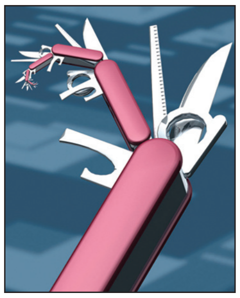

The Nitrogen Erlang Web Framework

Steve Vinoski

Erlang is best known for supporting scalable fault-tolerant systems, but it supplies a variety of features that also make it a great language for Web development. For example, its concurrency features let developers easily handle multiple simultaneous requests without having to perform explicit lock and thread management. Erlang's data types lend themselves well to dynamically generating pages based on common Web languages such as HTML, JavaScript Object Notation (JSON), and XML. The Erlang application model lets multiple components — each with its own supervisors and workers — run within the same virtual machine instance, even if they're independently developed, all of which simplifies the integration of different datastores, code generators, parsers, and other subsystems required for Web applications. The Erlang open source distribution even includes a built-in parser for HTTP messages, a basic Web server, and a Web client library.

Nitrogen, a popular open source Erlang Web framework (see http://nitrogenproject.com), adds to Erlang's support for Web systems. It builds on the foundation Erlang provides for Web development and extends it, making it easy for even those new to Erlang to quickly take advantage of the language. Because Nitrogen is quite feature-rich, I can't describe it in detail in this column, so instead I stick to the basics of Nitrogen and supply examples of its features, explain some of its capabilities, and show how it helps Web developers exploit Erlang to develop clean, robust websites that are straightforward to augment and maintain.

Building Nitrogen

Like many of today's open source projects, you can find Nitrogen on github.com. You can download the latest version by running the following command in a command shell:

```
git clone git://github.com/nitrogen/
  nitrogen.git
```

This creates a directory named `nitrogen` in your current working directory; changing to that directory lets you build Nitrogen. Because Nitrogen is a Web framework, it runs on top of a Web server, and it supports a variety of servers including Yaws (http://yaws.hyber.org),[1] Mochiweb (https://github.com/mochi/mochiweb), and inets, which is part of Erlang/OTP (http://erlang.org). The way you build Nitrogen depends on which server you want to use. For example, to build it on Yaws, you type the command

```
make rel_yaws
```

which builds a Nitrogen release including Yaws under the `rel/nitrogen` directory. Running the following commands then starts Nitrogen listening for requests on port 8000 with an interactive Erlang console attached:

```
cd rel/nitrogen
bin/nitrogen console
```

After starting Nitrogen, pointing your browser to http://localhost:8000/ will show you a "Welcome to Nitrogen" webpage.

Pages, Routing, and Elements

Part of the functionality Web servers and frameworks provide is converting HTTP requests into implementation-specific activities that attempt to fulfill each request. The server code bridges the boundary between client-side Web abstractions,

IC-16-06-Funw.indd 2 9/8/12 5:02 PM

such as HTTP verbs and URLs, and the implementation artifacts of the server-side application that make those abstractions concrete. Depending on the programming languages and Web frameworks they use, developers can typically implement Web resources in a wide variety of ways. Similarly, servers can use a variety of approaches to direct incoming requests to the implementation artifacts that instantiate Web resources.

In Nitrogen, each Web resource, or "page" in Nitrogen terminology, is represented by an Erlang module that typically exports several functions. One such function is `main/0`, where the "/0" represents the Erlang function's arity, or number of arguments. Nitrogen calls `main/0` when a client sends a request for the webpage associated with the module. As we'll see later, other functions in the module help build the page response for the client.

When Nitrogen receives a client request for a page, it uses a scheme based on page module names for routing the request to the appropriate page module. First, if the target URL path has an extension, such as ".jpg" or ".html," Nitrogen just treats it as a static file. The `index` module, found in the file `site/src/index.erl`, handles all requests made to the root URL path "/". Otherwise, Nitrogen translates the path portion of the target URL into a page module name by replacing "/" characters with underscores. For example, if a request arrives for a resource with URL path `/products/208809`, Nitrogen looks for a module named `products_208809`. In cases like this, though, having a separate module for each possible URL path is prohibitive; if thousands of products are listed on the website, for example, requiring a separate page module for each one would create a substantial development and maintenance headache. Fortunately, this isn't a problem because if it fails to find an exact page module match, Nitrogen instead looks for the longest matching module name. In this case, we could instead have a page module simply named `products` that would handle requests for all URL paths of the form `/products/<product-id>`. While handling a request, the functions in the `products` module can use the framework's `wf:path_info/0` function to retrieve the remainder of the URL path to determine which product ID was requested.

Creating a new page module is easy. Assuming you still have Nitrogen running as directed earlier, run the following commands from your shell (not from the Nitrogen console, but from your command shell) to create a `products` page module:

```
bin/dev page products
make
bin/dev compile
```

The first command generates an Erlang module named `products.erl` and stores it under the `site/src` directory. The generated `products.erl` file is based on the file `site/.prototypes/page.erl`. The second command recompiles any new files in the Nitrogen system, and in doing so picks up the new `products` module and compiles it (in this step, no other files are recompiled unless they're new or have been changed since the previous build). The third command communicates with the still-running Nitrogen system to tell it to rescan its installation to look for new files; when Nitrogen finds the newly compiled `products` module, it loads it to make it ready for client requests. Step 3 is helpful during development because it lets you reload modified modules and add new ones without restarting the Web server. If, after performing these steps, you then access http://localhost:8000/products from your browser, you'll be greeted with a page saying "Hello from products.erl!" Likewise, accessing http://localhost:8000/products/208809 produces the same result, given that it's also handled by the `products` page module due to Nitrogen's longest-match routing approach.

The new `products` page produces a webpage based on the `bare.html` template stored under `site/templates/bare.html`. This particular template is specified by the `products` page itself, in its `main/0` function:

```
main() ->
  #template { file=
    "./site/templates/bare.html"
}.
```

The body of `main/0` — which, as mentioned earlier, Nitrogen invokes to begin processing a request — returns an Erlang record of type `#template`. A record is a collection of named fields; here, the `file` field indicates the filename of the template Nitrogen will use to create the response to the client's request.

If you view the `bare.html` template file, you'll see it's mostly HTML, except for two unusual directives, both of which look like Erlang code. The first looks like a nested list containing a fully qualified invocation of an Erlang function defined in a module named `page`:

```
[[[page:body()]]]
```

In this context, Nitrogen treats the module `page` as referring to the current module, so for the `products` page, this snippet, known as a "callout" in Nitrogen terminology, invokes the `products:body/0` function, which appears in Figure 1.

Interestingly, this function appears to be a cross between Erlang and HTML, even though it's pure Erlang. The `#panel`, `#span`, `#p`, and `#button` records, called Nitrogen elements, resemble HTML elements, and their fields look much like HTML attributes. The body attribute of the `#panel`

```
body() ->
  [
    #panel { style="margin: 50px 100px;", body=[
      #span { text="Hello from products.erl!" },

      #p{},
      #button { text="Click me!", postback=click },

      #p{},
      #panel { id=placeholder }
    ]}
  ].
```

Figure 1. The `products:body/0` function. This function shows the use of Erlang records to model HTML elements, with record fields corresponding to HTML attributes. The function returns a list consisting of a single HTML panel, which in turn is composed of nested HTML elements. When a client requests a page implemented by the products module, Nitrogen invokes this function and converts the returned Erlang data to HTML and JavaScript to form the client response.

element is an Erlang list of child elements, and the return value of the `body/0` function is a list of one element, the `#panel`. When Nitrogen gets this return value, it translates each element in the list into HTML and JavaScript as appropriate, which it then uses to replace the `[[[page:body()]]]` callout in the template, after which it returns the completed page to the client.

For an actual website that lets customers order products, a development team using Nitrogen would write its own template for `product` pages. Such a template would have callouts wherever needed within the product page layout; these callouts would invoke functions in the `product` page module to obtain information such as the product name, manufacturer, price, and customer reviews. The `product` page module could obtain such information from a database, but in general, page modules could get information from virtually any back-end sources, because Nitrogen doesn't restrict applications to using only certain databases or back ends.

Actions and Events

Many Web applications today are incredibly responsive due to the choices Web developers have in terms of handling computations completely on the client in JavaScript, avoiding full-page refreshes using XMLHttpRequests or WebSocket, and dynamically and asynchronously updating page elements using server push approaches. To support these kinds of applications, Nitrogen provides a flexible, event-driven programming model based on actions, events, and long-polling techniques.

Nitrogen *actions* attach to pages or elements. Actions have *triggers* and *targets*; a trigger is the element that, when acted upon, causes an action to occur, and a target is the element that the action affects. Nitrogen supplies several actions for modifying pages, for effects such as showing, hiding, or fading, and for alerts and confirmations. Nitrogen builds many of these actions using the immensely popular jQuery JavaScript library (http://jquery.com). Attaching actions to elements is best done using the `wf:wire` functions. Each takes a list of actions to be applied, but they differ on triggers and targets: `wf:wire/1` treats the page as both trigger and target, `wf:wire/2` treats the page as trigger but takes a specific target, and `wf:wire/3` takes both trigger and target. For example, you could use `wf:wire/3` to set up a button as a trigger on the `products` page so that when it's clicked, it causes a photo of the product — its target — to appear or disappear.

Some actions occur entirely within JavaScript on the client side, but a nice feature of Nitrogen is how it enables events to also be easily handled on the server. For example, the `products:body/0` function in Figure 1 shows this button definition:

```
#button { text="Click me!",
  postback=click },
```

The `postback` attribute indicates that the value `click`, an Erlang atom, should be sent from the client back to the server when this button is clicked. Nitrogen directs it to the `event/1` function on the products page:

```
event(click) ->
    wf:insert_top(placeholder,
      "<p>You clicked the
      button!").
```

This function exploits Erlang's pattern-matching capabilities, such that only the atom `click` will match this function clause. This function calls the `wf:insert_top/2` function to place a new paragraph element at the top of the target element, which in this case is the `placeholder` panel at the bottom of the products page. Thanks to Erlang pattern matching, you can have as many different `event/1` function clauses for postbacks as you like by specifying different postback data, which can be any Erlang term, to indicate events from different elements. The `wf:insert_top/2` function is an example of Nitrogen's AJAX support, which lets servers efficiently add, remove, and update elements of the current page.

But Wait, There's More

As I mentioned earlier, Nitrogen provides numerous features, too many to cover in this column space. In describing only its most basic capabilities, I've hardly scratched the surface, given that Nitrogen also supports the following features and more:

- request redirection,
- session state and page state,
- cookies,
- HTTP header manipulation,
- authorization and authentication,
- asynchronous updates via HTTP long-polling,
- validation, and
- custom elements, actions, and templates.

Nitrogen's elegant combination of features — its event-driven programming model, the power of the jQuery library and JavaScript, its portability across Erlang Web servers, and the fact that it lets developers use Erlang for both client and server code — is definitely compelling. Even if you're a Web developer either new to Erlang or interested in trying out the language, Nitrogen could be a great way to ease into it. For more information, please refer to the Nitrogen website at http://nitrogenproject.com.

And Lastly, Thank You

Over the past few years, my guest columnists, coauthors, and I explored the application of functional programming languages, tools, and techniques to the domain of Web development. We dove into languages such as Erlang, Haskell, JavaScript, Roy, and Scala as we examined the details of a variety of Web servers and frameworks. My initial hope for the column was to take advantage of the broad Web development domain to help increase awareness of the general benefits of using functional programming approaches. Based on reader feedback, I believe we achieved that goal.

Now, after 10 years of writing for *Internet Computing*, the time has come for a break. I've been lucky to have collaborated with a number of bright, innovative coauthors and guest columnists. I've also been fortunate to have worked with some incredibly talented and patient editors — namely, Steve Woods in the early years, Jennifer Gardelle in the middle, and for the past few years, Rebecca Deuel-Gallegos — who work incredibly hard behind the scenes to ensure this magazine's consistent high quality. But perhaps best of all, I've been blessed to have thoughtful readers like you providing excellent feedback, asking great questions, and offering kind encouragement. Thank you all for everything.

Reference

1. S. Vinoski, "Yaws: Yet Another Web Server," *IEEE Internet Computing*, vol. 15, no. 4, 2011, pp. 90–94.

Steve Vinoski is an architect at Basho Technologies in Cambridge, Massachusetts. He's a senior member of IEEE and a member of ACM. You can read Vinoski's blog at http://steve.vinoski.net/blog/ and contact him at vinoski@ieee.org or on Twitter at @stevevinoski.