pure Python instead of forcing them to learn a new language which is embedded in HTML, similar to Quixote. Finally, it uses a DOM, or Document Object Model, to represent the document in memory during the rendering process, similar to XMLC.

Nevow also contains an Object Publishing framework similar to the ideas present in twisted.web and Zope. Each URL is mapped to an instance of the Page class using a simple traversal interface. An application server and HTTP server built using twisted.web is included which can get you started very quickly.

Nevow is built to be embedded in an environment which uses asynchronous I/O, such as twisted.web. It builds a stack of 'context' objects which it uses to keep track of state during the rendering process, and uses this to provide support for "pausing" rendering and resuming when an I/O operation has completed.

Nevow Tutorial

The rest of this paper will step you through building a web application using Nevow. The application we have chosen to build is a simple Calendar and Scheduling application, which can render a calendar for any month and year and integrates with a third party iCalendar parsing module to populate the calendar with events. To see the final result, change to the Example4 directory and run the following command:

twistd -noy sched.tac

Then, navigate to the following URL using the web browser of your choice:

http://localhost:8080/

You should then see a calendar for the current month. You should be able to choose any month to display, click on an events link to see the events for that day, and add an event to any day. Now, we will step through the process of building this application, introducing elements of nevow in the process.

HTML based templates

Nevow includes the ability to load HTML templates off disk. These templates may have processing directives which cause the execution of python methods at render time. The attribute technique was inspired by the attributes used by ZPT. However, no actual code may be embedded in the HTML template.

A *Directive* is a command to Nevow to invoke some Python code. Nevow uses *data directives* and *render directives*. The presence of a nevow:data or nevow:render attribute on an HTML node in a template will cause the related data_* or render_* method to be located and invoked.

A *Pattern* is a node which Python code can locate by name. When a node has a nevow:pattern attribute, Python code can use the context methods 'onePattern(name)', 'allPatterns(name)', and 'patternGenerator(name)' to locate and clone that pattern node.

A *Slot* is a fake node which Python code can replace with real content. When a node has a nevow:slot attribute, Python code can use the context method 'fillSlot(name, value)' to cause a slot to replace itself with the given value when the slot is rendered.

The first HTML template we will use in our Schedule example is a full HTML document which when viewed in a browser will show an approximation of what the final application will look like, with dummy data in place. Open the Example1/Month.html file in a browser and observe how it gives a reasonable preview of the final application. A designer may open this template in a graphical HTML editor and style it, and the directives we have placed in the document should be preserved.

Let's take a closer look at the directives, patterns, and slots present in the template:

```
<span nevow:data="currentMonth" nevow:render="month">
  <h1> <nevow:slot name="label"> The label goes here </nevow:slot> </h1>
  <table height="50%" width="50%" border="1">
    <tr>
      <td>Sunday</td>
      ...
      <td>Saturday</td>
    </tr>
    <tr nevow:pattern="calendarWeek" nevow:render="remove">
      <td nevow:pattern="calendarDay" align="center"></td>
      ...
    </tr>
    <nevow:slot name="calendarBody" />
  ...
```

In this case, we have a dummy node around the interesting portion of the template. This span node will not affect the presentation of the document, and is merely a carrier for some directives which will affect some nodes inside of the span. On this node, we see two directives: nevow:data="currentMonth" and nevow:render="month". When nevow begins to render this span node, it will first invoke a method named data_currentMonth, and pass the resulting data to a method named render_month. The return value of the render_month method will replace the entire span in the rendering DOM and further processing will take place on it.

Inside of the span node, there are two slots: *<nevow:slot name="label">* and *<nevow:slot name="calendarBody" />*. render_currentMonth will be sure to provide values to fill these slots before returning.

Finally, inside of the span node there are two patterns: *nevow:pattern="calendarWeek"* and *nevow:pattern="calendarDay"*. The render_currentMonth method will make clones of these pattern nodes as it renders the calendar.

One last note: The <tr *nevow:pattern="calendarWeek"*> pattern node has a nevow:render="remove" directive. Nevow does not currently automatically remove pattern nodes from the template, so to prevent the dummy data from ending up in the output we use the render_remove method. render_remove is defined on the Page class, and simply returns ".

data/render methods

When nevow locates a *data directive* in a template, it calls the associated data_* method on the Page class. In our example, the data method named data_currentMonth is invoked. The return value from the data method is remembered using the current context object and is passed as the 'data' parameter during the rendering of that node.

Similarly, a *render directive* in a template will invoke an appropriately named render_* method. In the example, the 'render_month' method will be invoked. The data will be located by looking in the current context object.

To tie our HTML template to our implementations of our data_* and render_* methods, we subclass the Page class. The skeleton implementation looks like this:

```
class ScheduleRoot(rend.Page):
    docFactory = rend.htmlfile('Month.html')

    def data_currentMonth(self, context, data):
        ...

    def render_month(self, context, data):
        ...
```

The signature of a data_* or render_* method is always (self, context, data). Self is the Page instance itself, context is the Context instance currently on top of the context stack, and data is the most recent data placed on the context stack.

In this case, our data_currentMonth method is going to produce some data for use during the rendering process by our render_month method. By default data_currentMonth will return a tuple of the year, month, and a list of lists of weeks and days in that month. However, we are also going to parameterize the data method so that the month and year to return can be specified as arguments in the URL:

```
def data_currentMonth(self, context, data):
    curtime = time.localtime()
    # We either get the year from the request, or if there is no
    # year argument in the request, use the current year
    year = int(context.arg('year', curtime[0]))
    # We either get the month from the request or the current month
    month = int(context.arg('month', curtime[1]))

    # Return a 3-tuple of the year, the month, and the list of lists
    # of the weeks and days
    return year, month, calendar.monthcalendar(year, month)
```

Our render_month method is going to use the data produced by our data method, as well as APIs available from the context object, to modify the render context. It will make copies of *pattern nodes* which it locates in the HTML template by name, populate these pattern nodes with data, and insert the final DOM construct into the final page using the context.fillSlots method:

```
def render_month(self, context, data):
    # Unpack our 3-tuple
    year, month, weeksAndDays = data

    weekPattern = context.patternGenerator('calendarWeek')
    dayPattern = context.with(weekPattern()).patternGenerator('calendarDay')

    calendarBody = []
    for week in weeksAndDays:
        currentWeek = weekPattern().clear()
        calendarBody.append(currentWeek)
        for day in week:
            currentDay = dayPattern().clear()
            if day != 0:
                currentDay.children.append(str(day))
            currentWeek.children.append(currentDay)

    context.fillSlots('label', "%s %s" % (calendar.month_name[month], year))
    context.fillSlots('calendarBody', calendarBody)
    return context.tag
```

Notice how this method, which does some fairly complicated view generation, is easy to read since it is pure python, and does not contain any knowledge of the layout or format of the HTML template since it uses named patterns and slots to manipulate it.

We have now completed the initial Example1 application. Change to the Example1 directory and start the application:

```
twistd -noy sched.tac
```

Visit the application on the URL:

http://localhost:8080/

You will see an accurate rendering of the current month's calendar. The application also contains a view which renders several years worth of links which you can click on to see the corresponding month.

Integrating with an External Data Source

Nevow was designed to easily allow you to provide a web ui for a third-party piece of code without modifying the original code. Nevow makes heavy use of *adapters* and *interfaces* to provide a *component-based architecture* where classes can provide implementations of aspects of functionality of another class.

However, it is not necessary to understand components in order to be able to use Nevow effectively. In our example, we are going to show a more explicit example of wrapping an external data source and exposing it to the web.

In order to support our Scheduling example, I did a web search for a python module which could parse the industry standard iCalendar format and present some simple Python objects. I located a module which was specifically designed for parsing the data files used by Apple's iCal application. With a few small modifications I generalized it to be more crossplatform. The original module can be found here:

http://www.devoesquared.com/Software/iCal_Module

The modified module, along with a sample .ics file, can be found in the Example2 directory. To begin integrating with this external data source, we override the __init__ method on our ScheduleRoot class:

```
def __init__(self, calendarEntries):
    self.calendarEntries = calendarEntries
    super(ScheduleRoot, self).__init__()
```

Then, we modify the configuration file, which creates our ScheduleRoot instance and starts a web server, creating an ICalReader instance and passing it to ScheduleRoot:

```
import iCal, schedule
from nevow import appserver
from twisted.application import service, internet

application = service.Application('Schedule')
webservice = internet.TCPServer(
    8080,
    appserver.NevowSite(schedule.ScheduleRoot(iCal.ICalReader()))
)
webservice.setServiceParent(application)
```

This configuration file, sched.tac, is a standard twisted application configuration file. When executed using 'twistd -y', the Application instance assigned to the 'application' variable will be started and will listen on any configured ports.

Finally, we modify our 'render_month' method to take this additional data source into account when rendering the page:

```
calendarBody = []
for week in weeksAndDays:
    currentWeek = weekPattern().clear()
    calendarBody.append(currentWeek)
    for day in week:
        if day != 0:
            events = self.calendarEntries.eventsFor(date(year, month, day))
        else:
            events = []
        currentDay = dayPattern(
            render=self.render_day, data=(year, month, day, events))
        currentWeek.children.append(currentDay)
```

In this example, we are calling the 'eventsFor' api on our ICalReader instance to discover if there are any events for the day we are about to render:

```
events = self.calendarEntries.eventsFor(date(year, month, day))
```

Then we *delegate* the rendering of the actual day fragment to the render_day method, passing it a tuple of (year, month, day, events) as data:

```
currentDay = dayPattern(
   render=self.render_day, data=(year, month, day, events))
```

Setting the renderer and the data on a node using Python code is similar to what happens when Nevow encounters a *render directive* or a *data directive* in an HTML template, except we are able to pass Python references more directly. When control returns from the render_month method and Nevow continues to render the DOM, it will encounter these nodes with additional processing directives on them and render them accordingly.

Generating URLs using the url module

We are going to set up our web application so that day detail pages are available at URLs in the format http://localhost:8080/2004/3/21. We will implement render_day such that it renders a link to a day detail page when a day contains events. To do this, we will use the url module. The url module contains several useful constructs, including a class which contains APIs for generating links programatically, and several useful instances of this class. In this case, we are going to use the 'here' instance, which represents the URL of the page which is currently being rendered:

```
def render_day(self, context, data):
   year, month, day, events = data
   if events:
      from nevow.url import here
      # Construct URL to child page
      url = here.child(str(year)).child(str(month)).child(str(day))
      eventsDOM = context.onePattern('events').clear()
      eventsDOM.attributes['href'] = url
      eventsDOM.children.append(
         "%s event%s" % (len(events), len(events) > 1 and 's' or ''))
   else:
      eventsDOM = ''

   context.fillSlots('date', day)
   context.fillSlots('events', eventsDOM)
   # The return value of a render method replaces the template node
   # We wish to use the template node after modifying the context
   return context.tag
```

We have now completed Example2. Changing to the Example2 directory, running twistd, and visiting the application's URL will present us with a modified calendar which renders an informative link if there are any events on any particular day. However, clicking on a link will result in a 404 error page, because we have not yet implemented the day detail page.

URL Traversal and Object Publishing

Nevow uses the concept of Object Publishing. Object Publishing is a web application construction methodology which dictates that every URL presented by a web application is published, or rendered, by an instance of a Python object. In Nevow's case, every URL is rendered by a subclass of the Page class. In our Schedule example, we have so far had one instance of the Page class, the *root Page instance*. In order to add more web pages to our application, we need to implement one of Nevow's *URL Traversal* APIs.

Nevow has one required URL Traversal API, locateChild. locateChild is defined as the following:

```
def locateChild(self, request, segments):
    """Locate another object which can be adapted to IResource given the tuple 'segments'
    Return a tuple of resource, path segments
    """
```

The Page class defines a default implementation of locateChild which provides additional convenient ways of specifying children: child_* methods and getDynamicChild. However, in this tutorial we are simply going to focus on locateChild briefly.

Example3 contains an implementation of locateChild on our ScheduleRoot class which prevents the day detail URLs from resulting in a 404 page. Since our earlier URL-generation code generated URLs of the form http://localhost:8080/2004/3/21 we attempt to map the segments ('2004', '3', '21') to year, month, and date integers. A request for any other URL, excluding http://localhost:8080/, results in a 404 Not Found page:

```
def locateChild(self, request, childSegments):
    # Handle the URL http://localhost:8080/
    if childSegments == ('',):
        return self, ()
    try:
        year, month, date = map(int, childSegments)
        import day
        return day.DayDetail(self.calendarEntries, year, month, date), ()
    except ValueError:
        # If the url doesn't consist of a tuple of year, month, day, or the segments
        # are not integers, then we render a 404 page.
        return rend.NotFound
```

In each case, we return a tuple of a Page instance and a tuple of the remaining, unhandled segments. All of the cases shown here return an empty tuple as the unhandled segments, indicating that the Page instance being returned is the Page instance which will be responsible for rendering this URL.

(Note: rend.NotFound is a tuple of a default Page instance which renders a 404, and an empty tuple)

Next, we are going to implement the DayDetail page subclass using a different technique than before.

Stan

One of the most powerful things about nevow is stan, an s-expression-like syntax for producing XML fragments in pure Python syntax. Stan is not required for using nevow, but it is both a simple and powerful way to both lay out your XHTML templates and express your display logic. Stan is merely a lightweight DOM built using basic Python types, such as strings, lists, and generators. It was created because the W3C DOM, which was used by Woven to represent the page as it was being rendered, was too cumbersome, heavyweight and slow to be truely useful. At the most basic level, Nevow's use of stan means you can write your render methods very consisely and conveniently:

```
def render_date(self, context, data):
   return "%s %s, %s" % (calendar.month_name[self.month], self.date, self.year)
```

In this case, we simply return a Python string as our view rendering logic. We could also return a list, tuple, integer, or our function could even be a generator. In fact, almost all basic Python types are supported (except dictionaries, which do not have a natural display order).

Stan also defines a Tag class. Tag instances have a 'tagName' string, an 'attributes' dictionary, and a 'children' list. Instead of providing baroque and cumbersome node-access APIs like the W3C dom, programmers simply use the normal python APIs for changing these attributes.

Nevow also ships with the 'tags' module, a module which defines instances of the special Proto class for every tag name defined by the XHTML 1.0 specification. A Proto is simply a Tag factory which sets the tagName attribute of the Tag instance.

However, the real utility of stan lies in how it overrides the special methods __call__ and __getitem__ on both the Proto and Tag classes. The __call__ implementation takes the keyword arguments passed to the call and updates the 'attributes' dictionary, and the __getitem__ implementation makes sure the object passed is a sequence and then calls extend on the 'children' list.

If you choose to use the stan Tag syntax, you will find that it is a readable, fast, and consise way to rapidly prototype HTML layouts. For example:

```
from nevow import tags as T

class DayDetail(rend.Page):
   def render_events(self, context, data):
      events = self.calendarEntries.eventsFor(
         date(self.year, self.month, self.date))
      if not events: return "No events yet."
      return T.ol[
         [
            T.li[str(e)]
            for e in events
         ]
      ]
```

If the 'eventsFor' method returned a single event for the day we were rendering, 'Full Moon', the HTML output resulting from this stan expression being inserted into the DOM would look like this:

```
<ol>
   <li>Full Moon</li>
</ol>
```

The rend module also contains a stan document factory which you can use to rapidly prototype your Page templates using stan instead of HTML:

```
docFactory = rend.stan(
T.html[
   T.head[
      T.title[
         "Detail for ", render_date]],
   T.body[
      T.a(href=root)["Back"],
      T.h1[
         "Detail for ", render_date],
      T.h2[
         "Events:"],
      render_events]])
```

If your requirements do not include designer interaction, you can develop an entire application without writing a single line of HTML.

With the implementation of URL Traversal and the addition of the day module, our Example3 application is now capable of rendering a day detail page for any day. Run the example using twistd and try it out. The included .ics file contains events for various moon-phase days, such as full moon, new moon, etc.

Formless

Form posting is one of the most tedious parts of developing a web application. Rendering the required html, post location generation and handling, type coercions, error handling and message rendering, and finally invoking some action when done adds up to a lot of error prone code. Handling forms using a higher level API while still retaining enough flexibility to render the form how you would like turns out to be a very difficult challenge. Nevow contains a third-generation implementation of some concepts originally developed for Woven in the twisted.python.formmethod and twisted.web.woven.form modules. The formless and freeform modules separate the concerns of *type description* from *form rendering/handling*.

Python is dynamically typed, which means it has no built-in controls for enforcing the types of objects which are passed to your methods. This is great for programmers, but not necessarily great if you are going to be passing user-entered input to those methods. Formless is a simple way to describe the types of objects that can be passed to your methods, as well as coerce from string input to those types. Other code can then accept user input from a command line or from a web form, validate the input against the types described using formless, and call the method once validation has passed.

Example4 will use Formless to add type descriptions to our code so that a form post may be handled automatically. We will create a subclass of formless.TypedInterface whose sole purpose is to carry type information for later introspection during the form rendering/handling process:

```
class IEventsAddable(formless.TypedInterface):
    def addEvent(self, description=formless.Text()):
        """Add Event

        Add an event to this day.
        """
        pass
    addEvent = formless.autocallable(addEvent)
```

In this case, we are declaring that the 'description' parameter to the 'addEvent' method of an instance which declares it implements the 'IEventsAddable' interface should be of type 'Text'. The formless module contains many subclasses of 'Typed' such as Integer, Boolean, and Choice. Typed is designed to be easily subclassed and implemented so you may provide additional type coercion and validation logic.

In addition, the IEventsAddable.addEvent method is declared to be *autocallable*, which means that as long as all validation as specified by the types passes, Nevow is allowed to automatically call this method with the appropriately coerced arguments.

Next, we will implement the autocallable method in our DayDetail page to add an event to the current day. First, we declare that DayDetail implements IEventsAddable, and then we provide an implementation of addEvent:

```
class DayDetail(rend.Page):
    __implements__ = IEventsAddable, rend.Page.__implements__

    def addEvent(self, description):
        newEvent = iCal.ICalEvent()
        newEvent.summary = description
        newEvent.startDate = date(self.year, self.month, self.date)
        self.calendarEntries.events.append(newEvent)
```

Now all that is left is to create a properly-formatted form which, when posted, will call our addEvent method. The form post implementation in Nevow is designed to be easy to recreate by hand, in addition to the automatic form rendering capabilities:

```
<form action="freeform_post!!addEvent" method="POST">
    <input type="text" name="description" />
    <input type="submit" />
</form>
```

However, we won't actually use this form in Example4, because we can have a nicer form rendered automatically using freeform.

Freeform: Forms for Free

Freeform is a nevow module which will automatically render web forms and accept form posts based on types described using the classes in formless. To use this ability, call freeform.renderForms() and pass information about which forms you want rendered. Place the return value of this function in the DOM, either in a stan tree or as the return result of a render_* method. Passing nothing to the freeform.renderForms function instructs freeform to render all available forms:

```
docFactory = rend.stan(
T.html[
...
   T.body[
...
      T.p[
         freeform.renderForms()]]])
```

There are also functions to aid you in rendering custom forms. Each is able to render a small portion of the overall form required for posting to an autocallable method.

- binding
- action
- argument
- value
- error

We have now completed the tutorial scheduling application. Change into the Example4 directory, run the application with twistd, and visit it. You should be able to go to any day detail page and add an event to that day. Note that these events are only saved in memory and are not persisted to disk anywhere, so when you shut down the server they will be lost. The iCal module does not implement generation of .ics files or provide a convenient API for adding an event to an existing calendar, but it could be added.

Our final example application is a web-based chat application. It showcases an experimental Nevow technology called LivePage.

LivePage

LivePage is a Nevow technology which allows programmers to receive server-side notification of client-side JavaScript events, and to send JavaScript to the client in response to a server-side event. Woven included a fairly extensive implementation of LivePage, including publish-and-subscribe model changed notification with implicit view rerendering. Nevow has taken a different approach, attempting to keep the implementation as simple and understandable as possible. Currently, the LivePage implementation is Mozilla (and Firefox) only, although work is underway to write an IE implementation. Since the technique that can be used to send and recieve out of band events is highly browser dependent, the Nevow LivePage implementation will attempt to abstract these differences behind a common interface, and select the appropriate implementation based on which browser is being used.

The Nevow LivePage implementation is in a module named 'liveevil', and primarily consists of the function 'handler'. The following incomplete code fragment should illustrate how it is to be used:

```
from nevow import liveevil

def greeter(user, nodeName):
     user.sendScript("alert('Greetings. You clicked the %s node.')" % nodeName)

# Any string arguments after the event handler function will be evaluated
# as JavaScript in the context of the web browser and results passed to the
# Python event handler
H = liveevil.handler(greeter, 'this.name')

class Live(renderer.Renderer):
   document = tags.html[
   tags.body[
      ol[
         li(onclick=handler, name="one")["One"]
         li(onclick=handler, name="two")["Two"]
         li(onclick=handler, name="three")["Three"]
      ]
   ]
]
```

The included 'Chatola' demo application contains a full implementation of a web based chat application in around 100 lines of Python. If you are interested in using LivePage in your application, you are encouraged to experiment, and while the current implementation is quite usable, it should still be considered experimental.

Note also that the current implementation is cookie session based, so you will need to use multiple machines or browser versions (such as Mozilla and Firefox) to see the multi user aspects of the application in action.

Conclusion

The Nevow package contains many tools which are useful for performing web and XML related tasks, from generating simple XML documents using easy-to-write pure-python syntax to building a full-blown, highly interactive web application. Nevow was designed to allow application programmers to remove all logic constructs from HTML templates and to give them the power of pure Python as often as possible. It attempts to provide tools for writing your application which allow you to do so as expressively as possible, so you can focus on the important parts of your application instead of the mechanics of the web, while still allowing access to the low level details of HTML and HTTP when necessary.