Build it with Nitrogen

The fast-off-the-block Erlang web framework

Lloyd R. Prentice & Jesse Gumm

dedicated to:

Laurie, love of my life— Lloyd

Jackie, my best half— Jesse

and to:

Rusty Klophaus

and other giants of Open Source— LRP & JG

Contents

Authors' note

A big question popped to mind when we decided to write this book. How much can we assume about you, the reader.

Here's what we came up with:

- We assume that you're comfortable with the Unix/Linux command line.
- We assume that you have experience building web sites; understand HTML and CSS.
- We don't assume that you know Erlang, but the more you know the better. We'll point the way and suggest learning resources. We have faith that you'll dig in, study hard, and pick up Erlang as you go.
- Note to wizards: No condescension intended if we cover stuff you already know.
- We further assume that you have a recent version of Erlang installed on your computer. Check out Chapter Two if not.
- Our last assumption is that you're passionate about building reliable, scalable web applications using high-productivity tools.

This is most definitely a learn-by-doing book. You may get the gist by reading at the beach. But to truly master Nitrogen, plop this book down beside your keyboard and work your way through command by command.

LRP & JG

Before we begin

Some URLs and directory paths are too long to fit on one line in this book. In these cases we will break them like this:

```
http://docs.basho.com/riak/1.3.0/tutorials/installation/
      Installing-Erlang/
```

Be sure to rejoin them when you paste them into your browser or use them in code.

Grey boxes depict command-line interactions. Computer display is in normal type; *the commands you enter are in* **bold face**.

Part I.

Frying Pan to Fire

1. You want me to build what?

Welcome to the madhouse! Glad to have you aboard.

As you see, we're a lean-and-mean outfit— more work piling up than we can handle. Clients banging at the door.

Erlingo! they call us. Your friendly webspinners.

Our language? *Erlang OTP*.

Preferred web framework? *Nitrogen.*

Don't know either? No worry. We'll get you squared away.

Why Erlang?

Our clients demand applications that handle high-traffic loads with nine nines reliability. Erlang excels at both.

Hard to learn?

Excellent resources in Appendix C. Dig in, persevere, and you'll be productive before you know it.

And why Nitrogen?

Nitrogen is one of the most productive ways we've found to build full-functionality web applications. You'll be working hand-in-glove with Rusty and Jesse, our in-house Nitrogen wizards.

Stick with the dynamic duo, kid, and you'll be a wizard in no time.

Chomping at the bit are you?

Marketing needs an interactive welcome board for our corporate lobby. Clamoring for it.

Deadline— day after tomorrow. Bet-the-company client conference coming up. Think you can deliver?

Here's Jesse, our head developer. He'll give you heads up on how we do things around this place.

2. Enter the lion's den

Whoa! Day after tomorrow? That's harsh. But that's Bossman— no moss under that dude's feet. So we best get crackin'.

These three boxes power our trusted in-house development network. We call them Alice, Bob, and Mallory. Yes, indeed, we take security seriously. Rusty will read you in on our security practices later.

We also have a remote server— hostname Charlie. Plan to lease another— probably call it Dora.

Why all the hardware?

Erlang is explicitly designed to support distributed computing. We use the machines on this network to develop and test distributed applications and databases. Set it up on the cheap.

Alice and Mallory are old Dell Optiplex 745s running Linux Debian Linux Wheezy. Dual-core, gig of RAM. Company up the road traded up so Bossman picked these puppies up for fifty bucks apiece. Bossman likes to say lean-and-mean. Truth— the dude's a cheapskate.

Yes, we could we use *Vagrant,* the cloud, or some such, instead of physical machines. But Bossman is old school. We're trying to talk him around.

Bob is a custom built PC running Ubuntu 14.04. Three-core AMD processor, six gigs of RAM.

I tap into the network with my personal MacBook Pro.

Fact is, you don't need all this kit to develop Nitrogen apps. You can do it on your *Windows* notebook at Starbucks if you're so inclined. I've heard of folks running Nitrogen on *Raspberry Pi*.

But we're looking toward bigger things here— reliable, industrial strength, scalable apps.

2.1. The big picture

Before we begin, let me paint the big picture.

The challenge of web application development comes down to managing a jumble of languages and network protocols.

You, Dude, learned your native language effortlessly in the bosum of your family. But as a web application developer your task is to convince hardware on both server and client sides to do your bidding. Problem is, the stupid machines don't speak your native tongue.

On the client side, the browser responds to HTTP/HTTPS protocols carrying HTML, CSS, and JavaScript messages which, in turn, convey and present information structured as natural language, sound, and images both still and moving.

The server responds to some babel of computer languages to marshal the HTTP/HTTPS, HTML, CSS, JavaScript, natural language, sound, and images both still and moving through the Internet to the client.

It's almost too much for the feeble human mind to encompass. The nitty gritty tedium of it all is mind numbing.

So this is where Nitrogen comes in.

Nitrogen harness the power of Erlang to manage all— well, most all— of the fiddly semantics and syntax of HTTP/HTTPS, HTML, CSS, and JavaScript. This means that you have that much less to think about when you craft your cunning web application. In the spell of creative ferment, you can produce way cool web apps all that much faster.

We're not saying that you don't have to understand the alphabet soup of web technologies. The deeper you understand them the better. We are saying that mastery of Erlang Nitrogen will make you far more fluent and productive.

What's the trick?

Nitrogen combines the structural convenience of Erlang records to structure data, the fluency of Erlang functions to execute logic and embed JavaScript, and the power of Erlang as a development platform to organize and abstract the semantic and syntactical fussiness of server/web/browser communication.

Enough already. Let's install Nitrogen.

2.2. Install Nitrogen

Take a seat, citizen, and we'll log into Bob— show you how to compile Nitrogen.

Nitrogen is written in Erlang and JavaScript. No, you don't need to know much JavaScript. Nitrogen translates.

Erlang is already installed on Bob, of course, but if you want to install it at home, take a look here:

```
http://docs.basho.com/riak/1.3.0/tutorials/installation/
      Installing-Erlang/
```

You could install Nitrogen by downloading a binary package. Here's the go-to link:

```
http://nitrogenproject.com/downloads
```

But fact is, Nitrogen is easy to compile from source. So let's do that instead.

Let's go ahead and clone it from Github and make a test project (here, conveniently called `testproj`)

```
~$ git clone git://github.com/nitrogen/nitrogen
~$ cd nitrogen
~/nitrogen$ make rel_inets PROJECT=testproj
```

All that text scrolling up the terminal? That's *make* working hard on our behalf to compile Nitrogen. You'll experience a few pauses while Erlang generates your release, so be patient.

Can you imagine entering all those terminal commands whizzing up your screen by hand? Be at it all week. That's the beauty of *make*— automates the build process.

Indeed, it's worth getting to know your way around *make*. Appendix B will give you a bird's-eye rundown on Erlang build tools including *make*.

Looks like we're done—

```
...
Generated a self-contained Nitrogen project in ../
      testproj, configured to run on inets.
make[1]: Leaving directory '/home/lloyd/Erl/Eval/
      nitro/nitrogen' Jesse@Bob:~/nitrogen$
```

Note that *inets* refers to Erlang's built-in webserver. That's one of many things we like about Erlang— batteries included.

OK, one more step:

```
~/nitrogen$ cd ../testproj
~/testproj$ bin/nitrogen console
```

And, we see:

```
...
Erlang/OTP 17 [erts-6.0] [source] [64-bit] [smp:3:3]
      [async-threads:5] [hipe] [kernel-poll:true]
Running Erlang
Eshell V6.0  (abort with ^G)
(testproj@127.0.0.1)1>
```

This tells us that we're in the *Erlang shell.* Much to explore here, but we'll save it for later.

Now, point your browser:

```
localhost:8000
```

Wallah! As me Cockney mates would put it, "Bob's your uncle!"

"WELCOME TO NITROGEN" in our very own browser.

Nitrogen lives!

Browse around while I snag us a jolt of *Club-Mate.*

Haven't tried it? Official drink of the Chaos Computer Club. Our Berlin consultant sent it over special.

2.3. Lay of the land

OK, I'mmm back!

Let's cruise the directories to see what strikes our eye.

Open a new terminal. This will give us two terminals— an *Erlang shell* and a *Unix shell.* In the *Unix shell*, cd down to *site*, and list it:

```
~$ cd testproj/site
~/testproj/site$ ls -l
```

Here we are:

```
drwxrwxr-x 2 jess jess 4096 May 9 13:21 ebin
drwxrwxr-x 2 jess jess 4096 May 9 13:20 include
drwxrwxr-x 4 jess jess 4096 May 9 13:20 src
drwxrwxr-x 6 jess jess 4096 May 9 13:21 static
drwxrwxr-x 2 jess jess 4096 May 9 13:20 templates
```

The code in this directory built the web page displayed in our browser.

First, let's look into *templates*—

```
~/testproj/site$ cd templates
~/testproj/site/templates$ ls -l
```

```
-rw-rw-r-- 1 jess jess 1442 May  9 13:20 bare.html
-rw-rw-r-- 1 jess jess 1265 May  9 13:20 mobile.html
```

And peek in on *bare.html.*

We're partial to *vim* around here, but you can develop Nitrogen applications in whatever code editor you prefer.

```
~/testproj/site/templates$ vim bare.html
```

As you see, *bare.html* is a standard **.html* file. In the head section it's loading a bunch of **.js* files and style sheets.

You've built websites, so there's nothing here that you haven't seen before.

But, in the body, we see:

```
<body>
[[[page:body()]]]
<script> [[[script]]] </script>
</body>
```

And here, my friend, is Nitrogen's secret sauce. We'll unveil the tantalizing mysteries over the next few hours.

But first, let's look into other directories in *site*. Open up *ebin* and list it:

```
~/testproj/site/templates$ cd ../ebin
~/testproj/site/ebin$ ls -l
```

```
-rw-rw-r-- 1 jess jess 47168 Jun  5 14:18 index.beam
-rw-rw-r-- 1 jess jess 47920 Jun  5 14:18 mobile.beam
-rw-rw-r-- 1 jess jess   351 Jun  5 14:18 nitrogen.app
-rw-rw-r-- 1 jess jess   992 Jun  5 14:18 nitrogen_app.beam
-rw-rw-r-- 1 jess jess   992 Jun  5 14:18 nitrogen_main_handler.beam
-rw-rw-r-- 1 jess jess  1200 Jun  5 14:18 nitrogen_sup.beam
```

Note *nitrogen.app*.

App files are a VERY BIG DEAL in Erlang. The instance in *ebin* was automatically generated by *nitrogen.app.src* in the *src* directory. We'll get to that in a moment.

Note next all the **.beam* files.

The grey-beards tell us that *beam* stands for Bodan's Erlang Abstact Machine.

Like Forth and Java, Erlang runs on a virtual machine. Erlang source compiles down to **.beam* files and the **.beam* files execute on an Erlang virtual machine.

If it doesn't already exist, the *ebin* directory and all in it is created automatically when you compile Erlang source. In principle, you'll never have to look into *ebin* again— unless you want to confirm that your program has compiled.

With that in mind, look now into *src*:

```
~/testproj/site/ebin$ cd ../src
~/testproj/site/src$ ls -l
```

```
drwxrwxr-x 2 ... actions
drwxrwxr-x 2 ... elements
-rw-rw-r-- 1 ... index.erl
-rw-rw-r-- 1 ... mobile.erl
-rw-rw-r-- 1 ... nitrogen_app.erl
-rw-rw-r-- 1 ... nitrogen.app.src
-rw-rw-r-- 1 ... nitrogen_main_handler.erl
-rw-rw-r-- 1 ... nitrogen_sup.erl
```

Note *nitrogen.app.src.* Compare content with *nitrogen.app* in *ebin.*

App files provide meta data that tell the Erlang compiler where to find start and stop functions and other resources the application needs.

Check out Appendix A. It'll put you way down the road toward understanding how Erlang applications are structured and the secrets behind the widely touted reliability of Erlang applications.

Notice also how all the **.erl* files in *src* have doppelgängers in *beam.*

Makes sense— **.erl* source files compile to **.beam* files, remember?

Explore the **.erl* files if you wish.

Squint while you eyeball the *nitrogen*.erl* files. The names and structure of these files follow patterns that you'll see across nearly every Erlang OTP program you'll ever develop. Your understanding of OTP will be wide and deep when you get a handle on why this is so.

Dig in here for details:

```
http://www.erlang.org/doc/design_principles/
      applications.html#id74089
```

So what's the point of the *actions* and *elements* directories?

We'll dive into them when we start developing your web app for real.

But for now, look into *index.erl*:

```
~/testproj/site/src$ vim index.erl
```

And, yeah man, there's the code that produced the Nitrogen home page displayed our browser!

Feel free to bop around the directories and subdirectories in *site*.

But look— time for lunch. We'll tackle that assignment Bossman gave you for real after we've fueled up.

Part II.

Projects

3. nitroBoard I

Detailed specs? From Bossman? You've got to be kidding.

Typical client— expects developers to be mind-readers. But no worry. We'll brainstorm.

Users— visitors drop into front office. What do they need to know?

Company logo. Check.

VIP welcome line? Hey, that's bodacious. Reads "Welcome!" on days when we don't expect VIP visitors— "Welcome <vip visitor> when a client or VIP is expected to drop in.

More than one VIP?

Good point. Matter of layout, I think.

A visitors' database?

Yeah, agreed, but simple simple.

OK, what else?

Hmmm— so, we'll need an admin page to keep the board up-to-date.

Authentication? Nah— It'll be on a trusted network.

Say again? Boring project? Might surprise you. Certainly more instructive than "Hello World!" wouldn't you say?

Work plan?

Hey— boss is going to love you.

3.1. Plan of attack

1. Create new project

2. Define routes

3. Prototype pages
    a) corporate logo
    b) welcome
    c) visitors admin
4. Develop welcome page
5. Data persistence
    a) visitor db
6. Develop admin page
    a) visitor form
7. Display visitors
8. Test/debug/revise

Kill, Baby, Kill!

Before we dive in, let's kill the Nitrogen instance displayed in your browser.

Why? It's hogging port 8000. We're going to need that puppy.

As you'll recall, the command *bin/nitrogen console* launched an Erlang shell. Turns out, it also started the *inets* webserver.

So what thinkest thou? Kill the shell, will we also kill Nitrogen?

Let's try. Type **"q()"** at the Erlang shell prompt:

```
Running Erlang
Eshell V6.0 (abort with ^G)
(nitrogen@127.0.0.1)1> q().
```

Promising— we see:

```
ok
jesse@Bob:~/testproj/site/src$
```

Now refresh your browser:

```
Unable to connect
```

Good on ya. The foul deed is done.

The *"q()"* command is one of several ways to exit from the Erlang shell. Dig into the Erlang docs to discover others. Don't forget that every Erlang shell command must be terminated with a period before the command will execute.

Sooner rather than later you'll need to know your way around the Erlang shell. Why not start now? Make it your best friend.

```
http://www.erlang.org/doc/man/shell.html
```

3.2. Create a new project

So now, back to business. Our first step is to create a new project. But first, we face two decisions:

1. Compile a "full" or "slim" *release*?
2. On which webserver?

As you'll recall, when we first installed the nitrogen demo, we entered the following command:

```
~/nitrogen$ make rel_inets
```

Result: Erlang compiled a *"full release"* on *inets*, Erlang's built-in webserver. The release included *ERTS*, the *Erlang Run Time System*, and all else required to run the demo.

```
http://erlang.org/doc/man/inets.html
www.erlang.org/documentation/doc-5.0.1/pdf/erts-5.0.1.pdf
```

Tar up a full release, ship it to another 'ix system, and it will run without the bother of installing Erlang separately.

Turns out, if Erlang is installed on the target system, you won't need *ERTS*. In this case you can compile a "*slim release*."

We'll go for slim here.

Added bonus: Nitrogen offers a selection of webservers. See Chapter 13: *How to Choose a Webserver*.

Since we expect *nb* to experience very light loads, we'll stick with *inets*. Also, let's call this project *nb* for *nitroBoard*. Thus, we enter:

```
~/testproj/site/src$ cd ~/nitrogen
~/nitrogen$ make slim_inets PROJECT=nb
```

From here on we'll morph the nitrogen demo source into *nb*. Anything can happen, so let's put *nb* under version control. *Git* is our version control system.

```
http://git-scm.com/
```

Appendix D will bring you up to speed on our git workflow. When you've finished your app, we'll post it up on GitHub.

```
~/nb$ git init
Initialized empty Git repository in
      /home/jess/nb/.git
```

Let’s peek at what we’ve got:

```
~/nb$ ls -al
```

```
...
drwxrwxr-x 2 jess jess   4096 May 16 12:14 bin
-rw-rw-r-- 1 jess jess     89 May 16 12:14 BuildInfo.txt
-rwxrwxr-x 1 jess jess  17922 May 16 12:14 do-plugins.escript
drwxrwxr-x 2 jess jess   4096 May 16 12:14 etc
-rwxrwxr-x 1 jess jess   1323 May 16 12:14 fix-slim-release
drwxrwxr-x 7 jess jess   4096 May 16 12:15 .git
drwxrwxr-x 8 jess jess   4096 May 16 12:14 lib
drwxrwxr-x 3 jess jess   4096 May 16 12:14 log
-rw-rw-r-- 1 jess jess   1927 May 16 12:14 Makefile
-rw-rw-r-- 1 jess jess   1027 May 16 12:14 plugins.config
-rwxrwxr-x 1 jess jess 135939 May 16 12:14 rebar
-rw-rw-r-- 1 jess jess   1216 May 16 12:14 rebar.config
drwxrwxr-x 3 jess jess   4096 May 16 12:14 releases
drwxrwxr-x 8 jess jess   4096 May 16 12:14 site
jesse@Bob:~/nb$
```

Way cool!

Lots to be learned in this directory— in particular, *bin* and *site* will play big in our life. The *bin* directory contains useful tools; *site* is where we'll find all the files we need to run our site.

For present purposes we can ignore the other directories so don't feel overwhelmed. Do note, however, the *.git* directory. This is where git stores version records.

For now, we want to bring up the demo source and start morphing it to our needs.

Double-check that you've closed the Nitrogen instance you were working with earlier and that no other programs are using port 8000.

Now enter:

```
~/nb$ bin/nitrogen console
```

We're back in the Erlang shell— covering old ground now:

```
Erlang/OTP 17 [erts-6.0] [source] [64-bit] [smp:3:3]
      [async-threads:5] [hipe] [kernel-poll:true]
Running Erlang Eshell V6.0 (abort with ^G)
      (nb@127.0.0.1)1>
```

There's ye olde Erlang shell. Keep an eye on it. It will come in handy.

Now, open our new instance of Nitrogen in your browser:

```
localhost:8000
```

And there, in the browser, is our patient, all prepped out for cosmetic surgery.

So now, IN A NEW TERMINAL, e.g. Unix shell, open up the site directory. You now have two terminals open. We'll work in the Unix shell:

```
~/nb$ cd site
~/nb/site$ ls -l
```

Looks familiar:

```
drwxrwxr-x 2 jess jess 4096 May 12 15:02 ebin
drwxrwxr-x 2 jess jess 4096 May 12 15:02 include
drwxrwxr-x 4 jess jess 4096 May 12 15:02 src
drwxrwxr-x 6 jess jess 4096 May 12 15:02 static
drwxrwxr-x 2 jess jess 4096 May 12 15:02 templates
```

And so, Nurse Rached, it's time. Scalpel!

3.3. Prototype welcome page

First cut, let's say *nb* needs two user-facing pages:

- *index* — lobby display
- *visitors_admin* — maintain visitor appointments

The Nitrogen demo conveniently provides an index page that we can morph into our welcome board. Indeed, you're looking at it in your browser. From your work terminal, e.g. Unix shell, bop into the *src* directory and open up index.erl:

```
~/nb/site$ cd src
~/nb/site/src$ vim index.erl
```

The *index.erl* file is a plain vanilla *Erlang module*. In a moment, we'll make minor changes to the function *inner_body/0*. But first, shift attention back to the Erlang shell:

```
...
Running Erlang
Eshell V6.0 (abort with ^G)
(nitrogen@127.0.0.1)1>
```

At the Erlang shell prompt enter the following:

```
(nitrogen@127.0.0.1)1> sync:go().
```

And you should see:

```
...
Starting Sync (Automatic Code Compiler / Reloader)
Scanning source files...
Growl notifications disabled
ok
```

What happened here?

The Erlang function *sync:go/0*[1] tracks and automatically compiles changes that you make in Erlang source files. It's pretty neat.

Let's change the function *inner_body/0* in *index.erl*. Out of the gate, the first few lines of *inner_body/0* look like this:

```
inner_body() ->
   [
      #h1 { text="Welcome to Nitrogen" },
      #p{},
      "
      If you can see this page, then your Nitrogen
      server is up and running. Click the button
      below to test postbacks.
      ",
      #p{},
      #button { id=button, text="Click me!",
            postback=click },
            #p{},
      ...
```

Make the following changes and save:

[1]Sync is an application for automatically recompiling and loading changes to Erlang code. While it was originally a part of Nitrogen, it has since been split off into its own application which can be used in any Erlang application. You can find it on Github: https://github.com/rustyio/sync

```
    inner_body() ->
       [
          #h1{ text="Erlingo! WEBSPINNERS" },
          #h1{ text="WELCOME!" },
          #h2{ text="Joe Armstrong" },
          #h2{ text="Rusty Klophaus" },
          #p{},
          #button { id=button, text="Click me!",
                postback=click },
                #p{},
          ...
       ].
```

Cast your eye at Erlang shell, refresh your browser, and behold:

```
=INFO REPORT==== 17-May-2014::14:32:48 ===
/home/jesse/nb/site/src/index.erl:0: Recompiled.
=INFO REPORT==== 17-May-2014::14:32:48 ===
index: Reloaded! (Beam changed.)
```

Refresh your browser and, by gum!, to delight of our eyes, your changes have been compiled!

No, that screen ain't pretty. But it gives us something to work with.

What's going on here?

3.4. Anatomy of a page

In Nitrogen-speak a *page* is an *Erlang module*. It is ***NOT*** an HTML page, but it will help build one.

Let's examine *index.erl*. The first few lines are plain vanilla Erlang:

```
%% -*- mode: nitrogen -*-
-module(index).
-compile(export_all).
-include_lib("nitrogen_core/include/wf.hrl").
```

The first line is a *comment.* The percent symbols at the beginning of the line gives it away. The next three lines are Erlang *attributes.* The hyphen at the beginning of the line give them away.

The first attribute, `-module`, with appropriate argument, is mandatory in every Erlang module. It declares the name of the module. Note that the name of the module is the same as the filename less the suffix "*.erl.*" This is also mandatory in Erlang. Also note that module attribute ends with a period, as do all Erlang attributes and functions.

The second attribute declares which functions in the module can be exported. In this case, every function in index is exported, that is, can called from other modules.

If we wished to export a subset of functions and keep others private, we'd use a different attribute, `-export([<export1/n1>, <export2/n2>, ...]).`

The third attribute, `-include`, imports the *wf.hrl* file from the include file. Usually, **.hrl* will import one or more *record* definitions.

Now note that index.erl has five functions, `main/0`, `title/0`, `body/0`, `inner-body/0,` and `event/1`. Each function has the form `<function name>(<function arguments>) -> <function body>.`

Again, note the period at the end.

If you glance back and forth between the body of the five functions in *index.erl* and the copy displayed in the browser, you should gain a fair understanding of each function's purpose in life. The function `main/0` might trip you up. If you guessed that it's calling the template *bare.html* you'd be right on the button.

What's with this *function/X* thing we keep writing?
You may have noticed that we're referring to functions in an odd way: `body/0`, `sync:go/0`, etc.. This is using an Erlang convention of referring to functions by their *arity.* Arity means "the number of arguments that can be passed to a function." So `sync:go/0` is simple: it takes zero arguments, meaning it can be called like `sync:go()`. By comparison, the Erlang function `lists:map/2` takes two arguments, and is called like `lists:map(SomeFunction, SomeList)`. Further, this convention passes as a part of Erlang's syntax and its support of first class functions . You can pass around function references as arguments by assigning them to variables using this arity syntax. For example:

```
    MyFunction = fun my_module:some_fun/3,
    MyFunction(A, B, C).
```

Simply put, this will assign the `my_module:some_fun/3` function to the variable `MyFunction`, and then `MyFunction` can be invoked exactly the same as `my_module:some_fun/3.`
So now you know.

The purpose of `event/1` shouldn't surprise you. It implements an *action* triggered by a button click, giving us a clue as to how Nitrogen implements interactive functionality.

Take away: an HTML page in Nitrogen is rendered by an HTML *template* that embeds an Erlang module called a Nitrogen *page.* Each Nitrogen *page* should accomplish just one task such as:

- Allow the user to log in (**user_login.erl**).
- Change the user's preferences. (**user_preferences.erl**)
- Display a list of items. (**items_view.erl**)
- Allow the user to edit an item. (**items_edit.erl**)

So how is a Nitrogen *page* rendered?

Here's the simple story:

1. User hits a URL.
2. URL is mapped to a module.
3. Nitrogen framework calls `module:main()`
4. `module:main()` calls a `#template{}`
5. `#template{}` calls back into the Nitrogen *page* (or other modules)
6. Nitrogen framework renders the output into HTML/JavaScript.

Hot diggity! We're in the thick of it now.

Brief aside: See if you can find an Erlang *list* in `inner_body/0`. It will look like `[a,b,c,...]`. What do you suppose that's about?

Lists are big business in Erlang. More here:

```
http://www.erlang.org/doc/man/lists.html
```

3.5. Anatomy of a route

Note step two above. A URL that maps to a module is called a *route*. Here's how Nitrogen processes routes:

- Root page maps to index.erl

  ```
  http://localhost:8000/ -> index.erl
  ```

- If there is an extension, assume a static file

  ```
  http://localhost:8000/routes/to/a/module
  http://localhost:8000/routes/to/a/static/file.html
  ```

- Replaces slashes with underscores

  ```
  http://localhost:8000/routes/to/a/module ->
        routes_to_a_module.erl
  ```

- Try the longest matching module

  ```
  http://localhost:8000/routes/to/a/module/foo/bar ->
        routes_to_a_module.erl
  ```

- Modules that aren't found go to *web_404.erl* if it exists
- Static files that aren't found are handled by the underlying platform (not yet generalized.)

This suggests that *nitroboard* will have at least three pages: *index.erl*, *visitors_admin.erl*, and *directory_admin.erl*. They will be called as follows:

```
http://localhost:8000/ -> index.erl
http://localhost:8000/visitors/admin -> visitors_admin.erl
http://localhost:8000/directory/admin -> directory_admin.erl
```

With routes under our belt, what's this template business?

3.6. Anatomy of a template

A *template* is your grandfather's HTML page with a dash of Nitrogen's secret sauce— one or more *callouts*. The *callout* below, for instance, slurps a Nitrogen page into the Template:

```
[[[module:body()]]]
```

This *callout* slips JavaScript into the Template:

```
    [[[script]]]
```

The *callouts* look like Erlang, but don't be fooled. They have similar form, *module:function(Args)*, but they're pure Nitrogen, and they won't render full

3.7. Elements

An *element* is a chunk of HTML or an *Erlang record* that renders to HTML.
Knew you'd ask. Here's Jesse's twenty-second take on *Erlang records:*

- A *tuple* is a basic Erlang data structure of the form:

  ```
  {<datum 1>, <datum 2>, <datum 3>}
  ```

- Tuples are fixed length, but can be any length. Problem is, *tuple* gets too long, you get confused as to which chunk of data goes into which slot.
- An *Erlang record* is, arguably, a hack to solve the problem. In source, an *Erlang record* is defined as:

  ```
  #label {name1=<datum 1>, name2=<datum 2>,
        name3=<datum3>}
  ```

- When compiled, our *record* is a plain old *tuple*. But the *record definition* provides the compiler with enough information to enable you to address fields in the *tuple* by name.

More here:

```
http://www.erlang.org/doc/reference_manual/records.html
```

So, if our page contains an element of the form:

```
#label { text="Hello World!" }.
```

It will render as:

```
<label class="wfid_tempNNNNN label">Hello World!</label>
```

Each Nitrogen element has a number of basic properties. All of the properties are optional.

- **id** – Sets the name of an element
- **actions** – Add Actions to an element. Actions are described later
- **show__if** – Set to true or false to show or hide the element
- **class** – Set the CSS class of the element
- **style** – Add CSS styling directly to the element

Look over the *Erlang record definitions*, er, I mean, *Nitrogen elements* in `index:inner_body/0.`

Extra credit: map each element to:

1. An Erlang tuple
2. HTML.

The `index:inner_body/0` bit? That's more Erlang speak for *module*, *function*, and *arity*, where *arity* is number of arguments. But you knew that already.

More here:

```
https://github.com/nitrogen/nitrogen/wiki/
      Nitrogen-Elements
```

Nitrogen sports more than 70 elements for most anything you want to display on the screen. Categories include:

- **html**
- **html5**
- **forms**
- **layout**
- **mobile**
- **tables**
- **other**

Check it out:

```
http://nitrogenproject.com/doc/elements.html
```

Find examples, including module source, here:

```
http://nitrogenproject.com/demos/simplecontrols
```

And more :

```
http://nitrogenproject.com/demos
```

If none of the elements provided by Nitrogen suit, you can create your own. See Chapter 15.

3.8. Actions

A *Nitrogen action* can either be JavaScript, or some record that renders into JavaScript.

Examples:

- `#button { text="Submit", actions=[`
  `#event{type=click, actions="alert('hello');"}`
  `]}`
- `#button { text="Submit", actions=[`
  `#event{type=click, actions=#alert { text="Hello" }`
  `]}`

Sometimes setting the `actions` property of an element can lead to messy code. Another, cleaner way to wire an action is the `wf:wire/N` function:

- `wf:wire(mybutton, #effect{effect=pulsate})`

The above code might not do what you expect. Indeed, as written, it would immediately cause the mybutton element to pulsate, rather than pulsating when you click the button. Instead, you'll want to use the `#event{}` action to require some kind of user interaction to trigger the action.

Example:

```
wf:wire(mybutton, #event {type=click, actions=[
    #effect {effect=pulsate}
]})
```

It's worth noting here that there are also a few elements that contain helper attributes called `click` (most notably `#link{}` and `#button{}`), so that you don't have to use the `actions` attribute:

- `#button{text="Submit" click=#alert{text="Hello"}}`

3.9. Triggers and Targets

All actions have a *target* property. The *target* specifies what element(s) the action effects.

The event action also has a *trigger* property. The *trigger* specifies what element(s) trigger the action.

For example, assuming the following body:

```
[ #label { id=mylabel, text="Make Me Blink!" },
  #button { id=mybutton, text="Submit" } ]
```

The following twocalls are identical in that both will make it so that when you click the "Submit" button, the label will pulsate (I've bolded the key differences for clarity):

```
wf:wire(#event{type=click, trigger=mybutton, target=mylabel,
        actions=#effect { effect=pulsate } }).
wf:wire(mybutton, mylabel, #event{type=click,
        actions=#effect { effect=pulsate } }).
```

As you see, you can also specify the Trigger and Target directly in `wf:wire/N`. It takes three forms:

- Specify a trigger and target.
- Use the same element for both trigger and target.
- Assume the trigger and/or target is provided in the actions. If not, then wire the action directly to the page. (Useful for catching keystrokes.)

Examples:

- `wf:wire(Trigger, Target, Actions)`

- `wf:wire(TriggerAndTarget, Actions)`
- `wf:wire(Actions)`

Confused? Here's the big picture:

1. *Elements* make HTML.
2. *Actions* make JavaScript.
3. An action can be *wired* using the actions property, or *wired* later with *wf:wire/N*. Both approaches can take a single action or a list of actions.
4. An *action* looks for *trigger* and *target* properties. These can be specified in a few different ways.
5. Everything we have seen so far happens on the client.

Hey, Dude, volleyball time— nerds vs. marketing. Let's wrap this puppy mañana.

3.10. Enough theory

Mornin', Dude. My my— look at those bloodshot eyes. Up all night studying Erlang I take it.

Marketing sent us a corporate logo file. It's called *erlingologo.png*. I've taken liberty of storing it in *~/nb/site/static/images*.

> **Note:** You can clone it here: XXXXXXX. Or, you can make your own **.png* image and fake it.

Erlingo!
WEBSPINNERS

As you'll recall, we stubbed the logo into our prototype page with the following element:

```
    #h1 { text="Erlingo! WEBSPINNERS" }
```

Problem is, we'd like the logo to show up on all user-facing pages. We COULD insert the logo element in every page. But there's a more efficient way— install it in the template.

So let's do that.

In your Unix shell, open bare.html:

```
~/nb/site/templates$ vim bare.html
```

And add the following code:

```
...
<body>
    <div class=container_12>
        <div class="grid_8 prefix_2 suffix_2">
            <div class="grid">
              <a href="/">
                 <img src="/images/erlingologo.png"
                      style="width:100%" />
              </a>
              [[[page:body()]]]
            </div>
        </div>
    </div>

    <script>
        [[[script]]]
    </script>
</body>
</html>
```

So that's good to go.

No we're going to make some slight changes to the homepage (index.erl). If, by chance, you closed your Erlang shell, re-open it:

```
~/nb$ bin/nitrogen console
```

Don't forget to call *sync:go()* so that we can take advantage of Erlang's sweet auto-code reloading.

```
...
Running Erlang
Eshell V6.0 (abort with ^G)
(nitrogen@127.0.0.1)1> sync:go().
```

Now, in your Unix shell, open *index.erl*:

```
~/nb/site/templates$ cd ../src
~/nb/site/src$ vim index.erl
```

Time now to morph index.erl for real. Delete `event/1 and inner_body()` and revise `body()` to look like this:

```
body() ->
    [ #h1 { text="WELCOME!" },
      #h2 { text="Joe Armstrong" },
      #h2 { text="Rusty Klopaus" }
    ].
```

Now, call up your browser:

```
localhost:8000
```

And— Bingo!

That was easy. With a touch of work in CSS we could reposition the welcome line to make it more attractive, but we'll do that later. Let's focus on the visitor functionality. This may get tricky.

3.11. Visitors

On some days *Erlingo!* has no VIP visitors. On a busy day, we may have three or four. Miss Moneypenny, Bossman's secretary, books visitors days in advance. So this suggests that we need a database of visitors. Nothing fancy. Suppose records in this database have the following fields:

```
Date
Time
Name
Company
```

Now, suppose we have a *cron* task that queries the database on the date field every morning just after midnight to retrieve visitors of the day. The system then sorts and formats the names and displays them on the welcome page. No visitors, it displays nothing.

But for now, let's keep it simple. Miss Moneypenny can simply refresh the browser every morning when she waters the plants.

So onward—

First off we need to define the visitor record.

Follow closely. We're going to skim over crucial Erlang concepts.

Review the following for nitty gritty details:

```
http://www.erlang.org/doc/reference_manual/records.html
http://www.erlang.org/doc/man/dets.html
```

Visitor record

First we need to drop into *~/nb/site/include*:

```
~/nb/site/src$ cd ../include
```

Now create a new file *nb.hrl*:

```
~/nb/site/include$ vim nb.hrl
```

Insert the following line:

```
-record(visitor, {date, time, name, company}).
```

And save the file.

We've just defined a visitor record.

So what's going on here? Our visitor record definition may be used in more than one module. So we've created a file, *nb.hrl*, that can be included in those modules. The *include* directory is an Erlang OTP convention designed for just this purpose. More here:

```
http://stackoverflow.com/questions/2312307/
      what-is-an-erlang-hrl-file
http://www.erlang.org/documentation/doc-5.2/
      doc/extensions/include.html
```

We have one more crucial step, so be patient. We need to create a set of functions that enable us to create and retrieve visitor records. Let's do this in a module called *visitors_db.erl*:

```
~/nb/site/src$ vim visitors_db.erl
```

Now insert the following:

```
%%% ---------------------------------------------------
%%% @author Lloyd R. Prentice and Jesse Gumm
%%% @copyright 2014 Lloyd R. Prentice and Jesse Gum
%%% @doc Visitor database functions
%%% @end
%%% ----------------------------------------------------
%% Store visitor data for nb
-module(visitors_db).
-compile(export_all).

-include("nb.hrl").
```

We'll add the create and retrieve functions later. But, for now, let's open up a **second** work terminal so we can play:

```
~/nb/site/include$ cd ../site
~/nb/site/$ erl -pa ebin
```

```
...
Running Erlang
Eshell V6.0  (abort with ^G)
1>
```

As you've guessed, the *erl -pa ebin* command opens the Erlang shell and points to the ebin directory.

To work with records in the shell, we must first read them in:

```
Eshell V6.0  (abort with ^G)
1> rr(visitors_db).
[visitor]
```

This command reads all the record definitions that have been included in the *visitors_db.erl* module.

Here's how we can examine the definition of visitor :

```
2> rl(visitor).
```

And we see:

```
-record(visitor,{date,time,name,company}).
```

This raises a question, however. How should we format date and time?

Erlang has a calendar library that can help. The following sequence of Erlang shell commands suggest how:

```
3> {Date, Time} = calendar:local_time().
{{2014,5,20},{16,6,57}}
4> Date.
{2014,5,20}
5> Time.
{16,6,57}
6>
```

As you can see, the command `{Date, Time} = calendar:local_time()` pattern matches the output of the Erlang function `calendar:local_time()`

to capture current date and time in the variables `Date` and `Time`. `Date` is represented as a tuple: `{Year, Month, Day}`; `Time` as: `{Hour, Minute, Second}`.

Note: The library module *calendar* is included in the *Erlang OTP STDLIB* application. It includes a number of useful date/time functions. More here:

    http://www.erlang.org/doc/man/calendar.html
    http://www.erlang.org/doc/man/STDLIB_app.html

That settled, let's create a record:

```
6> V1 = #visitor{date=Date, time=Time, name="Jesse James",
         company="Erlingo!"}.
#visitor{date = {2014,5,30}, time = {11,25,54},
         name = "Jesse James", company = "Erlingo!"}
```

We can retrieve data from this record in two ways.

By field:

```
7> V1#visitor.name.
"Jesse James"
```

Or, through pattern matching:

```
8> #visitor{date=Date1, time=Time1, name=Name,
         company=Company} = V1.
9> Date1.
{2014,5,30}
10> Time1.
{11,25,54}
11> Name.
"Jesse James"
12> Company.
"Erlingo!"
```

Yippy! We can now create records, stuff 'em with data, and pop it back out. Extra points if you can explain why we named the variables above *Date1* and *Time1* rather than *Date* and *Time*. Hint:

    http://www.theerlangelist.com/2013/05/working-with-immutable-data.html

But where should we store our records?

Erlang delivers just the ticket— *dets*.

Persistence

Dets stands for *Disk Erlang Term Storage*.

    http://www.erlang.org/doc/man/dets.html

An *Erlang term* is any data item. So *dets* helps us store any Erlang data item to disk.

Let's explore. First let's create a visitors database, specify the key position, and table type:

```
13> dets:open_file(visitors, [{keypos,#visitor.date},
        {type,bag}]).
{ok,visitors}
```

So what's this `{type,bag}` bit? We could tell you, but then we'd have to shoot you. Hint: check the dets man page— link noted above.

Now we can play:

```
14> dets:insert(visitors, V1).
ok
15> dets:lookup(visitors, Date).
[#visitor{date = {2014,5,30}, time = {11,25,54},
      name = "Jesse James", company = "Erlingo!"}]
16> V2 = #visitor{date=Date,time={12,20,11},
name="Rusty Scupper",company="Erlingo!"}.
17> dets:insert(visitors,V2).
ok
18> V3 = #visitor{date={2014,6,1}, time={14,0,0},
          name="Joe Armstrong"}.
#visitor{date = {2014,6,1},
time = {14,0,0},
name = "Joe Armstrong",
company = undefined}
```

Excelente!

Let's see who's coming in on May 30, 2014:

```
19> dets:lookup(visitors, Date).
[#visitor{date = {2014,5,30},
time = {11,25,54},
name = "Jesse James",
company = "Erlingo!"},
```

```
#visitor{date = {2014,5,30},
time = {12,20,11},
name = "Rusty Scupper",
company = "Erlingo!"}]
```

Ah, a list of two visitors, Jesse and Rusty.

Close out the database for now.

```
20> dets:close(visitors).
```

Formatting the visitor data

It'll pay us to think through one thing before we go further: how to format our data.

Note— our May 30, 2014 query returned two visitors.

For each record in the list, we need to extract the data and format it for suitable display. Let's focus first on how to extract data from a single record. Pattern matching serves us well here. Say we have the following record :

```
21> V4 = #visitor{date={2014,5,21}, time={14,58,03},
          name="Francesco Cesarini",
          company="Erlang Solutions"}.
```

We don't care about date and time since we're not going to display it, so we can format the name like this:

```
22> NN = V4#visitor.name.
"Francesco Cesarini"
23> CC. = V4#visitor.company.
"Erlang Solutions"
24> [NN," - ",CC].
"Francesco Cesarini, Erlang Solutions"
```

Wait a sec! What kind of cockamammy thing is going on in step 24?

Well, we can see that it's a list— a variable, a string, and another variable. But what good does that do us?

A bunch!

It's what's known in Erlang circles as an *i/o list* (see page 211 for more information). Erlang will conveniently instantiate the variables and concatenate all the terms for us when it sends it through standard output or a network socket. Saves programming hassle and considerable CPU cycles.

So, now, our database query returns a list of names. What next?

Erlang has a powerful tool for processing elements of a list— the *list comprehension*:

```
http://www.erlang.org/doc/programming_examples/
     list_comprehensions.html
```

Think about what we need to do:

- Extract every item in a list
- Format each item as it comes off the list
- Push the formatted value onto a new list

So lets look at how to do this with our list of visitors using an Erlang list comprehension:

```
25> Formatted_Visitors = [format_visitor(Visitor) ||
      Visitor <- Visitors]
    [["Jesse James"," - ","Erlingo!"],
    ["Rusty Scupper"," - ","Erlingo!"]]
26>
```

Read the list comprehension right-to-left and it should be transparent. Note the result— two i/o lists.

So now, we need to wrap up what we've learned.

Visitors database

So now we turn our attention back to the visitors database. Open a file called visitors_db.erl

```
~/nb/site/src$ vim visitors_db:erl
```

And enter the following:

```
%%% ----------------------------------------------
%%% @author Lloyd R. Prentice and Jesse Gumm
%%% @copyright 2014 Lloyd R. Prentice and Jesse Gum
%%% @doc Visitor database functions
%%% @end
%%% -----------------------------------------------
-module (visitors_db).
-compile(export_all).
-include("nb.hrl").
%% ------------------------------------------------
%% Visitors: Exported functions
%% ------------------------------------------------
%% @doc Open the visitors database
open_visitors_db() ->
   File = visitors,
   {ok, visitors} = dets:open_file(File,
        [{keypos,#visitor.date}, {type,bag}]).
```

```
%% @doc Close the visitors database
close_visitors_db() ->
   ok = dets:close(visitors).
%% @doc Enter VIP visiting the very day
put_visitor(Record) ->
    open_visitors_db(),
    ok = dets:insert(visitors, Record),
        close_visitors_db().
%% @doc Enter VIP visiting this very day
put_vip(Name, Company) ->
    {Date, Time} = calendar:local_time(),
    Record = #visitor{date=Date, time=Time,
        name=Name, company=Company},
    put_visitor(Record).
%% @doc Get visitors coming by today
get_visitors(Date) ->
    open_visitors_db(),
    List = dets:lookup(visitors, Date),
    close_visitors_db(),
    List1 = lists:sort(List),
    List1.
%% @doc Dump the db; handy for debugging
dump_visitors() ->
    open_visitors_db(),
        List = dets:match_object(visitors, '_'),
    close_visitors_db(),
        List.
```

```
%% @doc Pretty print visitor by name, company, or both
format_name(#visitor{name=Name, company=""}) -> Name;
format_name(#visitor{name="", company=Company}) ->
      Company;
format_name(#visitor{name=Name, company=Company}) ->
      [Name," - ",Company].
```

Not much new here. Note the *@doc* comments. The Erlang documentation utility *edoc* reads these to create beautiful documentation. Also note the function `format_name/1`. It employs pattern matching to deal with three possible user inputs:

- name only
- company only
- name and comany

Visitors admin

We need now a form to enter visitors into our visitors' database. Let's create that in a file called *visitors_admin.erl:*

```
~/nb/site/src$ vim visitors_admin:erl
```

Here's the code:

```
%%% ---------------------------------------------------
%%% @author Lloyd R. Prentice and Jesse Gumm
%%% @copyright 2014 Lloyd R. Prentice and Jesse Gum
%%% @doc Visitors admin page
%%% @end
%%% ---------------------------------------------------

-module (visitors_admin).
-compile(export_all).

-include_lib("nitrogen_core/include/wf.hrl").
-include("nb.hrl").
main() -> #template { file="./site/templates/bare.html" }.

title() -> "Visitor Admin".

body() ->
    #panel{id=inner_body, body=inner_body()}.

inner_body() ->
    %% We use defer here because this could
    %% potentially be during a redraw. We want to
    %% ensure the validators are attached *after*
    %% the form is drawn
    wf:defer(save, name, #validate{validators=[
        #is_required{text="Name or Company is
             required", unless_has_value=company}
   ]}),
wf:defer(save, date1, #validate{validators=[
    #is_required{text="Date is required"}
]}),
[
        #h1{ text="Visitors" },
        #h3{text="Enter appointment"},
        #label {text="Date"},
        #datepicker_textbox{
            id=date1,
            options=[
                {dateFormat, "mm/dd/yy"},
                {showButtonPanel, true} ]
        },
        #br{},
        #label {text="Time"},
        time_dropdown(),
        #br{},
        #label {text="Name"},
        #textbox{ id=name, next=company},
        #br{},
        #label {text="Company"},
        #textbox{ id=company},
        #br{},
        #button{postback=done, text="Done"},
        #button{id=save, postback=save, text="Save"}
    ].

time_dropdown() ->
    Hours = lists:seq(8,17),
        #dropdown {id=time, options=
            [time_option({H,0,0}) || H <- Hours]}.
    time_option(T={12,0,0}) ->
        #option{text="12:00 noon", value=wf:pickle(T)};
    time_option(T={H,0,0}) when H =< 11 ->
        #option{text=wf:to_list(H) ++ ":00 am",
            value=wf:pickle(T)};
    time_option(T={H,0,0}) when H > 12 ->
        #option{text=wf:to_list(H-12) ++ ":00 pm",
            value=wf:pickle(T)}.
    parse_date(Date) ->
        [M,D,Y] = string:tokens(Date, "/"),
        {wf:to_integer(Y), wf:to_integer(M),
            wf:to_integer(D)}.
```

```
event(done) ->
    wf:wire(#confirm{text="Done?", postback=done_ok});
event(done_ok) ->
    wf:redirect("/");
event(save) ->
    wf:wire(#confirm{text="Save?", postback=confirm_ok});
event(confirm_ok) ->
    save_visitor(),
    wf:wire(#clear_validation{}),
    wf:update(inner_body, inner_body()).

save_visitor() ->
    Time = wf:depickle(wf:q(time)),
    Name = wf:q(name),
    Company = wf:q(company),
    Date = parse_date(wf:q(date1)),
    Record = #visitor{date=Date, time=Time, name=Name,
                      company=Company},
    visitors_db:put_visitor(Record).
```

The functions `main/0`, `title/0`, `body/0`, and `inner_body/0` should look comfortably familar. If not, look back at *index.erl*. The lesson here is that Nitrogen forms are as easy to create as straight HTML— just different elements.

The two `wf:defer/3` functions at the top `inner_body/0` set up form-field validation. More here:

```
http://nitrogenproject.com/doc/api.html#sec-4
```

`wf:defer?`
It's worth mentioning `wf:defer/N` is what's considered a sibling to `wf:wire/N` (ther other sibling being `wf:eager/N`). Both functions wire actions to the browser. The difference here is that any actions wired with defer will execute after any other actions wired with wire. Because our code is destroying and redrawing form after every save (with the wf:update call), we need to re-wire the validators. And because the validators are wired inside the `inner_body()` function, when all is said and done, if the validators were wired with `wf:wire`, the validators would be wired before the update would be wired.
It's all a little advanced for this early, so just go with us here. We're revisit `wf:eager`, `wf:wire`, and `wf:defer` later.

Also note the Erlang function— `time_dropdown/0`. Look closely at the list comprehension in `time_dropdown/0`. What's that about? We leave that to your brilliance. Hint:

`http://www.erlang.org/doc/programming_examples/list_comprehensions.html`

The function `time_option/1` demonstrates again pattern matching on function parameters. It also introduces a new Erlang concept— *guard sequences.* Guard sequences are quite handy. For details drop down to section 8.24 in the *Erlang Reference Manual User's Guide*:

`http://erlang.org/doc/reference_manual/expressions.html`

While you're checking out guard sequences, study the rest of the *Erlang Reference Manual User's Guide* with great care. It will teach you much.

Note: We'll show you how to create a custom element for picking time in Chapter 15 to replace `time_dropdown/0`. Stick with us. Should be fun.

There's one more thing to observe with profit in *visitors_admin.erl*—*event/1*. Here's our very first *Nitrogen action* in the wild. Here's where we validate data entry and post data back to the server. Note first that *event/1* is pattern matching on two Erlang atoms— *save* and *confirm_ok*.

First, glance back up to the two button elements at the end of *inner_body/0*. These define the *Done* and *Save* buttons at the bottom of our form. *Done* simply redirects the page to *index.erl*. *Save* initiates a *postback* to *event(save)* which, in turn, brings up a confirmation dialog:

```
http://nitrogenproject.com/doc/actions/confirm.html
```

Note the postback in the *#confirm* element. Now, who in the world is it talking to?

Give yourself a big gold star if you say *event(confirm_ok)*.

Homework: What is *event(confirm_ok)* doing for us? If you're stumped, check out these links:

```
http://nitrogenproject.com/doc/api.html#sec-3
http://nitrogenproject.com/doc/actions/clear_validation.html
```

We have one more crucial task. Know what?

Way to go, Dude , you're way ahead of me— we need to display visitors-of-the-day on our welcome page.

Open index.erl and we'll make some major changes.

```
~/nb/site/src$ vim index.erl
```

When done, it should look like this:

```
%%% ---------------------------------------------
%%% @author Lloyd R. Prentice and Jesse Gumm
%%% @copyright 2014 Lloyd R. Prentice and Jesse Gum
%%% @doc nb home page
%%% @end
%%% ----------------------------------------------

-module (index).
-compile(export_all).

-include_lib("nitrogen_core/include/wf.hrl").
-include("nb.hrl").

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

title() -> "Welcome Board".

body() ->
    Visitors = visitors_db:get_visitors(date()),
   [
      #h1{ text="WELCOME!" },
      ~~#h2 { text="Joe Armstrong" },~~
      ~~#h2 { text="Rusty Klopaus"~~
      #list{numbered=false, body=
          format_visitors(Visitors)},
      #br{}
   ].

format_visitors(List) ->
    [format_visitor(X) || X <- List].

format_visitor(Visitor) ->
    Name = visitors_db:format_name(Visitor),
    #listitem{text=Name, class="visitors"}.
```

This might not be a bad time to check our progress. Open the browser to:

```
localhost:8000/visitors/admin
```

If all looks well, fill in the appointment form and repoint your browser to:

```
localhost:8000
```

3.12. Styling

Don't know about you, my friend, looks good but the composition of the welcome screen offends me. We can fix that with a few judicious CSS statements:

```
~/nb/site/src$ cd ../static/css
~/nb/site/static/css$ vim style.css
```

Now, in the h1 declaration, change:

```
font-size: 1.875em;
line-height: 1.066667em;
margin-top: 1.6em;
margin-bottom: 1.6em;
```

To:

```
font-size: 3em;
line-height: 1.066667em;
margin-top: 0.2em;
margin-bottom: 0.6em;
```

And add the following declarations:

```
li.visitors {
    font-size: 2em;
    line-height: 0.2em;
    margin-top: 1em;
    margin-bottom: 1em; }

li.associates {
    font-size: 1em;
    line-height: 0.2em;
    margin-top: 1em;
    margin-bottom: 1em; }
```

Refresh your browser.

```
localhost:8000
```

Not too shabby, eh?

3.13. Debugging

If the form had not shown up in the admin page, the problem would most likely be in *visitors_admin.erl*. If the entry to the appointment form had not show up, we'd have been in for stint of debugging— mainly check over our our code in *visitors_db.erl*, and *visitors_admin.erl* meticulously. We could also test exported functions in those modules by running them in the Erlang shell. Here's how:

```
(nitrogen@127.0.0.1)14> l(visitors_db).
```

This loads the module visitors_db.erl into the Erlang shell.

```
(nitrogen@127.0.0.1)14> visitors_db:dump_visitors().
```

You should see a record depicting the appointment you entered. If not, check the functions in *visitors_db.erl* with great care.

3.14. What you've learned

So, good chum, you've employed and seen in action 18 Nitrogen elements:

- #template

- #container
- #grid_8
- #h1
- #h3
- #list
- #br
- #listitem
- #panel
- #validate
- #is_required
- #label
- #datepicker_textbox
- #button
- #dropdown
- #option
- #confirm
- #clear_validation

You've created Nitrogen forms and used Nitrogen events. You've created an Erlang dets database and learned a smattering of Erlang along the way.

Good day's work!

3.15. Think and do

Deploy and bring up nitroBoard on a local network so Miss Moneypenny can enter VIP visitor's from her desk.

4. nitroBoard II

Miss Moneypenny wants an associates directory under the VIP list.

Easy enough, but here's the tricky thing— she want's the listing to show whether the associate is in or out.

So, we need an associates database. Check. The associate record needs a field to denote in or out. Check. But who toggles it? When and how?

OK, say we create a module called *iam.erl*. When associate enters *localhost:8000/iam* into the browser address bar, they will:

- choose from a list which associate they are
- toggle their *in* or *out* status.
- redirects back to the welcome page.

Makes sense. Let's get to it.

4.1. Plan of attack

1. Data persistence
   a) associates db
2. Develop admin pages
   a) associates form
3. Update welcome page

    a) display associates

4. Style

5. Test/debug/revise

4.2. Associates

Let's dive into the associates database first. Pretty much same-ol' same-ol'.

Associate record

Drop into *~/nb/site/include* and create a new file *nb.hrl*:

```
~/nb/site/src$ cd ../include
~/nb/site/include$ vim nb.hrl
```

Insert the following line:

```
-record(associate, {lname, fname, ext="", in=true}).
```

Save the file.

Associates database

Now, on to the associates database. Open a file:

```
~/nb/site/src$ vim associates_db:erl
```

And enter this code:

```
%%% -----------------------------------------------
%%% @author Lloyd R. Prentice and Jesse Gumm
%%% @copyright 2014 Lloyd R. Prentice and Jesse Gum
%%% @doc Associates data store
%%% @end
%%% -----------------------------------------------

-module (associates_db).
-export([
     put_associate/1,
     get_associate/1,
     get_associates/0,
     format_name/1,
     format_in_status/1]).

-include("nb.hrl").

%% ------------------------------------------------
%% Associates: Exported functions
%% ------------------------------------------------
```

```
open_associates_db() ->
    File = associates,
    {ok, associates} = dets:open_file(File,
       [{keypos,#associate.lname}, {type,set}]).


close_associates_db() ->
    ok = dets:close(associates).


put_associate(Record) ->
    open_associates_db(),
    ok = dets:insert(associates, Record),
    close_associates_db().

get_associate(LName) ->
    open_associates_db(),
    [Record] = dets:lookup(associates, LName),
    close_associates_db(),
    Record.


get_associates() ->
    open_associates_db(),
    List = lists:sort(dets:match_object(associates,
        '_')),
    close_associates_db(),
    List.

format_name(#associate{lname=LName, fname=FName,
      ext=Ext}) ->
    format_name(LName, FName, Ext).

format_name(LName, FName, []) ->
     [LName, ", ", FName];
format_name(LName, FName, Ext) ->
     Last_First = [LName, ", ", FName],
     [Last_First, " - ext: ", Ext].
```

Note that differences between *associates_db.erl* and *visitors_db.erl* are trivial.

But we do have an issue. We need an icon or other indicator to depict when associate is in or out.

So let's modify *format_name/1* and *format_name/3* and we'll also add a *format_in_status/1.*

```
format_name(#associate{lname=LName, fname=FName,
                       ext=Ext, in=In}) ->
    Status = format_in_status(In),
    format_name(LName, FName, Ext, Status).

format_in_status(true) -> "IN: ";
format_in_status(false) -> "OUT: ".
format_name(LName, FName, [], Status) ->
     [Status, LName, ", ", FName];
format_name(LName, FName, Ext, Status) ->
     Last_First = [LName, ", ", FName],
     [Status, Last_First, " - ext: ", Ext].
```

Associates admin

Now, following the example of visitors, we need an associates admin page:

```
~/nb/site/src$ vim associates_admin:erl
```

The code is similar to *visitors_admin.erl*:

```
%%% ----------------------------------------------------
%%% @author Lloyd R. Prentice and Jesse Gumm
%%% @copyright 2014 Lloyd R. Prentice and Jesse Gum
%%% @doc Associates directory admin page
%%% @end
%%% ----------------------------------------------------
-module (associates_admin).
-compile(export_all).
-include_lib("nitrogen_core/include/wf.hrl").
-include("nb.hrl").
main() -> #template { file="./site/templates/
      bare.html" }.
title() -> "Associates Admin".
body() ->
    wf:wire(save, lname, #validate{validators=
        #is_required{text="Last Name Required"}}),
    [
        #h1{ text="Associates Directory" },
        #h3{text="Enter directory listing"},
        #flash{},
        #label {text="LName"},
        #textbox{ id=lname, next=fname},
        #br{},
            #label {text="FName"},
            #textbox{ id=fname, next=email},
            #br{},
            #label {text="Extension"},
            #textbox{ id=ext},
            #br{},
            #button{id=save, postback=save, text="Save"},
            #link{url="/", text="Cancel"}
        ].
    clear_form() ->
        wf:set(lname, ""),
        wf:set(fname, ""),
        wf:set(ext, "").

    event(save) ->
        wf:wire(#confirm{text="Save?", postback=
            confirm_ok});
    event(confirm_ok) ->
        [LName, FName, Extension] = wf:mq([lname,
            fname, ext]),
        Record = #associate{lname=LName, fname=FName,
            ext=Extension},
        associates_db:put_associate(Record),
        clear_form(),
        wf:flash("Saved").
```

We do see one new element and three new Nitrogen API functions in this module however:

- `#flash{}`

- `wf:set/2`
- `wf:mq/3`
- `wf:flash/1`

The `#flash{}` element defines a placeholder for flash messages created by the *wf:flash/1* command.

```
http://nitrogenproject.com/doc/elements/flash.html
```

The API function *wf:set/2* should be obvious.

Getting up close and friendly with *wf:mq/3* and its kissing cousins *wf:q/1*, *wf:qs/1*, *wf:mqs/1*, *wf:q_pl/1*, and *wf:qs_pl/1* is well worth your time since these functions enable you to retrieve on the server side data posted back from client-side forms.

```
http://nitrogenproject.com/demos/postback2
```

Display associates

So what's missing?

Right on!

We still need to display associates on the welcome board. We bring up *index.erl*....

```
~/nb/site/src$ vim index.erl
```

And a few additions to *index.erl* should do the trick:

```
...
body() ->
  Visitors = visitors_db:get_visitors(date()),
  Associates = associates_db:get_associates(),
    [
      #h1{ text="WELCOME!" },
      #list{numbered=false,
        body=format_visitors(Visitors)},
        ~~#br{}~~
        #hr{},
        #h4{ text="Associates Directory"},
        #hr{},
        #list{numbered=false,
          body=format_associates(Associates)}
    ].
format_associates(List) ->
  [format_associate(X) || X <- List].
format_associate(Associate) ->
  Name = associates_db:format_name(Associate),
  #listitem{text=Name, class="associates"}.
```

Let's take a look at what we've achieved:

```
localhost:8000/associates/admin
```

Enter a few associate names, then cancel. If all is well, you should see the associates listed on the welcome board.

4.3. I am in/I am out

Our associates need to toggle their in/out status. We COULD do this by adding a function to *associates_admin.erl*. But putting the in/out status update function in a separate module gives us a more elegant, easier to remember, url.

So, let's create the module:

```
%%% ------------------------------------------------
%%% @author Lloyd R. Prentice and Jesse Gumm
%%% @copyright 2014 Lloyd R. Prentice and Jesse Gum
%%% @doc Toggle in field
%%% @end
%%% ------------------------------------------------
-module (iam).
-compile(export_all).
-include_lib("nitrogen_core/include/wf.hrl").
-include("nb.hrl").
main() ->
   #template { file="./site/templates/bare.html" }.
title() -> "I am...".
body() ->
    #panel{id=inner_body, body=inner_body()}.
```

```
inner_body() ->
    [
        #h1{ text="I am..." },
        associate_dropdown(),
        #button{id=toggle, postback=toggle,
text="Toggle"}
    ].
format_iam(List) ->
    [format_name(X) || X <- List].
format_name(Associate) ->
    #associate{lname=LName, fname=FName, in=In} =
       Associate,
    Status = db_associate:format_in_status(In),
    Name = [Status, LName, ",", FName],
    #option { text=Name, value=LName }.
associate_dropdown() ->
    Associates = associates_db:get_associates(),
    #dropdown{
        id=associate,
        value="",
        options=format_iam(Associates)
    }.
event(toggle) ->
    Associate = wf:q(associate),
    Record = associates_db:get_associate(Associate),
    wf:info("~w~n",[Record]),
        Record1 = toggle_status(Record),
        wf:info("~w~n", [Record1]),
        associates_db:put_associate(Record1),
        wf:redirect("/").

    toggle_status(Rec = #associate{in=Status}) ->
       Rec#associate{in = not(Status)}.
```

All the mysteries here are in *format_name/1*, but easy enough to tease out. First we pattern match on the associate record to extract name and status variables. Next we toggle the status flag and, finally concatenate the variables to form a dropdown option string.

Note to Erlang newbies: Traditionally Erlang represented *strings* as memory-hogging *lists*. This meant that the full fire-power of the list library could be brought to bear.

```
http://schemecookbook.org/Erlang/StringBasics
```

Since we're not concerned about memory, we use the list representation for option strings in *format_name/1*.

Erlang also offers binaries which are more compact internally, and anywhere we use strings, we *could* use binaries.

```
http://www.erlang.org/doc/efficiency_guide/binaryhandling.html
```

But, fact is, it's easy to convert between the two representations using the BIFs `list_to_binary/1` and `binary_to_list/1`.

```
http://www.erlang.org/doc/man/erlang.html#binary_to_list-1
http://www.erlang.org/doc/man/erlang.html#list_to_binary-1
```

Nitrogen also offers some convenience methods `wf:to_list/1` and `wf:to_binary/1`, which are a bit more flexible than Erlang's BIFs `list_to_binary/1` and `binary_to_list`/1.

4.4. Styling

Let's tweak the styling:

```
~/nb/site/src$ cd ../static/css
~/nb/site/static/css$ vim style.css
```

Add the following declarations:

```
li.visitors {
    font-size: 2em;
    line-height: 0.2em;
    margin-top: 1em;
    margin-bottom: 1em; }
```

```
li.associates {
    font-size: 1em;
    line-height: 0.2em;
    margin-top: 1em;
    margin-bottom: 1em; }
```

Refresh your browser.

```
localhost:8000
```

Not too shabby, eh?

4.5. What you've learned

So what think you, Dude? May have been a “boring” project, but *nb* had much to teach, wouldn't you say:

- Introduction to Nitrogen elements and actions
- How to build Nitrogen pages
- How to build forms
- Form-field validation
- Postbacks
- Simple data storage and retieval:
- A taste of Erlang

4.6. Think and do

Suppose a VIP Visitor wants to leave a message for an associate who is out. How would you modify nitroBoard to cover that case?

5. A Simple Login System

Before moving onto the next project, we're going to go on a tangent to help you with a quick login system.

Most web applications will require the ability to log in. In this tangent, we'll give you a quick overview of how to implement a simple login system. For this example, we're going to build a straight login system, something self-contained that doesn't rely on other web services (like Facebook, Twitter, or Google). To keep it simple, we're going to create a brand new application just for this.

Also, as we've only covered DETS for storage, we're going to continue to use DETS for this chapter. In the coming chapters, we'll focus more on other databases.

5.1. Getting Started

You still have the original Nitrogen project directory in `~/nitrogen`, right? Let's head back there and make a new project.

```
$ cd ~/nitrogen
$ make slim_inets PROJECT=nitro_login
...
**********************************************
Generated a slim-release Nitrogen project
in ../nitro_login, configured to run on inets.
**********************************************
$ cd ../nitro_login
```

We went with the "Slim" release here just to make it faster.

5.2. Dependencies

Before we can continue, we will want to include a decent hashing mechanism[1], and also ensure that the underlying hashing application is started.

5.2.1. Rebar Dependency: `erlpass`

The first step here will be adding the `erlpass` application as a rebar dependency. Let's open up your application's `rebar.config` and add the following line to the `deps` section:

[1]More about proper password hashing and security on page 138

```
$ vim rebar.config
...
{deps, [
   {erlpass, "", {git, "git://github.com/ferd/erlpass",
                 {branch, master}}},
   ...
]}.
```

Then we want to make sure we start the `bcrypt` app, which is actually a dependency of `erlpass`. The simplest method would be to launch those two apps from the command line. Edit the `etc/vm.args` file and add the following line:

```
-eval "application:start(bcrypt)"
```

(Yes, there are better ways to start an app like this, but this is the quick way to get it done).

Finally, let's run "make" again, to bring in the erlpass and its dependencies

```
$ make
```

5.3. The `index` page

We're going to make a very simple index page that informs us of our logged-in status. If we are logged in, it will present a logout link. Otherwise, it will give us a link to a login form or an account creation form.

For example, we could do something like:

```
$ vim site/src/index.erl
-module(index).
-compile(export_all).
-include_lib("nitrogen_core/include/wf.hrl").

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

body() ->
  case wf:user() of
    undefined ->
      [
        "Not Logged in.",
        #br{},
        #link{text="Log In", url="/login"},
        " or ",
        #link{text="Create an Account", url="/create_account"}
      ];
    Username ->
      [
        #span{text=["Logged in as ", Username,". "]},
        #br{},
        #link{text="Log out", postback=logout}
      ]
  end.

event(logout) ->
  wf:logout(),
  wf:redirect("/").
```

This simple module does one primary thing: Checks if we're logged in or out, and if we're logged out, links us to the login screen, othewise, it gives us a link to log out.

Alright, our index page is created, let's build it and fire up Nitrogen right away to make sure everything is good to go. If you still have Nitrogen from the previous chapter, you should probably kill it right now with `q()`. Otherwise, you'll get port-conflict errors (You could also change the port in `etc/simple_bridge.config`).

Also, since we're going to be doing live-coding from here on out, let's run `sync:go()` once Nitrogen is started.

```
$ make
$ bin/nitrogen console
1> sync:go().
Starting Sync (Automatic Code Compiler / Reloader)
Scanning source files...
Growl notifications disabled
ok
```

Go ahead and fire up your browser and navigate to **http://localhost:8000**

You should be greeted with a simple menu.

5.4. Creating an account

Now that we have our index page, we want to add a page to create our account. Let's get the boilerplate out of the way:

```
$ vim site/src/create_account.erl
-module(create_account).
-compile(export_all).
-include_lib("nitrogen_core/include/wf.hrl").

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

title() -> "Create an account!".
```

Now let's add the registration form:

```
body() ->
  [
    #label{text="Username"},
    #textbox{id=username},
    #label{text="Password"},
    #password{id=password},
    #label{text="Confirm Password"},
    #password{id=password2},
    #br{},
    #button{text="Save Account", postback=save}
  ].
```

Finally, let's handle the postback generated by the "Save Account" button.

```
event(save) ->
  [Username, Password] = wf:mq([username, password]),
  ok = db_login:create_account(Username, Password),
  wf:user(Username),
  wf:redirect("/").
```

So far, this should be pretty straightforward: We have a basic form that presents a username and a password, and we have a postback that retrieves the username and password, then calls a `create_account/2` function in a `db_login` module (which we haven't made yet, but we will in a moment). This function gives us a `Userid`, and we store the `Userid` in the session with `wf:user/1` and finally redirect back to the homepage.

5.4.1. `db_login` module

We need to create a module called `db_login` that interfaces with our login database, and which also interfaces with our `erlpass` application.

Let's get the boilerplate out of the way.:

```
$ vim site/src/db_login.erl
-module(db_login).
-export([
    create_account/2
]).
open_db() ->
    Options = [{type,set}],
    {ok, logins} = dets:open_file(logins, Options).
close_db() ->
    ok = dets:close(logins).
```

So far, all we've done is create the module, export the create_account/2 function, set up a login record to normalize the storage in DETS, and set up some basic, whi open and close operations for DETS.

Now it's time to write our create_account/2 function. I'll take it slowly for you so it's clear what we're doing and why:

```
create_account(Username, Password) ->
   open_db(),
   PWHash = erlpass:hash(Password),
```

Alrighty, so far, all we've done is open the database, and then hashed the password. We will end up storing this hashed password as our database. For more explanation, see page 138.

```
   dets:insert(logins, {Username, PWHash}),
   close_db(),
   ok.
```

The rest of this should be straightforward: we store the username and hash in DETS (with the table key being Username), close DETS, and return ok. In a real application, you would want to check to make sure that username doesn't already exist, and return an error if it does, but for the sake of our demo, we won't worry about that.

For the sake of clarity, here's the full create_account/2 function:

```
create_account(Username, Password) ->
   open_db(),
   PWHash = erlpass:hash(Password),
   dets:insert(logins, {Username, PWHash}),
   close_db(),
   ok.
```

At this point, account creation works. Navigate to **`http://localhost:8000`** and click the "Create an Account" link. You should be presented with a new account form, and will be able to fill it out to attempt to log in.

5.5. The login form

Now it's time to create the actual login form everyone knows and loves. All we need is a username, password, and "Login" button:

```
$ vim site/src/login.erl
-module(login).
-compile(export_all).
-include_lib("nitrogen_core/include/wf.hrl").

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

title() -> "Log In".

body() ->
  [
    #label{text="Username"},
    #textbox{id=username},
    #label{text="Password",
    #password{id=password},
    #br{},
    #button{text="Log In", postback=login}
  ].
```

So far, it's been pretty straightforward: we made a module that has a simple login form. Now we just need to handle the postback `login`.

```
event(login) ->
  [Username, Password] = wf:mq([username, password]),
  case db_login:attempt_login(Username, Password) of
    true ->
      wf:user(Username),
      wf:redirect("/");
    false ->
      wf:wire(#alert{text="Invalid Username or Password"})
  end.
```

Here we can see that we expect the `db_login:attempt_login/2` function to return a boolean `true` or `false`, depending on the validity of the login information provided. If it succeeds, we store the `Username` in the session, and redirect back to the home page, otherwise, let's give an error and let the user correct their input.

5.5.1. Verifying the password (back to `db_login`)

And so we need to head back to `db_login` to add the newly referenced `attempt_login` function (don't forget to `export` it as well).

```
$ vim site/src/db_login.erl
-export([
  create_account/2,
  attempt_login/2
]).
...
attempt_login(Username, Password) ->
  open_db(),
  Result = case dets:lookup(logins, Username) of
    [] -> false;
    [{_, PWHash}] ->
      erlpass:match(Password, PWHash)
  end,
  close_db(),
  Result.
```

So once again, we start by opening the table. Then we use the provided username as the key to lookup if there are any records with that username. If it returns an empty list, then we don't have any matching usernames in the database, and we can just return false. If it returns a single-element list, we use pattern matching to pull out the PWHash. Then compare the provided password's hash against the password hash that was stored with the username when it was created with `erlpass:match/2`.

5.6. Some finishing touches

At this point, you should be able to create an account, log in, and log out.

But there are a few critical things missing, particularly validation on our forms: verifying the provided passwords are the same, as well as verifying that a username isn't already taken.

So let's add that necessary validation.

5.6.1. Adding Validation to the Login Form

The easy one is on the login form, so we'll do that first. We just want to make sure we require both fields.

```
$ vim site/src/login.erl
...
body() ->
  wf:defer(login, username, #validate{validators=[
      #is_required{text="Username Required"}]}),
  wf:defer(login, password, #validate{validators=[
      #is_required{text="Password Required"}]}),
  [
    #label{text="Username"},
    ...
    #br{},
    #button{id=login, text="Log In", postback=login}
  ].
...
```

Notice the `wf:defer`[2] commands inserted before the form itself. These will add validators to the fields with the id set to **`username`** and **`password`**, and the trigger being the button with the ID set to **`login`**. Notice then we added **`id=login,`** to the **`#button{}`**.

Go ahead and check it out on the browser. The login form will now only

[2]We use `wf:defer` instead of `wf:wire` as a convention. `wf:defer` ensures that the validators are sent to the browser last. In the situation above, it wouldn't make a difference, but if reworked the form to dynamically add or remove fields, we would want to make sure the validators are sent last (to make sure that the newly generated fields exist on the page before we try to attach validators to them).

work if you provide a username and a password, otherwise it'll give you some validation messages next to the input fields.

5.6.2. Adding Validation to Account Creation

Now it's time to add validation to the account creation page.

We want to validate a few things:

1. The username, password, and password confirmation are all provided

2. The password and password confirmations match.

3. The username does not already exist in our system

Conveniently, items 1 and 2 are simple. Item 3 will require a little extra work. We'll do 1 and 2 first.

```
$ vim site/src/create_account.erl
...
body() ->
  wf:defer(save, username, #validate{validators=[
      #is_required{text="Username Required"}]}),
  wf:defer(save, password, #validate{validators=[
      #is_required{text="Password Required"}]}),
  wf:defer(save, password2, #validate{validators=[
      #confirm_same{text="Passwords do not match",
                    confirm_id=password}]}),
  [
    #label{text="Username"},
    ...
    #br{},
    #button{id=save, text="Save Account", postback=save}
  ].
...
```

So this pretty closely models what we did with the login form, but with the addition of comparing the password and confirmation password. We attach a `#confirm_same{}` validator to the `password2` field, and tell it to compare against the field called `password`.

You might notice that we didn't even attach an `#is_required{}` validator to the password confirmation form. This is not an error. Since the password field is already required, if the password confirmation field is left blank, the validator will fail with "Passwords do not match."

Now let's add the validator to make sure the provided username isn't taken. We need to check the database for the username first, so let's open up db_login right away and make a `username_exists/1` function. We'll end up using that in our `create_account` module.

```
$ vim site/src/db_login.erl
-module(db_login).
-export([
  create_account/2,
  attempt_login/2,
  username_exists/1
]).
...
username_exists(Username) ->
  open_db(),
  Exists = case dets:lookup(logins, Username) of
    [] -> false;
    _ -> true
  end,
  close_db(),
  Exists.
```

This should look strikingly familiar to the existing `attempt_login/2` function. Indeed, it has many similarities structurally. In this case, all we're doing is getting a list of records with the key set to `Username`. If it returns an empty list, and we return `false`. Any other return value then the username does exist in the database, so we return `true`.

You might even recognize that the whole `case` expression could be replaced with a one-liner like this:

```
Exists = dets:lookup(logins, Username) =/= [],
```

Which means "if the lookup returns anything other than an empty list, then the username exists." Feel free to use either, but I use the case statement for the sake of obviousness.

Finally, let's add the actual validation to our create_account page. For this, we need to use a custom validator, conveniently named `#custom{}`. Here, we'll be providing a function for the validator to call.

```
$ vim site/src/create_account.erl
...
is_username_available(available, Username) ->
  not(db_login:username_exists(Username)).
body() ->
  wf:defer(save, username, #validate{validators=[
      #is_required{text="Username Required"},
      #custom{text="Username is taken",
              function=fun is_username_available/2,
              tag=available}]}),
  wf:defer(save, password, #validate{validators=[
      #is_required{text="Password Required"}]}),
  wf:defer(save, password2, #validate{validators=[
      #confirm_same{text="Passwords do not match",
                    confirm_id=password}]}),
  ...
```

We've only added a few lines here. The `#custom{}` validator needs a function (with arity 2), which returns `true` if the validator succeeds and `false` if the validator fails. In our case, we assign the function to `fun is_username_available/2`, which we've defined right in our module. The `tag` is a common convention within Nitrogen for identifying things. In our case, we assigned it to the atom `available`, and matched again on it in the first argument to `is_username_available/2`.

Go ahead, load up the page and try everything out. You will no longer be able to create an account with the same name as an existing account, and all fields will be validated accordingly.

5.7. Closing thoughts on login systems

No doubt, there are still things missing from this: most notably password changing and password recovery. Password changing is relatively easy, especially with `erlpass:change/3` and `erlpass:change/4` functions. I leave that exercise up to the reader.

Password recovery is a harder problem because you have to deal with email, and is beyond the scope of this tiny chapter. We will deal with email later in the book.

You can also tie your system to third-party login systems like Google, Facebook, Twitter and the like, but for most business applications, you're likely to want to keep your password and user management system internal.

5.8. Links

- The complete code for nitro_login:
  `https://github.com/choptastic/nitro_login`

6. A Tale of Three Backends

7. nitroProjectLog

Rush job!

Ain't they all.

Client needs a project log so team can post key points of progress and store project documents in Dropbox-like storage.

Specs?

You kidding?

Brainstorming time!

OK, project team is scattered around the world. How about a blog-like interface to maintain key points of progress?

Great idea. Post date, time, and name of team member at top of each post.

These project documents? How big?

Blobs, hey— Big. Gotcha. How many?

Yikes!

How critical?

Bet-the-company critical, you say? Sounds like a job for a distributed db like leoFS.

We'll need a good index.

Yeah, and authenticated access.

My suggestion? Let's whip up a prototype for client approval before we make it pretty.

Plan of attack

1. Create new project

2. Page templates
3. User access
4. Review log
5. Log entry
6. Review document index
7. Enter document
8. Retrieve document

We're on it

- Design and specs
- Uploading files
- Postbacks
- Comet/Websockets
- Simple Comet
- Comet Pools
- Working with the processes
- Sessions and State Tracking
- Sessions
- Authentication
- Page State

- Binding
- Basic Binding
- Advanced Binding
- Postbacks with Binding
- When not to use Binding
- Working with Databases
- REST interface
- REST handler
- leoFS

Notes

SKILLS

- log-in/user validation - user role enforcement - file upload - image validation - storing image data - tag-based lookup - db facade

USE CASES

1) Software developers need to coordinate and track progress with off-site developers.

Developers can enter and edit reports and upload text, *.jpg, and*.pdf documents Clients can access and read reports and documents

2) International consulting firm needs to coordinate and track progress of collaborative projects

Consultants can enter and edit reports and upload text, *.jpg, and*.pdf documents Clients can access and read reports and documents

ROLES

– project manager - can grant access privileges – consultant - read/write/upload privileges – client - read-only privilege

LOG-IN

– verify credentials – assign access privileges

**** I will need help with roles and log-in

READ

– select timeframe; e.g. date 1 through date 2 – see titles/abstracts of entries over timeframe – select item to read

– select tag – see tagged titles/abstracts by sorted by date – select item to read

WRITE

– date time – title – abstract – tags – item/report – upload document - text, *.jpg, *.pdf

**** I'm presuming that we'll use tinyMCE to edit items/reports. Looks straight forward from your docs, but I may need help

**** Need to validate uploads: how do we assure images are valid?

**** It would be meat to factor this app into reusable components. I don't know the best way to do this. What do you think? My thoughts re factoring below:

- log-in and user validation - db facade to test alternative dbs – open – close – put – get – get-some – get-all - db - add-edit (tinyMCE)/file upload – need to validate input - select by date range - select by tag - display document – html, *.jpg, *.pdf

**** How do we display *.pdfs?

8. NitroMail

Let me see those specs—

Users: mail clients, marketing department
Data: up to 500,000 records;
- e-mail address
- include-in-list tags,
- record source,
- status flags, e.g. opt-in|opt-out, active|bounced

Mail client interface: "Add me to your mailing list"
Marketing interface:
- List maintenance; add, edit, delete
- Bounce management
- Edit mail text
- Prepare and launch mailing
- Mailing status dashboard

SMTP vendor interface: Mailgun

Looks like straight-forward CRUD with a few flourishes. So, Dude, can you cobble up a work plan?

Plan of attack

1. Pick a database

2. Create new project

3. Define records
4. Template user interfaces
5. Route
6. Integrate user interfaces with databases
7. Style
8. Test
9. Document

Off to the races

- Design and specs
- DB backends
- - MySQL (sigma_sql?)
- - PostgreSQL - Riak
- - CouchDB
- - MongoDB
- Drag and Drop
- Google Charts
- Tweaking the web server
- Making your own custom elements
- Making your own custom actions

- Custom Routing handler
- Working with Plugins
- - vagabond/gen-smtp
- Programming to an API
- Launching your own addons during Startup
- Using Nginx for running multiple instances
- Hacking on Nitrogen (your own fork)

Notes

SKILLS

- log-in/user validation - user role enforcement - file upload - image validation - storing image data - tag-based lookup - db facade

USE CASES

1) Software developers need to coordinate and track progress with off-site developers.

Developers can enter and edit reports and upload text, *.jpg, and*.pdf documents Clients can access and read reports and documents

2) International consulting firm needs to coordinate and track progress of collaborative projects

Consultants can enter and edit reports and upload text, *.jpg, and*.pdf documents Clients can access and read reports and documents

ROLES

– project manager - can grant access privileges – consultant - read/write/upload privileges – client - read-only privilege

LOG-IN

– verify credentials – assign access privileges

**** I will need help with roles and log-in

READ

– select timeframe; e.g. date 1 through date 2 – see titles/abstracts of entries over timeframe – select item to read

– select tag – see tagged titles/abstracts by sorted by date – select item to read

WRITE

– date time – title – abstract – tags – item/report – upload document - text, *.jpg, *.pdf

**** I'm presuming that we'll use tinyMCE to edit items/reports. Looks straight forward from your docs, but I may need help

**** Need to validate uploads: how do we assure images are valid?

**** It would be meat to factor this app into reusable components. I don't know the best way to do this. What do you think? My thoughts re factoring below:

- log-in and user validation - db facade to test alternative dbs – open – close – put – get – get-some – get-all - db - add-edit (tinyMCE)/file upload – need to validate input - select by date range - select by tag - display document – html, *.jpg, *.pdf

**** How do we display *.pdfs?

Part III.

Core Nitrogen Concepts

9. How to choose a webserver

Cowboy is the newest popular webserver on the Erlang scene. Created by Loic Hoguin, and under heavy development, it currently supports SPDY, WebSockets (which Nitrogen will soon support), and follows a model that differentiates itself from the other Erlang webservers: It doesn't maintain a process dictionary. Instead, it follows a more 'pure' flow, by passing around the a request object to each of Cowboy's functions. It's a lightweight and high performance webserver, and very capable.

Inets is the built-in Erlang Webserver. It's lightweight, and gets the job done. Its biggest benefit is that it does not require any external dependencies, since it's a part of the standard Erlang release. If you have a simple app that doesn't require much by way of high performance, Inets will work.

Mochiweb is a simple webserver that sticks mostly to maintenance releases these days, new features are no longer being added, so it seems Mochiweb will forever not support Websockets. But it is a perfectly performant server for most web application uses.

Webmachine is a webserver with an emphasis on creating great RESTful APIs. The perfect use-case for using Webmachine with Nitrogen would be to provide a powerful RESTful API for external apps (like mobile device APIs), while also providing an interactive interface utilizing Nitrogen for web users.

Yaws is the granddaddy of Erlang webservers. It's tried and true, has been around for many years, and continues to improve, including adding Websocket support. It has solid performance metrics, and has a configuration system modelled after Apache's.

IN GENERAL if you are completely unsure which webserver to use, we recommend choosing either *Cowboy* or *Yaws* for production environments, as both provide quality performance, support websockets (which Nitrogen is slated to support as of version 2.3), and both handle large files smoothly.

10. How to structure your Nitrogen projects

11. Nitrogen Functions

12. Nitrogen Elements

13. Nitrogen Actions

- Wiring
- ...to the page
- ...to an element
- ...to a bunch of elements
- ...to another element triggered by another element

14. Nitrogen Templates

15. Anatomy of a page and the path of execution

16. Comet and long-running processes

17. Secure your site

Jesse munching on a pastrami sandwich:

Securing web applications in any framework requires employing a number of standard practices.

Trust No One!

As displayed before most *X-Files* episodes: “Trust No One!”

Any data sent from the client cannot *in any way* be trusted: form inputs (POST Variables), URL paths and Query Strings (any characters found after a question mark in the URL), even headers - all can be spoofed quite trivially.

Conveniently, Nitrogen cryptopgraphically signs and validates postback requests, preventing CSRF (Cross-Site Request Forgery) attacks. If a postback request is made, but Nitrogen fails to decode the page_state token, the `event/1` function on your page or element will never be executed.

But while the signed postbacks close that particular attack vector, that still leaves plenty for us to do:

Prevent HTML injection

One of the most common methods for attacking a web application (popular with spammers) is by attempting to inject HTML into fields in hopes that it’s not sanitized and entered directly into the database.

But Jesse, who cares if someone enters HTML into the database? HTML is just text!

In many cases, it might be innocuous, like adding `<b>` or `<i>` tags for formatting, or making a friendly link to their resume, rather than pasting in a long and ugly URL. But someone malicious might inject an image known to exploit a weakness in the browser, providing the ability to run anything an attacker might want. As an example, see http://j.mp/remotecode.

Another, more obvious attack would be the injection of a `<script>` tag.

Consider a forum. If a user can successfully get...

```
<script>location.href="http://evil-forum-clone.com/
      fakethread"</scipt>
```

...to other clients, then every user who opened the post by the malicious user would be redirected to the attacker's fake site, which, incidentally is a visual clone of the forum, in which the user is perpetually logged out. Perhaps the user wishes to reply to the fake post, not noticing that the domain name has changed – it looks the same after all! So the user clicks the "Sign In" link, which takes you to a perfectly legitimate-looking login screen and attempts to log in by entering their email address and password.

BOOM! YOU'VE JUST BEEN COMPROMISED.

Since you were on the malicious page, you've just sent your email address to the malicious site. To be extra sneaky about it, the fake page could, upon log-in form submission, immediately redirect back to the legitimate site's log-in form (since the log-in itself will fail). At that point, the user would be completely unaware that anything malicious happened, and that their account information has already been harvested.

This is the kind of danger present to your users if you fail to sanitize your data from HTML or JavaScript injection.

Now that you know *why* you should sanitize your data, we will show you *how* to sanitize your data.

The easiest option: An element's text attribute

Most Nitrogen elements have both a `body` and a `text` attribute. The `body` attribute will display data as presented, but the `text` attribute will apply HTML encoding before rendering. Here's a simple example:

```
MaliciousText = "<script>alert('HAHA! Gotcha')</script>",
#panel{body=MaliciousText}.
```

In the code snippet above, the injected javascript would be sent to the client and executed in the browser because `MaliciousText` is not properly encoded. The simplest way to ensure `MaliciousText` is not executed in the browser is to use the `text` attribute:

```
MaliciousText = "<script>alert('HAHA! Gotcha')</script>",
#panel{text=MaliciousText}.
```

Simple, eh? I thought so.

Another easy option: HTML encoding.

The easiest option is to simply employ Nitrogen's built-in HTML encoding function: `wf:html_encode/1`. Calling `wf:html_encode("<i>my awesome words</i>")`, will return `"&lt;i&gt;my awesome words&lt;/&gt;"`, completely eliminating the possibility of injecting HTML.

Other options

There are other alternatives for HTML encoding to prevent injection:

- **HTML sanitizing**: Strip out potentially damaging tags or attributes. This will allow your users to enter HTML into their form fields to format their text, without the threat of injecting maliciousness. The Erlang CMS *Zotonic* offers an HTML sanitizer you can use the `z_html:sanitize` function from https://github.com/zotonic/z_stdlib

- **Markdown**: Markdown is the new popular formatting kid on the block. Markdown is a simple and ubiquitous, being used heavily from Github to presentation slidedecks (like Reveal.js) to even authoring books. You can find a solid Erlang Markdown encoder currently maintained by *Erlware*:

  `https://github.com/erlware/erlmarkdown`

- **BBCode**: BBCode is a formatting markup language popularized by the phpBB forum software. It looks like HTML, and when run through a BBCode parser, gets converted to HTML while all actual HTML gets encoded. BBCode encoding is very simple, without a standard, and mostly okay. You can find a simple BBCode parser and converter here (which converts between BBCode, Textile, RTF, and HTML):

  `https://github.com/evanmiller/jerome`

But *even with the Markdown or BBCode solutions*, you still need to make sure that malicious attributes are stripped, as such, it's usually a good idea to do some HTML sanitizing after converting Markdown/BBCode to HTML. Otherwise, a malicious user could do BBCode that looks like:

```
[url=javascript:alert('I'm hacking your gibson!')]
      See my homepage[/url]
```

which would unfortunately render as

```
<a href="javascript:alert('I'm hacking your gibson')">
      See my homepage</a>
```

...which allows an attacker to (once again) execute arbitrary JavaScript on the client.

Under most circumstances, simply encoding HTML is the appropriate solution. The other methods should only be used when you absolutely need to allow users to arbitrarily format text, such as forum posts or pages of a Content Management System (CMS).

Prevent JavaScript Injection

While in the previous section, we witnessed JavaScript injection by means of simply dumping in some `<script>` tags containing JavaScript, there are other ways to get arbitrary JavaScript to execute on your site. If you happen to have some script on your page using JavaScript strings, you're going to make sure you escape any quotes, so as to prevent the script from ending early. Consider something simple that you might have seen on a homepage circa 1996, complete with <blink> tags, animated dripping blood dividers and background MIDI music. `Something like:`

```
<a href="javascript:alert('Welcome to my site, Jesse')">
      Click to welcome</a>.
```

That little script might be complete innocuous on its own, but obviously you can't hardcode someone's name – no one wants to be called by someone else's name, that's just rude! Instead, you might build something on the server and send it to the client, like this:

```
Name = wf:user(),
wf:f("<a href=\"javascript:alert('Welcome to my site, ~s')\">
      Click me</a>", [Name]).
```

No doubt, this would work like above, except what if your user's name was something like "D'angelo", surely you see what would happen:

```
<a href="javascript:alert('Welcome to my site, D'angelo')">
Click to welcome</a>
```

The apostrophe in D'angelo will undoubtedly cause our script to crash, a bad thing for user experience, but an even *worse* thing for security. Imagine our user was malicious set their username to `"'); do_something_malicious("`. Now you've got a real problem on your hands, as it would render like (bolding added to point out our so-called "username"):

```
<a href="javascript:alert('Welcome to my site, ');
      do_something_malicious();">Click to welcome</a>
```

Suddenly, you're executing whatever you want on the client. Now, in this example, it's rather simple, and there's nothing happening that we couldn't already do on our client with a simple JavaScript console (CTRL+SHIFT+I in Chrome, for example). But if this is rendering on *another* user's browser, then you can now execute arbitrary JavaScript on another client.

The solution to this problem can be handled in a couple of different ways:

- You could be a terrible person and face the wrath of geeks and user-experience folks everywhere by preventing users from adding non-alphanumeric characters to their name. But that's dumb, and you don't *want* to be dumb, do you? I thought not. Let's abandon that idea.

- You could escape your javascript using `wf:js_escape(Name)`. This is a good solution when you have no choice to embed some kind of javascript. It would look something like this:

  ```
  wf:f("<a href=\\"javascript:alert('Welcome to my
       site, ~s')">Click me</a>",[wf:js_escape(Name)]).
  ```

- The best option is to use as many Nitrogen elements again with their `text` attribute.

  ```
  #link{text="Click to welcome", click=[
        #alert{text=wf:f("Welcome to my
        site, ~s", [Name])}
  ]}.
  ```

  This approach is superior because you neither have to call any escape functions (escaping is automatic with the `text` attributes), nor do you have to waste time dealing with escaping anything in the HTML (like `\\"`). Further, your code will be syntactically consistent, since you're likely using Nitrogen elements everywhere already. Why muttle your codebase with hand-rolled HTML?

Prevent SQL Injection

SQL Injection is done much like JavaScript injection, but with even greater potential for damage. The most concise demonstration of this is probably the famous XKCD Cartoon about "Little Bobby Tables": http://xkcd.com/327. Go check that out, I'll wait, I got nothin' better to do right now.

Simply put, it's one thing to execute arbitrary JavaScript on a client's browser. It's another thing altogether to let random strangers execute full SQL queries on your production database. They could be destructive and delete data, or they could be spammers or identity thieves trying to harvest as much information from your database as possible. Either way, it's bad.

The most common means by personal information is leaked in unsecured websites is through SQL injection. And most of the time, it goes unnoticed, since SQL injection used for harvesting rarely breaks the system - it just ends up being another line-item in the log files.

If you employ proper practices from the get-go, you don't need to worry about retrofitting your site with proper SQL escaping. Consider old-style PHP taught from books that recommended doing silly things like:

```
mysql_query("Select * from login where loginid=$loginid").
```

While this method is (thankfully) no longer the recommended method of dealing with SQL in PHP[1], it's still no less of a concern when you are writing your SQL statements in code - regardless of language. As such, you want to ensure that whatever database library you use will properly escape your inputs (and most of them do), but you can certainly still execute full arbitrary SQL statements.

In practice, this is simple to do. For this example, I'll use sigma_sql[2], a simple wrapper I use on top of the Emysql driver[3], and demonstrate the right and wrong way to do it

```
%% WRONG!
db:q("Select firstname, lastname
      from login where loginid="
      ++ Loginid).

%% Right
db:q("Select firstname, lastname
      from login where loginid=?",
      [Loginid]).
```

Most SQL drivers will do variable and escaping with the use of a `?` token like that. Then you don't have to worry. Indeed, had the first query been employed, then if a user were to specify a Loginid in a query as simple as `"loginid"`, the final SQL string would be:

[1]Unfortunately, while this method is no longer recommended, this poor programming practice persists even today. Indeed, less than a year ago, I had a client need their PHP website overhauled, and one of the first things I saw was the rampant use of unescaped SQL. With a cleverly constructed Login name, a malicious user could literally log into this site as any user they wanted.

[2]https://github.com/choptastic/sigma_sql

[3]https://github.com/Eonblast/Emysql

```
db:q("Select firstname, lastname
      from login where loginid=loginid").
```

Sweet, we just got all the users in the whole system, how convenient. But let's take it a step further. What if the Login variable was bound to `"0 union select credit_card_number, expiration from client_credit_cards"`.

Uh oh, we got a serious problem here, buddy. That query will be called as:

```
db:q("Select firstname, lastname
      from login where loginid=0
      union
      select credit_card_number, expiration
      from client_credit_cards").
```

Suddenly, the probably innocuous list of users' first and last names we originally intended on getting are now actually returning the full list of credit card numbers and their expiration dates from our database. So to avoid this problem altogether, make sure you use your library's provided methods for escaping variables:

```
db:q("Select firstname, lastname login
      where loginid=?", [Loginid]).
```

Be careful with file paths

When presenting files directly from the file system, you want to be careful once again to sanitize inputs. This means not allowing users to specify arbitrary paths for certain files. Basically, if you allow a user to request a file with something like `?file=/etc/passwd`, *you're gonna have a bad time.* Basically, it means stripping offensive things from user-specified data, most specifically, by removing slashes or periods. It's not enough to remove only leading slashes, or leading double-dots (..), because a user can get to files by adding a string like "?file=random/../../../etc/passwd" to the URL. For an attacker (like other injection attacks) is largly trial and error, but once a user has figured out that they can navigate the file system by throwing around requests like that, it's already too late.

Hash those passwords properly

Let's talk about how to properly use passwords in Nitrogen and Erlang.

First, you never, *ever*, want to store passwords in plaintext - that is, in the same form as was sent over the wire. If the database gets compromised (as seems to be happening quite frequently these days, even at internet giants like eBay and Linked In), and if your attackers are able to just see the passwords, you will have caused an unknown but not insiginificant amount of discomfort to your users. Users tend to use the same passwords on most services, or slight variations, like appending a "1" or a "!" depending on the different password strength rules each site chooses to enforce.

If passwords must be reversible, which is still a bad idea, you could use some kind of encryption, like AES. But I'm not going to tell you how to do that, because, as I said, *it's still a bad idea*.

What you want to do is use what's called a hashing function.

A hashing function takes an input of some sort, and returns a unique term based on your input. That unique term? That's called a hash. Using a hash theoretically makes it impossible to get the original input from the hash (called *reversing a hash*), but clever cryptographers have found some hashing functions to be weaker than others, and that it is possible to reverse some hashing functions.

Beyond that, some hashing functions have not yet been broken, but due to their nature (fast processing), will probably be broken some day.

Indeed, hashing is one of those oddities in computers where doing something *slowly* is desired.

Enter bcrypt.

Bcrypt is a hashing function *designed* to be slow, and designed to be scalable in slowness for the conceivable future, even with the advancement of the speed of computing hardware. Every bcrypt hash contains a *work factor*, which tells bcrypt how hard it has to work to produce a hash. It also contains a *salt*, which is an additional random term added to the input to produce a unique hash even if the input value is the same.

There is an Erlang library called erlang-bcrypt[4] which does the heavy lifting for us here, but even more useful is a library called Erlpass[5]. A simple example for working with bcrypt using Erlpass is as follows:

```
%% Hash the password
Hash = erlpass:hash("this is my password"),
...
%% Check if the provided password hashes
Password = wf:q(password),
Worked = erlpass:match(Password, Hash).
```

For more examples of working with erlang-bcrypt and Erlpass, check out one of Jesse's blog posts[6].

Verify logged-in status in user-specific postbacks

While Nitrogen does in fact verify that requests are valid postbacks from our server, it does not automatically validate if a user's session has expired or a user has logged out or logged in as a different user. This is something we will need to do on postbacks containing user-specific data, or requiring user-specific permissions.

Here's a simple example:

[4]https://github.com/smarkets/erlang-bcrypt
[5]https://github.com/ferd/erlpass
[6]http://sigma-star.com/blog/post/proper-password-hashing-in-erlang-with

```
main() ->
    case wf:role(admin) of
        true -> #template{file="./templates/admin.html"};
        false -> wf:redirect_to_login("/login")
    end.
body() ->
    Userid = wf:to_integer(wf:q(userid)),
    #button{
        text="Delete User",
        postback={delete, Userid}
    }.

event({delete, UseridToDelete}) ->
    MyUserid = wf:user(),
    my_database:delete_something(MyUserid, UseridToDelete),
    wf:wire(#alert{text="User Deleted"}).
```

This very likely contains a bug. The `main()` function is indeed checking that we're logged in and redirecting to login if we're not. All this is well and good. However, what happens if the user opens leaves the computer idle for an hour and the system automatically logs us out, before finally clicking that "Delete User" button?

I'll tell you what happens: the user still gets deleted, and what gets recorded as the user who deleted that record? `'undefined'`.

This is where you need to make sure you're checking your role status again in user-sensitive or role-sensitive postbacks. Consider the following change that

```
event(E) ->
    case wf:role(admin) of
        true -> admin_event(E);
        false -> wf:redirect_to_login("/login")
    end.
admin_event({delete, UseridToDelete}) ->
    MyUserid = wf:user(),
    my_database:delete_something(MyUserid, UseridToDelete),
    wf:wire(#alert{text="User Deleted"}).
```

Note the changes: We've added an additional check during the `event/1` function for checking our role, and if the role passes, executing `admin_event/1`, passing the arguments.

This fix is relatively simple, and will ensure that any postbacks issued will have the role checked again prior to processing the postbacks. If the user had gone idle, or logged out in another tab, this will check to make sure that the user is still logged in as an administrator before deleting our user.

But even this might not be sufficient. Conceivably, we may want to guarantee that we haven't logged out from one admin and logged in as another. Indeed, this could be done by putting the initial role into the page state (with `wf:state/2`), and comparing it against the return value of `wf:user/0`. Consider the following final version:

```
main() ->
    wf:state(original_user, wf:user()),
    case wf:role(admin) of
        true -> #template{file="./templates/admin.html"};
        false -> wf:redirect_to_login("/login")
    end.
body() ->
    UserToDeleteid = wf:to_integer(wf:q(userid)),
    #button{
        text="Delete User",
        postback={delete, UserToDeleteid}
    }.

event(E) ->
    IsSameUser = wf:state(original_user)==wf:user(),
    IsAdmin = wf:role(admin),
    case IsSameUser andalso IsAdmin of
        true -> admin_event(E);
        false -> wf:redirect_to_login("/login")
    end.
admin_event({delete, UseridToDelete}) ->
    MyUserid = wf:user(),
    my_database:delete_something(MyUserid, UseridToDelete),
    wf:wire(#alert{text="User Deleted"}).
```

This version will finally be safe enough for us. We know that if the user logs in as a different user, or logs out, that our system will ensure that we still have the necessary role and also that the user is the same, rejecting the postback otherwise.

Part IV.

Advanced Nitrogen Concepts

18. Custom Elements

Part of the power of Nitrogen is how easily you can take advantage of Nitrogen's "Erlang-Record-As-Element" abstraction. Following the DRY[1] principle, if you're doing a common pattern a little too frequently, it's time to abstract it. While you could certainly just do function calls that return HTML or other Nitrogen elements, to keep your programs more consistent, sometimes it's time to create your own elements.

For learning purposes, we're going to start from scratch, rather than using the built-in commands for generating custom elements.

A Simple Element: Bolding with `#b{}`

Let's start with something ultra-simple. A "Hello World" of custom elements, if you will: We'll create a `#b{}` element to correspond to the HTML `<b>` tag, which bolds the content. For this demo, we'll only use a text attribute which automatically encodes the content.

In your application, let's edit `site/include/records.hrl` and add the following line

```
-record(b, {?ELEMENT_BASE(element_b), text="", html_encode=true}).
```

Okay, what does this mean?

- **`-record(b`** - The name of our element. This is a pretty standard looking record definition so far.

[1]DRY = "Don't Repeat Yourself"

- **`{?ELEMENT_BASE(element_b)`** - This invokes an `?ELEMENT_BASE` macro which creates the base element fields (the base element contains such things as the the `id`, `class`, `style`, and `title` attributes, among others. The macro takes, as its only argument the name of the module that will actually render our element. In our case, we're going to call our module `element_b`.

- **`text="", html_encode=true`** - This establishes which custom attributes we'd like to use when creating our element. We want the user to be able to specify a `text` value, and also specify how it will encode `text`, with the default values for `text` being the empty string, and the default value for `html_encode` being `true`.

So that's the first step.

Now we need to create our module. By convention, custom elements typically go into `site/src/elements`. So let's create a new file in our editor called `site/src/elements/element_b.erl` (remember how we called it `element_b` in the `?ELEMENT_BASE` macro?)

Custom elements expect two functions to be exported:

- `reflect/0` - returns a list of the fields used in the element and is mostly used internally by Nitrogen.

- `render_element/1` - this converts our record into something usable.

So let's make our module.

```
-module(element_b).
-include_lib("nitrogen_core/include/wf.hrl").
-include("records.hrl").
-export([reflect/0, render_element/1]).
```

Alright, here's the beginning of the file. Note that just like in Nitrogen pages, we must include `nitrogen_core/include/wf.hrl`, and we must also include `records.hrl`, which actually defines our element record. This gives us the ability to invoke other Nitrogen elements in our `render_element`/1. **In fact, you can define Nitrogen elements in terms of other Nitrogen elements.** But it can't be "turtles all the way down" - eventually, you have to return HTML.

With the header out of the way, let's define `reflect/0`.

```
    reflect() -> record_info(fields, b).
```

That was easy! This little function is needed because, if you remember, Erlang records get converted to plain tuples at compile time, meaning we otherwise do not have access to our element's fields at runtime. The `record_info(fields, RecordName)` function gives us access to those fields.

Finally, let's get to the meat of our new element, the `render_element/1` function:

```
    render_element(Rec = #b{}) ->
        Text = Rec#b.text,
        Encode = Rec#b.html_encode,
        Body = wf:html_encode(Text, Encode),
        wf_tags:emit_tag(b, Body, [
            {class, Rec#b.class},
            {title, Rec#b.title},
            {style, Rec#b.style},
            {data_fields, Rec#b.data_fields}
        ]).
```

Okay, so we did a fair amount here. Let's go over it.

Notice in the `render_element` definition we did `Rec = #b{}`. This is a simple Erlang way to ensure that our `render_element` function is only called with a `#b{}` record.

Then we capture the desired encoding and our element's `text` and run it through `wf:html_encode`.

Finally, in the biggest chunk, we invoke a new function, `wf_tags:emit_tag/3`. This function produces HTML tags from the provided information. The arguments for `wf_tags:emit_tag` are as follows:

```
wf_tags:emit_tag(TagName, Content, Attributes)
```

- **`TagName`** in our case was `b`, as we're making a `<b>` tag. Had we put `blink`, it would produce the long-deprecated `<blink>` tag (and also simultaneously transport us back to 1995).

- **`Content`** is what goes between the opening and closing tags. In our case, it ends up being the HTML-encoded version of our `text` field.

- **`Attributes`** is a proplist, which gets converted into the `attribute="value"` pairs found inside an HTML tag.

We can experiment with this in our shell:

```
> wf_tags:emit_tag(blink, "Welcome to 1995", [{class, my_blink}]).
["<","blink",
  [[" ","class","=\"","my_blink","\""]],
  ">","Welcome to 1995","</","blink",">"]
```

Well, that sure doesn't look clear at all! For optimization purposes, Nitrogen's `emit_tag` function actually returns what's referred to in Erlang as an IOList. IOLists are a part of Erlang that lends itself to extremely fast string building, but that goes beyond this chapter[2].

To get a more readable version, just wrap that previous call with `iolist_to_binary/1`.

```
> iolist_to_binary(v(-1)).
<<"<blink class=\"my_blink\">Welcome to 1995</blink>">>
```

Well, that certainly looks more like what we are expecting.

[2]See page 211 for more about strings and IO Lists.

WHOA! WAIT A SECOND? What is that `v(-1)` thing there?
That's a shell function to retrieve the last return value from the shell. It's a convenience function only available in the Erlang shell, meaning it cannot be used inside a module. Basically, whatever number you put in there, it will go back that many return values.[a] So `v(-1)` retrieves the last return value, `v(-2)` retrieves the 2nd last return value, and so on.
My apologies for being a Sneaky Pete!

[a]See http://www.erlang.org/doc/man/shell.html for more sneaky Erlang shell-only commands.

Using your new `#b{}` element

Now all we need to do is compile our module and use it in our application.

For the sake of demonstration, let's create a sample page called "sweet-tag" to sample our sweet new tag.

```
$ vim site/src/sweettag.erl
-module(sweettag).
-include_lib("nitrogen_core/include/wf.hrl").
-include("records.hrl"). %% bring in our custom element
-compile(export_all).

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

body() ->
    #panel{body=[
        #h2{text="Are you ready for the awesome?!"},
        #h1{text="I said: ARE YOU READY FOR THE AWESOME?!"},
        #br{},
        #br{},
        #b{text="BOOM BABY, THIS IS THE AWESOME!"}
    ]}.
```

And there it is! Our custom `#b{}` element from start to finish!

Function Readability, Pattern Matching, and Convention
For the sake of ease of readability for newcomers, we referenced the fields of the record as `Record#b.fieldname`. While this certainly works, it's actually far more common to use pattern matching to eliminate those extra lines.
For example, one would probably be more likely to redefine the `render_element/1` function above as:

```
render_element(Rec = #b{text=Text, html_encode=Enc}) ->
    Body = wf:html_encode(Text, Enc),
    wf_tags:emit_tag(b, Body, [
        {class, Rec#b.class},
        {title, Rec#b.title},
        {style, Rec#b.style},
        {data_fields, Rec#b.data_fields}
    ]).
```

`Text` and `Enc` get bound right in the function definition, cutting down on the lines of code.
This is a pretty common convention, more than a rule, but it shortens your code and tends to increase the readability of your program.

Advanced Custom Element: Time Selector with `#time_selector{}`

Alrighty, we've already cut our teeth on something simple, let's go a little more advanced. I'm going to move a little faster here, rather than spelling everything out.

Let's make a very simple #time_selector{} element: we can specify a start time, and end time, an increment (in minutes), and a current selected value. The value submitted will be the number of minutes from midnight.

Let's get started.

First, we create the record in site/include/records.hrl.

```
$ vim site/include/records.hrl
...
-record(time_selector, {?ELEMENT_BASE(element_time_selector),
        start=0, finish=1440, increment=60, value}).
```

Now we create and start editing site/src/elements/element_time_selector.erl

```
$ vim site/src/element/element_time_selector.erl
-module(element_time_selector).
-include_lib("nitrogen_core/include/wf.hrl").
-include("records.hrl").
-export([reflect/0, render_element/1]).

reflect() -> record_info(fields, time_selector).
```

So far, so good. We've done everything with the boilerplate we need to do.
Now to the meat:

```
render_element(Rec = #time_selector{start=Start,
        finish=Finish, increment=Inc, value=Value}) ->
    Times = lists:seq(Start, Finish, Inc),
    Options = [format_time(T) || T <- Times],
    #dropdown{
        options=Options,
        value=Value,
        class=Rec#time_selector.class,
        style=Rec#time_selector.style,
        title=Rec#time_selector.title,
        data_fields=Rec#time_selector.data_fields
    }.
```

Okay, what did we do here? First, we bound `Start`, `Finish`, `Inc`, and `Value` in our function definition. Then we used `lists:seq/3` to create a list of times from our `Start`, `Finish`, and `Inc`. Then we iterated through the Times with a as-of-yet undefined function `format_time/1` function to give us what we want to be a final list of `#option{}` elements. Finally, we return a `#dropdown{}` element.

Notice something interesting here?
Yeah, we return a Nitrogen element instead of HTML! This is part of the beauty of Nitrogen. If you think about the beauty of Lisp being that everything is an s-expression, the beauty of Nitrogen is that everything can be done in terms of Nitrogen Elements. The Nitrogen rendering engine will recognize that a Nitrogen element has been returned, and in turn, will call the appropriate `render_element` function for the returned element (or list of elements).

Let's finish off our element by creating the `format_time/1` function referenced in `render_element/1`. Since our problem is simple enough, we can write it directly. But if we were doing something more complicated with the time (like having to account for timezones or multiple formats), you might use a date and timezone library like qdate[3].

```
format_time(Time) ->
    Hour = Time div 60,
    Minute = Time rem 60,
    TimeStr = [wf:to_list(Hour), ":", wf:to_list(Minute)],
    #option{text=TimeStr, value=wf:to_list(Time)}.
```

Here we do a pretty naive time formatting. 24-hour formatted time (“13:45” instead of “1:45pm”). A simple integer division with `div` and a simple integer modulo (remainder) with `rem`.

This will finish off our element implementation.

Using your `#time_selector` element

Let’s invoke our brand-spankin’-new `#time_selector{}` element. We’ll just use our previous “sweettag.erl” module (unless you feel some emotional attachment to it, in which case, feel free to make a new page module for this).

We can leave the header the way we started with, and let’s redefine the `body()` function.

[3]https://github.com/choptastic/qdate

```
-module(sweettag).
-include_lib("nitrogen_core/include/wf.hrl").
-include("records.hrl").
-compile(export_all).

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

body() ->
    [
        #h1{text="Pick a time, any time (during the business 
        #time_selector{start=480, finish=1020, increment=30}
    ].
```

Pretty simple! This will render a dropdown that shows the times 8:00 through 17:00 (5:00pm) in half-hour increments.

But we probably want to do something with the dropdown, like initiate a postback and see the new value.

Let's do that that. Let's add a button and an `event/1` function.

```
body() ->
    [
        #h1{text="Pick a time, any time (during the business day)"},
        #time_selector{id=time, start=480, finish=1020, increment=30
        #button{text="I’ve made my decision", postback=select_time}
    ].

event(select_time) ->
    Time = wf:to_integer(wf:q(time)),
    FormattedTime = element_time_selector:format_time(Time),
    wf:wire(#alert{text=["You picked: ", FormattedTime]}).
```

Now, clicking the button “I’ve made my decision” will pop up a JavaScript alert with the text “You picked: 9:30” (or whatever time you selected). Note that we’re just using the `element_time_selector:format_time/1` function to present a readable time to the user. The observant reader would have noticed that we didn’t export `format_time/1` from `element_time_selector`, so let’s go back and do that. Just add `format_time/1` to the list of exports in `site/src/elements/element_time_selector.erl`:

```
-export([reflect/0, render_element/1, format_time/1]).
```

Now, our function will work! Feel free to compile our custom element, and try it out by loading **http://localhost/sweettag** in your browser.

Be mindful of the infinite loop, Luke!
When creating custom elements, you want to make sure you don't accidentally create an infinite rendering loop by having two or more elements cyclically rendering each other. Like if you create a `#x{}` and a `#y{}`, where `element_x:render_element/1` returns a `#y{}` element, and `element_y:render_element/1` returns an `#x{}` element.
If that happens, the Nitrogen rendering engine will go into an infinite loop rendering `#x{}` then rendering `#y{}`, then rendering `#x{}`, and so on.

More Advanced Elements: Adding Custom Postbacks

You may have noticed that some Nitrogen elements generate their own postbacks in the page, rather than using the usual `event/1` function (like the `#inplace_textbox{}` generates a call to `inplace_textbox _event/2`).

Adding this kind of functionality is actually rather simple. In short, you want to generate a postback and use the `delegate` attribute to redirect the postback to our element's render module. Then handle the postback, and make a call back to our page module.

So let's say we want to have our `#time_selector{}` element automatically generate a postback called `time_selector_event(Tag, FormattedTime)`, where `FormattedTime` is a string representation of the time (`"8:00"`), rather than the integer (`480`). So first, let's modify our record definition to include the a `tag` attribute which will be used to identify our element on the page (to disambiguate it from potentially other `#time_selector{}` elements we might have on the page).

```
$ vim site/include/records.hrl
-record(time_selector, {?ELEMENT_BASE(element_time_selector),
        start=0, finish=1440, increment=60, value, tag}).
```

Now we edit the the time_selector element code:

```
$ vim site/src/elements/element_time_selector.erl
-module(element_time_selector).
-include_lib("nitrogen_core/include/wf.hrl").
-include("records.hrl").
-export([reflect/0, render_element/1, format_time/1, event/1]).

reflect() -> record_info(fields, time_selector).
render_element(Rec = #time_selector{start=Start,
        finish=Finish, increment=Inc, value=Value, tag=Tag}) ->
    Times = lists:seq(Start, Finish, Inc),
    Options = [format_time(T) || T <- Times],
    ID = wf:temp_id(),
    #dropdown{
        options=Options,
        value=Value,
        id=ID,
        postback={Tag, ID},
        delegate=?MODULE,
        class=Rec#time_selector.class,
        style=Rec#time_selector.style,
        data_fields=Rec#time_selector.data_fields
    }.

event({Tag, ID}) ->
    Time = wf:to_integer(wf:q(ID)),
    FormattedTime = format_time(Time),
    Page = wf:page_module(),
    Page:time_selector_event(Tag, FormattedTime).
```

Okay, we did kind of a lot here:

1. First, we ensured that we're binding the `Tag` attribute.

2. We generated a temporary ID, which is necessary for these kind of "internal postbacks."

3. Then we assign the `ID, postback`, and change the postback `delegate` to the `element_time_selector` module (`?MODULE`) to the generated `#dropdown{}` element. Note that the postback contains both the `Tag` and the temporary `ID`.

4. We create an `event/1` handler function, which:

   a) retrieves the `Tag` and the `ID.`

   b) Uses that `ID` to get the value of the `#dropdown{}` and converts that value to an integer.

   c) Formats the time into a usable string

   d) Gets the current page module (`wf:page_module`/0)

   e) Calls the `time_selector_event/2` function on our page.

Next, let's make the changes to our page to handle the newly updated `#time_selector element`. Our new function is called `time_selector_event(Tag, FormattedTime)`, so let's add it. Open up `site/src/sweettag.erl` in our editor and modify it to look something like this (as always, changes are in bold):

```
    body() ->
        [
            #h1{text="Pick a time, any time (during the business day)"},
            #time_selector{tag=time, start=480, finish=1020, increment=30},
            ~~#button{text="I've made my decision", postback=select_time}~~
        ].

    ~~event(select_time) ->~~
        ~~Time = wf:to_integer(wf:q(time)),~~
        ~~FormattedTime = element_time_selector:format_time(Time),~~
        ~~wf:wire(#alert{text=["You picked: ", FormattedTime]}).~~
    time_selector_event(Tag, FormattedTime) ->
        Msg = io_lib:format("You picked: ~s with tag ~p", [FormattedTime,
        wf:wire(#alert{text=Msg}).
```

Save your changes and your updated page will be ready to go. As soon as you select a time, it will trigger the `time_selector_event/2` event and produce a JavaScript alert.

What's the deal with `Tag`
-`Tag` is a common convention for tracking postbacks for elements that generate their own custom postbacks. Because those elements, for internal purposes, will likely be using their own internal IDs for tracking the movement of a value throughout postbacks, we need some way to track it so that the user can eventually refer to it. In many cases, there may only be one such field on a page, in which case `Tag` remains mostly unused (matched against `_`). However, in plenty of other cases, you may have more than one of the same type of event-generating element on a page, in which case, the `Tag` is invaluable. *You can think of `Tag` as an alternate field for `postback` when an element uses its own custom event.*

19. Nitrogen Plugins

With the previous chapter on creating custom elements fresh in your mind, let's move quickly into the next logical step for many custom elements: plugins. In most situations, custom elements are largely reusable, and when you're running multiple different applications, you want to avoid copy and paste as much as possible.

The natural progression to avoid code duplication is to package up your custom element into a plugin. Simply put, a plugin is an Erlang application which contains the record definition and code of your custom element.

Reformatting your elements into a distributable plugin is a rather simple task.

Using the previous chapter's `#time` element, we've modifed the following files:

- site/include/records.hrl
- site/src/elements/element_time.erl

The first step is to create your dependency application. For this demonstration, let's name our plugin the same as our time element's module: `element_time`. From your application's directory, type the following

```
$ mkdir lib/element_time_selector
$ cd lib/element_time_selector
$ ../../rebar create-lib libid=element_time_selector
==> element_time (create-lib)
Writing src/element_time_selector.app.src
Writing src/element_time_selector.erl
```

This has gotten us started with a new *library application* (an application that does not need to be "started", but merely contains functions and the like). While the rebar create-lib command creates an element_time_selector.erl file, we won't be using it at all, we just need to overwrite it with our element's code.

So let's move our element code here:

```
$ cd ../..
$ mv site/src/elements/element_time_selector.erl \
  lib/element_time_selector/src
```

And cut the record definition from our relevant include file. So let's create an include directory in our new plugin application directory, then copy our time definition from site/include/records.hrl:

```
$ mkdir lib/element_time_selector/include
$ vim site/include/records.hrl
-include("plugins.hrl").
~~-record(time_selector, {?ELEMENT_BASE(element_time_selector),~~
~~        start=0, finish=1440, increment=60, value, tag}).~~
```

Then paste those contents into the new file `records.hrl`:

```
$ vim lib/element_time_selector/include/records.hrl
-record(time_selector, {?ELEMENT_BASE(element_time_selector),
        start=0, finish=1440, increment=60, value, tag}).
```

We're almost there and ready to go. The last step to creating our plugin is to create a blank file called nitrogen.plugin in the root of our plugin's directory:

```
$ touch lib/element_time_selector/nitrogen.plugin
```

At this point, let's see the basic structure of our plugin:

```
$ tree
    include
        records.hrl
    nitrogen.plugin
    src
        element_time_selector.erl
        element_time_selector.app.src
```

*Almost done, we want to push our new plugin to GitHub[1] (or whatever VCS you prefer).

```
$ cd lib/element_time_selector
$ git init
Initialized empty Git repository in\
           /home/user/myapp/lib/element_time_selector/.git/
$ git add .
$ git commit -m "Initial Commit"
[master (root-commit) 585a960] Initial Commit
 4 files changed, 159 insertions(+)
 create mode 100644 include/records.hrl
 create mode 100644 nitrogen.plugin
 create mode 100644 src/element_time_selector.erl
 create mode 100644 src/element_time_selector.app.src
$ git remote add origin \
      git@github.com/YourUsername/element_time_selector.git
$ git push origin master
```

[1]More information about working with git can be found on page on page 221

Finally, let's add our new plugin to our app's rebar.config file, and rebuild.

```
$ cd ../..
$ vim rebar.config
{deps, [
   {element_time_selector, ".*",
      {git, "git://github.com/YourUsername/element_time_selector",
        {branch, master}}},
     ...
]}.
```

Finally, rebuild your app and try it out:

```
$ make
```

Then open up **http://localhost:8000/sweettag** in your browser. It should render just as if you weren't using the plugin system.

From here, you can add your plugin to your `rebar.config` as just shown, and then simply refer to your `#element_time_selector{}` element without having to do anything else.

Related links for plugins

- Code for the above element_time_selector plugin:
  `https://github.com/choptastic/element_time_selector`

- Official plugin documentation:
  `http://nitrogenproject.com/doc/plugins.html`

- Other plugins:
  `https://github.com/nitrogen/nitrogen/wiki/Nitrogen-Plugins`

- Code of the plugin handling code, which is also an alternative source of plugin documenation
  `http://j.mp/plugincode`
- Jesse's blog post describing the introduction of the plugin system:
  `http://sigma-star.com/blog/post/why-nitrogen-plugins`

20. Advanced Postbacks

21. Custom Handlers

22. Clustering

23. Troubleshooting

Basic overview of the frame of mind you have to be in to try and troubleshoot. Looking for common errors.

24. The challenges of e-mail

Part V.

DevOps

25. Deployment

So here's Bossman running through a cobbled-up slideshow at the weekly *Erlingo!* bag-lunch seminar:

As me dear departed Granny Fran put it, "Many a slip 'twixt the cup and the lip."

It's one thing, in other words, to develop a crafty web application and another thing entirely to deliver it reliably to the user.

History is rife with software deployment disasters. Need we say Affordable Care Act aka Obamacare to make the point?

Here are a few best practices offered in hindsight by an organization with a less than stellar record— our very own U.S. government:[1]

- Understand what people need
- Address the whole experience, from start to finish
- Make it simple and intuitive
- Use data to drive decisions
- Manage security and privacy through reuable processes
- Automate testing and deployments
- Deploy in a flexible hosting environment

[1]https://playbook.cio.gov/

The first four points have everything to do with design and implementation of our Nitrogen applications. But they go far beyond what we can cover here.

But here's the cool thing about Nitrogen— it enables us to bring up interactive prototypes "fast off the block."

An interative cycle of feedback and revise can go a long way toward knocking down deployment gotchas.

We do our best here at *Erlingo!* to track with our clients. But I've been wondering if we can't do better without investing in a fancy testing lab or professional focus groups.

With this in mind I ran across a promising tool— ngrok.[2]

ngrok

ngrok is a dead-easy-to-install-and-run jewel of open source software that enables us to create "introspected tunnels to localhost."

Usecase: "I want to securely expose a local web server to the internet and capture all traffic for detailed inspection and replay."

How does it work?

-
  ```
  Download it.

  $ unzip /path/to/ngrok.zip
  $ ./ngrok -help
  ```

Say you have a web prototype running on port 8000, enter:

```
./ngrok 8000
```

[2]https://ngrok.com/download

And you'll see something like:

```
ngrok  (Ctrl+C to quit)

Tunnel Status online
Version  1.7/1.6
Forwarding  http://<assigned hexidecimal>.ngrok.com ->
                    127.0.0.8000
Forwarding  https://<assigned hexidecimal>.ngrok.com ->
                     127.0.0.8000
Web Interface  127.0.0.1:4040
# Conn  0
Avg Conn Time  0.00ms
```

Now anyone on the web can can "grok" your prototype simply by entering http://<assigned hexidecimal>.ngrok.com in their browser .

Great for getting feedback from clients early and often to assure that we stay on track with their vision and expectations. Our off-site consultants can use ngrok to advantage as well.

But even our most loyal and satisfied client won't be happy for long if the applications we deliver break for whatever reason— security holes, deployment glitch, failure to scale. Countless things can go wrong.

Enter IaaS

Over the past few years our clients have been moving ever more applications out to the "cloud." A few clients have set out to build "private clouds" on their own hardware.

"Cloud" is one of those fuzzy buzz words. But the promise of "cloud computing" is dynamic and reliable deployment of scalable processing, memory, network and, storage resources, indeed, entire clusters, in minutes rather than days, weeks, and months.

Consultants and bloggers call this new tech-trend *IaaS*— Infrastructure as a Service.[3]

The magic behind IaaS is a fast-evolving complement of software tools and technologies including virtual memory providers, configuration and provisioning tools, software containers, stripped-down operating systems, and software repositories.

With these tools we can:

- Provision workstations and servers faster and more securely
- Assure consistency between development, testing, and production environments
- Sandbox risky software
- Harvest more bang for a buck out of our hardware resources
- Respond to demand spikes more effectively
- Quickly bring up and tear down complex stacks and clusters for training and self-learning
- Indeed, meet head-on the challenges of the last three items on the U.S. Government's best-practices list.

A recent on-line presentation by Basho's Bryan Hunt called *Ansible, Vagrant and Riak*,[4] brought all this together for me.

Bryan's presentation demonstrates how to use two important IasS tools to provision and deploy a Riak CS cluster.

To the point, we need to bone up on these technologies for two reasons:

[3] http://www.interoute.com/what-iaas

[4] https://www.youtube.com/watch?v=WVaZxaJ9NOA; http://www.slideshare.net/dataloop/ansible-vagrant-and-riak-bryan-hunt-at-doxlon

- To up our own game
- To help our clients up theirs

Bottom line: Developers need to consider more seriously how their code will be deployed. Operations folks need to better understand code.

The goal: more secure, consistent, and reliable deployment of our applications.

I propose that we devote several bag-lunch seminars toward bringing us up to speed with these issues and technologies.

26. Choose a host

home server

remote server

cloud

http://cstar.io/2014/07/02/nitrogen-on-heroku.html

cluster

Resources

No Single Points of Failure
http://techblog.mdsol.com/2014/06/16/no_single_points_failure.html

27. Stack the deck

Here's Lloyd London, knees knocking, setting out to make his first presentation to the bag-lunch group:

As we all know, you can build Erlang from source on nearly any Unix/Linux system, including OS X.

Get the skinny here:

```
http://www.erlang.org/doc/installation_guide/
      INSTALL.html#id60976
```

If your system runs Erlang, you can install and run Nitrogen.

More here:

```
https://github.com/nitrogen/nitrogen/blob/master/rel/
      overlay/win/README.md
```

With due respect and apologies to all in the Microsoft Windows world, we live on planet Linux so can't help much with MS ideosyncracies. But if Microsoft is your game, don't let that stop you from running Nitrogen.

Grok the know-how here:

```
http://www.erlang.org/doc/installation_guide/
      INSTALL-WIN32.html
```

If you're truly fearless and would like to experiment with the low-power low-cost ARM boards popping up on the market, here's a place to start:

```
http://nerves-project.org/
```

But why limit yourself to one OS when you can partake of an OS smorgasbord?

A fast-evolving world of software techologies, including virtual machines (VMs), software containers, and a flourishing crop of configuration, provisioning, and VM management tools makes it not only possible, but downright easy.

If you haven't followed the news, VM and rival Linux container technologies enable you to:

- Install nearly any OS of choice in an isolated environment on top of your current development workstation or server OS
- Share cpu, memory, and I/O resources with the base system
- Build, test, and tear-down application stacks at a whim
- Easily recover your application stack in event of disaster
- Easily move your application across physical machines and up to the cloud
- Maintain consistent configuration across development, testing, and production systems

- Build your own private or public cloud in the cloud or on your own hardware

So let's take a low fly-over these technologies, looking first at virtual machines.

VirtualBox

Oracle's open-source VirtualBox is one of many VM providers. We like VirtualBox because it's free and runs on many operating systems. Google "virtual machine software" for other options.

VirtualBox "Presently... runs on Windows, Linux, Macintosh, and Solaris hosts and supports a large number of guest operating systems including but not limited to Windows (NT 4.0, 2000, XP, Server 2003, Vista, Windows 7, Windows 8), DOS/Windows 3.x, Linux (2.4, 2.6 and 3.x), Solaris and OpenSolaris, OS/2, and OpenBSD."

With VirtualBox installed on your machine, you can up, run, and destroy virtual machines hosting nearly any OS and software stack your heart desires.

Dive deeply here:

```
https://www.virtualbox.org/
https://www.virtualbox.org/manual/UserManual.html
```

We happen to run Ubuntu 14.04 as our base system. The recipe found here put us in the VirtualBox game *tout de suite.*

```
http://www.n00bsonubuntu.net/content/install-virtualbox-
      ubuntu-14-04/
```

OK, OK, pardon my French.

So what's the big win with a virtual machine provider like VirtualBox in your toolbox?

- It enables you to run multiple operating systems simultaneously
- It simplifies software installation
- It provides a safe testing environment
- It facilitates disaster recovery
- It allows you to do more with your exisiting computer/network infrastructure

Indeed, a VM provider on your development workstation or server is a gateway to an exciting new world of DevOp opportunities.

Drawback? Setting up a VM is a minor pain in the tush. This is where Vagrant comes in.

Vagrant

Vagrant docs put it this way: "Instead of building a virtual machine from scratch, which would be a slow and tedious process, Vagrant uses a base image to quickly clone a virtual machine."[1]

We looked here to learn how to install Vagrant:

```
    https://www.vagrantup.com/downloads.html
```

Install took less than 10 minutes. Bringing up a new OS was even easier:

[1]https://docs.vagrantup.com/v2/getting-started/boxes.html

```
~$ vagrant version
Installed Version: 1.6.3
Latest Version: 1.6.4
...
~$ vagrant init hasicorp/precise64
A ‘Vagrantfile‘ has been placed in this directory.
You are now ready to ‘vagrant up‘ your first virtual
 environment! Please read the comments in the
Vagrantfile as well as documentation on ‘vagrantup.com‘
for more information on using Vagrant.
~$ vagrant up
...
~$ vagrant ssh
Welcome to Ubuntu 12.04 LTS (GNU/Linux 3.2.0-23-generic
x86_64)
 * Documentation:  https://help.ubuntu.com/
New release ’14.04.1 LTS’ available. Run ’do-
release-upgrade’ to upgrade to it.
Welcome to your Vagrant-built virtual machine.
Last login: Fri Sep 14 06:23:18 2012 from 10.0.2.2
 vagrant@precise64:~$
```

With three simple terminal commands we’ve installed Ubuntu 14.04.1 LTS in a virtual machine— could have as easily been Debian 7.3.0 or OpenBSD 5.4.

You can find a list of Vagrant boxes here:

```
http://www.vagrantbox.es/
```

Steep your mind here to learn the vast extent of Vagrant's wiles:

```
https://www.vagrantup.com/
https://leanpub.com/vagrantcookbook
```

So now, staring at the terminal prompt of our newly minted virtual machine hosting Ubuntu, we're faced with a chore— provisioning. We COULD configure and build a software stack manually or we could cobble up a cunning shell script. But a host of provisioning tools such as Ansible, Chef, and Puppet can automate the process for us.

This is particularly important if we wish to configure more than one workstation or server.

We like Ansible for the task because it's simple to use and has numerous other charms.[2] But don't let that stop you from using Chef or Puppet or whatever other provisioning tool floats your boat.

Ansible

"Ansible is the simplest way to automate IT," claims the Ansible website.

A devoted blogger put it this way:

"Ansible is used from a developer- or continuous-integration-machine, which executes tasks on hosts from an inventory. You only need SSH servers running and private keys to connect to them to get it working. With the inventory of hosts to operate on, you can chose to execute ad-hoc commands using ansible commands or playbooks. Playbooks are files written in YAML.[3]

[2]http://www.ansible.com/home
[3]"http://www.whitewashing.de/2013/11/19/setting_up_development_machines_ansible_edition.ht

YAML, according to the official website, “is a human friendly data serialization standard for all programming languages.”[4]

Here’s an example of an Ansible “inventory,” e.g. list of systems under Ansible direction:

```
mail.example.com

[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
three.example.com
```

Assuming SSH connection, a user can issue direct commands to any or all of the systems in the inventory:

```
$ ansible all -m ping
127.0.0.1 | success >> {
"changed": false,
"ping": "pong"
}
```

But a more reliable method is to create an Ansible “playbook.” Want to install Erlang? Here are two playbooks that will do it for you:

[4]http://www.yaml.org/; http://www.yaml.org/spec/1.2/spec.html

```
Ansibles/erlang
https://github.com/Ansibles/erlang

stwind/ansible-erlang
https://github.com/stwind/ansible-erlang
```

Ansible resources

- Ansible
  - http://www.ansible.com/home
- An Ansible Tutorial
  - https://serversforhackers.com/editions/2014/08/26/getting-started-with-ansible/
- Ansible, Vagrant and Riak- Bryan Hunt at #DOXLON
  - https://www.youtube.com/watch?v=WVaZxaJ9NOA
  - http://www.slideshare.net/dataloop/ansible-vagrant-and-riak-bryan-hunt-at-doxlon

(NOTES)

I would like to develop an Ansible playbook to install Nitrogen, but need time to play.

apt-get update
apt-get install git
sudo apt-get install python-software-properties
sudo apt-get update

sudo apt-get install -y ansible

vagrant@precise64:~$ ansible –version ansible 1.4.4

Scalable and Understandable Provisioning with Ansible and Vagrant

https://julien.ponge.org/blog/scalable-and-understandable-provisioning-with-ansible-and-vagrant/

Ansible

http://docs.ansible.com/intro.html

Evaluating Ansible

http://blog.wains.be/category/automation/

Getting started with ansible

http://lowendbox.com/blog/getting-started-with-ansible/

Using Vagrant to provision/setup multiple node Riak cluster for development environments.

http://suvashthapaliya.com/blog/2012/11/riak-multi-node-cluster-setup-for-development-environment-with-vagrant/

Docker

Virtual machines deliver process isolation, but they carry heavy overhead: e.g. for each VM, they load an OS on top of the base OS. Docker and similar "software containers " provide isolation with considerably less call on hardware resources. Here's how *Linux Journal* puts it:

"Containers now can be used as an alternative to OS-level virtualization to run multiple isolated systems on a single host. Containers within a single operating system are much more efficient, and because of this efficiency, they will underpin the future of the cloud infrastructure industry in place of VM architecture.

"Compared to a virtual machine, the overhead of a container is disruptively low. They start so fast that many configurations can launch on-demand as requests come in, resulting in zero idle memory and CPU overhead. A container running systemd or Upstart to manage its services has less than 5MB of system memory overhead and nearly zero CPU consumption. With copy-on-write for disk, provisioning new containers can happen in seconds."[5]

Docker employs a "Dockerfile" to provision Docker containers. See mingfang/docker-erlang, for instance, to see how Erlang might be installed undder Docker.[6] But Ansible is up to the job as well.[7]

Will software containers replace virtual machines? Not everyone thinks so.

[8]

But the momentum for now favors containers.

Here are several projects that build Erlang in Docker:

```
https://github.com/mingfang/docker-erlang
http://blog.docker.com/2013/09/powering-voxoz-ecosystem-with-docker/
http://castro.io/2014/07/11/using-docker-to-manage-erlang-
environments-for-riak.html
```

CoreOS

OK, so now you're spinning up a storm of Docker containers in the cloud or on your own hardware. Do you really need heavy-duty OS's to do the heavy lifting?

"No," say the gurus.[9]

[5]http://m.linuxjournal.com/content/containers—not-virtual-machines—are-future-cloud

[6]https://github.com/mingfang/docker-erlang/blob/master/Dockerfile

[7]http://www.ansible.com/docker

[8]Containers vs Hypervisors: The Battle Has Just Begun
http://www.linux.com/news/enterprise/cloud-computing/785769-containers-vs-hypervisors-the-battle-has-just-begun/

[9]http://blog.hendrikvolkmer.de/2013/10/11/the-missing-piece-operating-systems-for-web-scale-cloud-apps/

So this is where CoreOS comes in. Here's how CoreOS developers position it: "CoreOS enables warehouse-scale computing on top of a minimal, modern operating system."

CoreOS is explicitly designed to run multiple Docker instances. The win: faster boot, less burden on hardware resources.

Learn more about CoreOS here:

```
https://coreos.com/
```

Panamax

My, my, managing that big fluffy cloud built with Ansible on CoreOS and Docker can quickly get out of hand. Enter Panamx .

Panamax is billed by the developers at CenturyLink as "Docker Management for Humans."[10] The goal: "...deploying complex containerized apps as easy as Drag-and-Drop."

The big picture

So here we are.

If you happen to have half-a-dozen or more hardware boxes sitting idle, or budget for a hefty cloud account, you too can build an elastic Nitrogen/Erlang infrastructure and/or Riak data warehouse that can handle nearly any computing load you can imagine. Think big. The world is your oyster.

Questions?

[10]http://panamax.io/

PXE

OK, OK, chicken and egg. Yes, I agree. Loading an OS on bare metal is fun the first time around, but get's dreary by the nth. Here's where PXE comes in— Preboot eXecution Environment.

"PXE booting is one of the many ways you can boot a computer (or embedded device, or just about anything, possibly a toaster even) entirely from the network without any form of storage on the destination computer (RAM aside). It can be used for most anything. You can run a complete OS off the network using just the RAM, or you can mass-install/update an OS stored on the local hard disk. This HOWTO will get the core of PXE booting setup, and from there you can use one of the above links to setup a specific distro. "[11]

More here:

```
How To : Setup a PXE Boot Server on Debian Part 1
http://sirlagz.net/2011/05/07/how-to-setup-a-pxe-boot-
     server-on-debian-part-1/
How To : Setup a PXE Boot Server on Debian Part 2
http://sirlagz.net/2011/05/09/how-to-setup-a-pxe-boot-
      server-on-debian-part-2/
How To : Setup a PXE Boot Server Part 3 : Installing
      Debian Squeeze
http://sirlagz.net/2011/05/10/how-to-setup-a-pxe-boot-
      server-part-3-installing-debian/
```

[11]http://pxe.dev.aboveaverageurl.com/index.php/PXE_Booting

28. Launch day

Server Security

Configuration

Virtual hosts

Domain names

DNS

29. Maintain maintain

Monitoring

Linux-Dash
http://linuxdash.com/

Backup

Code change

Part VI.

Appendices

A. Erlang from the top down

In our humble opinion, syntax is not Erlang's greatest charm. No, we're not of the "hate Erlang syntax" school. Cast habit and prejudice aside and Erlang syntax offers treasures enough— first-class functions, immutable variables, pattern matching, list comprehensions. Erlang binaries are wicked cool. Truth be told, Erlang syntax is not that difficult to learn.

No, we vote for OTP.

Indeed, OTP is so boffo we won't even let you in on what the acronym stands for— it oh so short-changes the range of application.

OTP is a collection of libraries and underlying philosophy that significantly slaps down the pain that accompanies development of highly scalable, reliable, distributed software systems.

To convey a taste, here's a fly-over.

Erlang Release

The whole point of the development enterprise is to produce and ship a software product that installs and runs with ease and satisfaction on client hardware.

In Erlang parlance, this is called a *release.*

Superficially, a release looks like a *.tar.gz file. Ship it, untar it, start it, and it runs— and runs.

Consider that most Erlang programs coordinate swarms of light-weight *processes*, all actively processing messages according to grand design.

Bug? (Tell us your software doesn't have bugs.)

Simply hot-code update the release. That is, install a patch without taking the program out of service.

Tell us that's not cool! While you're going about surgery, the processes just keep doing their thing without pause.

Stop the whole buzzing crew when you wish with a single command.

Think for a moment and you'll understand why every Erlang *release* ideally has a *version number.* Hint: makes hot-code updates possible.

Erlang *releases* are an artful construction of Erlang *applications* plus the Erlang *VM*, *e.g.* virtual machine. Note that the word *application* is a technical term in Erlang— it means what it means no more nor less.

http://www.erlang.org/doc/design_principles/release_structure.html

Erlang Application

An Erlang *application* is a set of **.beam* files plus a **.app* resource file. You met up with **.app* files back in Chapter 2, recall? App files contain explicitly structured meta data that tells the outside world what it needs to know to play nice.

You can think of an Erlang *application* as a more-or-less stand-alone high-level component of an Erlang *release.* The functionality of an Erlang application is hammered out and lives life as a set of Erlang *modules.*

As you may have surmised in Chapter 2, Erlang applications start life as a collection of **.er*l source files.

By convention, all elements of an application are organized in the following sub-directories:

- src
- ebin
- priv

- include

src Contains the Erlang source code.

ebin Contains the Erlang object code, the beam files. The .app file is also placed here.

priv Used for application specific files. For example, C executables are placed here. The function code:priv_dir/1 should be used to access this directory.

include Used for include files.

Erlang applications can be *active* or *static*. Active applications implement, monitor, and control active processes such as *servers*, *state machines*, or *event handlers*. Active applications need to be started and stopped, thus they include an application callback module that defines start and stop procedures. The callback module is usually named, simply, <application>.erl.

Active applications also need to be monitored or supevised. You'd hate to have your whole system die sudden death if an active process fails. So here's one of the keys behind Erlang's touted reliability: active Erlang applications include monitor and/or supervisory modules that take appropriate action should a process choke on a bug or bad data. Supervisors usually have a name of the form <application>_sup.erl.

Most Erlang systems implement a hierchical *tree of supervisors*, where a master supervisor monitors and controls minion supervisors who, in turn, supervise lessor supervisors and workers. Phew! Now you know the deep secret of Erlang reliability.

Library applications, solely composed of static *functions* that do support and grunt work, do not need to be started or stopped, thus do not need application callback nor supervisory module*s.*

http://www.erlang.org/doc/design_principles/applications.html

http://www.erlang.org/doc/design_principles/sup_princ.html

Modules

Modules are the building blocks of Erlang. Open up any *.erl source file and you will see distinctive similarities: comments, a short set of attributes, and a sequence of functions.

Comments look like this:

```
%%% This is a comment.
%% As is this.
% And this.
```

We'll leave it to you to discover the differences.

Attributes look like this:

```
-Tag(Value).
```

Every Erlang module starts like this...

```
-module(Module).
```

...where Module is the name of the module. The module attribute could follow any number of comments in the file but, otherwise, must come before other attributes and functions.

Dig into Erlang docs to discover other important attributes; in particular, note -export(...), -compile(...), and -behaviour(...).

http://www.erlang.org/doc/reference_manual/modules.html

Functions

Functions execute sequential logic and spawn active processes. They're the worker-bee heavy-haulers of Erlang. They take zero or more *parameters*; the number of parameters designates the function's *arity*.

In code, a function looks like this:

```
    <function_name>(<param1>, <param2>) -> <function body> .
```

There's that period at the end again. Pay attention. It's important. In documentation, the function above would be written <function_name>/2.

Note that *my_function/0* is totally distinct from *my_function/1* which, in turn, is totally distinct from *my_function/2.*

http://www.erlang.org/doc/reference_manual/functions.html

BIFs

Erlang comes complete with a whole set of built-in functions called *BIFs*, saving you considerable effort. Fact is, you couldn't even write most of these in Erlang if you tried.

http://www.erlang.org/doc/man/erlang.html

Data Types

Erlang supports a rich set of datatypes including, as you'd expect, integers and floats. Named constants are called atoms. Maps, tuples, and lists make it possible to process related data items in powerful ways including construction of yet more sophisticated data structures.

Useful aside: whenever you see the word term in Erlang documentation, the author is most likely taking about an item of data.

http://www.erlang.org/doc/reference_manual/data_types.html

String Building and IOLists

IOLists are Erlang's unique way of efficiently building strings. Simply defined, an IOList is a list of any combination of binaries, "word-sized" integers (0-255), and recursively other IOLists. Erlang does some fantastic things in the name of optimization with large binaries. So in other languages, when you append elements to a string, you undoubtedly have to copy string1 and string2 and combine them producing string3. Or you're resizing string1 to accommodate the additional bytes in string2. In Erlang, you should rarely do normal string concatenation (with the `++` operator) – using `++` requires not only copying the data you wish to append, but also requires first traversing the list on the left side of the `++` operator.

Instead, you'll want to use use IOLists, which as far as I know, is something unique to Erlang.

Say you have

```
A = "Hello",
B = "World"
```

And you wish to produce "Hello World", probably to send along the wire, or save to a file, or print to the shell, or whatever.

Rather than doing

```
Msg = A ++ " " ++ B
```

Which seems like the obvious thing to do, instead you would do:

```
Msg = [A," ", B]
```

`Msg` would now have the value `["Hello", " ", "World"]`. But the value comes in when we actually *do something* with the text, where "do something", almost invariably means *send somewhere*.

Why *IOList* you might ask? Well, something that's received is input, and something that's sent is output. Input/Output. I/O, hence IOList: *a list optimized for I/O.*

When that `["Hello", " ", "World"]` list gets sent (to a database, or the screen, or the network, or our browser), Erlang sequentially and recursively reads through the list and transmits the bytes it finds, essentially dropping whatever recursive structure they were in. It'll end up being transmitted as the byte string **`Hello World`**.

Which leads us to the power of IOLists. As mentioned, IOLists can be comprised of any combination of lists containing binaries, or integers from 0-255.

So let's say we wanted to send over a socket connection a copy of the screenplay to Forrest Gump (which we have stored in binary format on our hard drive) followed by "...and that's all I have to say about that", if we were using string concatenation, it'd be really slow, since we'd have to first traverse the whole string-list of the script (some 70,000+ bytes), just to add 42 characters. It'd look something like this:

```
{ok, Bin} = file:read_file("forrest_gump.txt"),
Str = binary_to_list(Bin),
Followup = "...and that's all I have to say about that",
Msg = Str ++ Followup,
gen_tcp:send(Socket, Msg).
```

That method would undoubtedly work (and you could also convert our followup string to binary and make one big binary then send it). But the better method, and one you'd see more common, especially when combining many binaries and strings together is again to just use an IOList. Like so:

```
{ok, Bin} = file:read_file("forrest_gump.txt"),
Followup = "...and that's all I have to say about that",
Msg = [Str, Followup],
gen_tcp:send(Socket, Msg).
```

That code is shorter, cleaner, and doesn't involve converting a binary to a list only to traverse that whole list to add a few characters at the end, and even though we've combined a binary and a string list in the same list structure, **the end result is exactly the same**.

Indeed, you can experiment with this using the BIF `iolist_to_binary/1`.

```
> iolist_to_binary("a simple string").
<<"a simple string">>
> iolist_to_binary([<<"a">>, 32, ["complicated", [<<" ">>, <<"iolis
<<"a complicated iolist">>
```

Notice how that 32 gets converted to a space? That's because 32 is the ASCII code for the space character.

The Erlang shell

We could write a whole booklet on the Erlang shell. It enables you to enter and execute Erlang code interactively and it's a damned useful if not essential tool of Erlang and Nitrogen development. Here we'll show you how to fire it up and let you experiment on your own:

```
my_directory$ erl

Erlang/OTP 17 [erts-6.0] [source-07b8f44] [64-bit] [smp:3:3] [async-

Running Erlang Eshell V6.0 (abort with ^G)
1>
```

That last line, "1>", is your cue to start typing. The first thing we'd suggest that you investigate is how to exit. Hint: there's more than one way and they each have their advantages.

http://www.erlang.org/doc/man/shell.html

http://erlang.org/doc/man/erl.html-

And much much more

http://www.erlang.org/doc/man/ets.html

http://www.erlang.org/doc/man/dets.html

http://www.erlang.org/doc/man/mnesia.html

http://erlang.org/doc/apps/stdlib/

http://www.erlang.org/faq/libraries.html

B. Erlang build tools

There is much controversy in the Erlang world about The Right Package ManagerTM

make

Makefile

Rebar

erlang.mk

relx

fogfish/makefile
https://github.com/fogfish/makefile

C. Erlang resources

Most definitely start here: *http://www.erlang.org/*

Bedside reading:

- The bible: Armstrong, Joe, *Programming Erlang: Software for a Concurrent World 2nd Ed.,* Pragmatic, 2013
- Gentle introduction: St. Laurent, Simon, *Introducing Erlang*, O'Rielly, 2013
- Fun and instructive: Hebert, Fred, *Learn You Some Erlang for Great Good!: A Beginner's Guide*, Pragmatic, 2013
- Rigorous: Cesarini, Francesco and Simon Thompson, *Erlang Programming*, O'Reilly Media, 2009
- OTP Emphasis: Logan, Martin; Eric Merritt, and Richard Carlsson, *Erlang and OTP in Action*, Manning Publications, 2010'
- Advanced ninja-mode: Herbert, Fred, *Stuff Goes Bad: Erlang in Anger,* Heroku eBook (erlang-in-anger.com), 2014

Websites

- From Install to First Working App in 30 minutes: *http://introducingerlang.com/*
- The big big picture: *http://erlangotp.com/*

- Conferences and consulting: *https://www.erlang-solutions.com/*
- As always, Google is your friend: *https://erlangcentral.org/*
- Mailing lists: *http://www.erlang.org/static/doc/mailinglist.html*

D. Just enough git

Git is distributed version control system built by Linux Torvalds to handle the thousands of concurrent Linux kernel hackers. Since then, and largely with the rise of Github.com, it has become the main choice of open source projects throughout the world.

While Git is rather simple, it can get complicated, especially to someone new to version control. We're going to do our best to distill git down to the "need to knows" and "probably will need to knows" for the sake of developing a Nitrogen app and a handful of users. Rather than bullet-pointing and "TL;DR"ing, I'm just going to present a narrative. If you've been using git on a regular basis for a few months, and have had to resolve merge conflicts, then you can very likely skip this Appendix.

So let's get started.

Git itself is just a command-line program. Git the program executes commands on a *repository*, which change the state of the repository. A git repository consists of a tree of *commits*. A commit is a set of changes to any number of files in a repository.

Whoa, holy cow, already hitting you with too much abstract terminology and boring stuff— I can see your eyes glazing over. My apologies.

Let's just do things and learn as we go, shall we? If you wanted to read the git manual, you could just type `man git`. So let's do some practical things, and I'll try to explain as we go.

So let's get started... again. I promise to make every effort to make your eyes not glaze over.

Initializing a Repository, Your First Commit, and Configuring

First things first, let's create a repository.

```
$ mkdir my_project
$ cd my_project
$ git init
Initialized empty Git repository in /home/gumm/
      my_project/.git/
```

So there we go. Every git project starts that way. Congratulations, you have something in common with every git-based ultra-successful project ever. *If they started this way, you can too!*

The next step is to create some files and add them to the repository

```
$ touch a
$ touch b
$ ls
a  b
$ git add a b
$ git status
On branch master
Initial commit
Changes to be committed:
   (use "git rm --cached <file>..." to unstage)
        new file:   a
        new file:   b
```

So, we created some files (`a` and `b`), added them to our git *index*, and then run git status to show us the status of our repository. Git's *index* is a temporary location for changes you plan to commit. In our case, we added files `a` and `b` to the index, but we never commited them. Changes that have been added to the index are said to be *staged*. You'll notice that git even hints at this by letting us know how we can *unstage* our files `(use "git rm --cached <file>..." to unstage)`

So let's commit them, what are we waiting for?

```
$ git commit

*** Please tell me who you are.
Run
  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this
      repository.
fatal: unable to auto-detect email address
      (got 'gumm@my-laptop.(none)')
```

What the heck is this? If this is the first time using git, git doesn't know who you are, and every commit must be attributed to someone. So let's tell git who we are:

```
$ git config --global user.email "gumm@sigma-star.com"
$ git config --global user.name "Jesse Gumm"
```

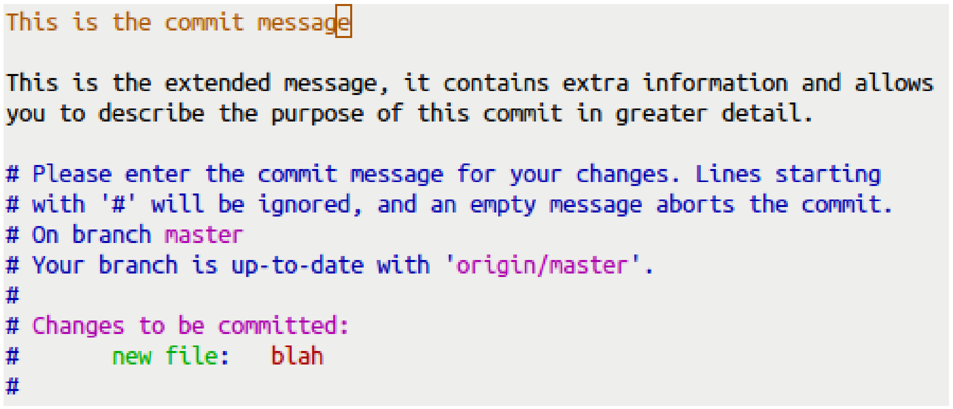
Okay, cool. Git knows who we are now. Let's try that commit again.

```
$ git commit -m "Initial Commit"
[master (root-commit) 9731d0e] Initial Commit
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 a
 create mode 100644 b
```

Sweet! Our first commit.

`git commit` vs `git commit -m`

You'll notice that through this appendix, we'll be sticking with `git commit -m "Some message"`. This is for brevity. More commonly, you'll probably be using just `git commit` without the `-m`. Typing just `git commit` will bring up your text editor and let you type an extended message to attach to the commit. It might look something like this:



```
This is the commit message

This is the extended message, it contains extra information and allows
you to describe the purpose of this commit in greater detail.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#       new file:   blah
#
```

The first line is, of course, our commit, the 2nd line is blank, and the 3rd and 4th line are the extended message. Anything starting with # is a comment and is just there as a summary of what you're about to commit.

Our Change History and Understanding Git Commit Identifiers

Let's have a look at our commit history, shall we? There should only be one commit, but because we're so excited, we want to check it out anyway:

```
$ git log
commit 9731d0edf87d6c6fca5e3aa924ab3bba84dd0805
Author: Jesse Gumm <gumm@sigma-star.com>
Date:   Sat Jun 7 18:19:56 2014 -0500
    Initial Commit
```

Alright! Our commit is in there. Fabulous!

Hey, wait! What the heck is that `9731d0edf87d6c6fca5e3aa924ab3bba84dd0805` string doing there?

That's actually how you identify git commits. If you care, it's a SHA hash of the contents of the commit. You'll also notice that the first 7 characters of that same string are showing when we made the `git commit` command above.

You can refer to commits in commit by that string. For example, to see the details of our last commit, we could do:

```
$ git show 9731d0edf87d6c6fca5e3aa924ab3bba84dd0805
commit 9731d0edf87d6c6fca5e3aa924ab3bba84dd0805
Author: Jesse Gumm <gumm@sigma-star.com> Date:
      Sat Jun 7 18:19:56 2014 -0500
    Initial Commit
diff --git a/a b/a
new file mode 100644
index 0000000..e69de29
diff --git a/b b/b
new file mode 100644
index 0000000..e69de29
```

We see "new file" a twice there, but for the most part, let's not worry about the rest of the contents. I'll show you some better examples later.

It's also worth noting that you do not need to refer to the commit by its whole hash string. You can refer to it by the first X unambiguous characters. Usually the first 5-7 characters are plenty. For example, the following is exactly the same as the previous command:

```
$ git show 9731d0
```

Also, there is a special commit called `HEAD`. `HEAD` is a "moving commit", if you will, in the sense that whenever we add a new commit moves along to the newest commit. For example.

```
$ git show HEAD
```

As a useful little trick, if our repository had more than one commit, you could also refer to commits as relative to HEAD (or any other commit):

```
$ git show HEAD~1  # show commit before HEAD
$ git show 9731d0~1 # show commit before 9731d0
$ git show 973d0^1 # show commit after 9731d0
```

Anyway, enough of that nonsense, let's get back to good stuff.

Committing More Changes

Let's make our repository useful. Let's add some actual data to our files. Let's fire up vim and edit the file called "a" and add the following text:

```
$ vim a
This is my file.
It contain some lines.
Of text.
That last line was not a complete sentence.
```

Then let's see what git thinks of our current *working directory*.

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
modified:   a
Untracked files:
  (use "git add <file>..." to include in what
      will be committed)
a~
```

So git sees that a has changed, but it also sees this file called `a~`. A convention of vim is to save a backup file by appending `~` to the filename. We'll get back to that, but let's commit the changes we care about.

```
$ git add a
$ git commit -m "Add some content to a"
[master ba7c349] Add some content to a
 1 file changed, 4 insertions(+)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will
      be committed)
a~
nothing added to commit but untracked files present
      (use "git add" to track)
```

So far so good. But again, we have this “Untracked file” a~ which is annoying.

Ignoring Things We Don’t Care About

In any given project, there will be numerous files of this nature: files generated by the make system, files left over by our editor, log files, files created by our application itself (like Erlang’s DETS or Mnesia files).

To deal with this, git employs the use of a file called .gitignore. This file lists files and file patterns to simply be ignored by your git project altogether. Let’s create a pretty simple .gitignore file for us here:

```
$ vim .gitignore
*~
```

That should be good enough for now. Now let’s add it to our repo and save.

```
$ git add .gitignore
$ git commit -m "Add .gitignore"
[master f7f5b24] Add .gitignore
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
$ git status
On branch master
nothing to commit, working directory clean
```

Perfect! No more pesky `a~` file messing with our heads. We can also rest assured that any file ending with ~ will be dutifully ignored by git from here on.

This is especially convenient if you decide to add an entire directory to a repository. For example doing the command `git add .` will, without a .gitignore file, add every file git encounters in the whole working directory. This may seem benign, after all, the ~ files are just backup files, who cares if they're in the repo? This however will undoubtedly present issues with dealing with *merging* branches.

I know what you're thinking "Whoa! Whoa! You haven't said *anything* about merging yet! What the heck is that all about?"

Merging Branches

Imagine you're working on a project, and you have your current main stable branch, but you also have a development branch for new features and breaking changes. You don't want to merge those old changes and new changes until the new stuff is stable. This is the fundamental need for branches.

So let's make a new branch:

```
$ git checkout -b dev
Switched to a new branch 'dev'
```

If you come from other version control systems, the term "checkout" might be confusing. In other systems, "checkout" typically means "place a lock on something, preventing others from making changes". In git, it merely means "change the working directory to the structure of that branch." There are no locking of files in git. Specifying the `-b` option with checkout creates a new branch from the current commit.

Awesome, we have a brand new branch. Let's make some changes while we're in here. Let's open up our a file and add a line.

```
$ vim a
This is my file.
It contain some lines.
Of text.
That last line was not a complete sentence.
This is a new line, it fears no man.
```

Excellent, now we commit:

```
$ git add a
$ git commit -m "Add a line that fears no man"
[dev f80773f] Add a line that fears no man
  1 file changed, 1 insertion(+)
```

Fantastic! We've added a sweet new features to our project. As you can see, the commit specifies that we were indeed commiting to the "dev" branch. Let's switch back to the "master" branch but before we do this, let's check the status of our files.

```
$ cat a
This is my file.
It contain some lines.
Of text.
That last line was not a complete sentence.
This is a new line, it fears no man.
$ git checkout master
Switched to branch 'master'
$ cat a
This is my file.
It contain some lines.
Of text.
That last line was not a complete sentence.
```

As you can see, the added line was isolated to the “dev” branch. After switching to “master”, the changes were lost. Now let’s make some “bugfix” type changes.

```
$ vim a
This is my file.
It contains some lines of text.
That last line was indeed a complete sentence.
```

We fixed some grammatical typos in our original text. Let’s commit our changes.

```
$ git add a
$ git commit -m "Fix some typos"
[master 650b90e] Fix some typos
 1 file changed, 2 insertions(+), 3 deletions(-)
```

Great. In the meantime, we've determined that the changes in our "dev" branch are actually stable, so it's time to merge our "dev" branch into "master".

```
$ git merge dev
Auto-merging a
CONFLICT (content): Merge conflict in a
Automatic merge failed; fix conflicts and then commit the resul
```

Uh oh, our first merge, and already we have merging conflicts! <sarcasm>Wonderful!</sarcasm>

Let's see what happened:

```
$ cat a
This is my file.
<<<<<< HEAD
It contains some lines of text.
That last line was indeed a complete sentence.
=======
It contain some lines.
Of text.
That last line was not a complete sentence.
This is a new line, it fears no man.
>>>>>>> dev
```

So git was unable to resolve the differences in the files and auto-merge (the vast majority of merges will be easy enough to be auto-merged). Git so helpfully injected these >>>>>>, ======, and <<<<<< lines to help us know which branches each change is from. We have to remove these lines specifically, and manually fix the merges before we can complete the merge.

Let's resolve this conflict by removing the following lines (marked with strikeout):

```
$ vim a
This is my file.
~~<<<<<<< HEAD~~
It contains some lines of text.
That last line was indeed a complete sentence.
~~=======~~
~~It contain some lines.~~
~~Of text.~~
~~That last line was not a complete sentence.~~
This is a new line, it fears no man.
~~>>>>>>> dev~~
```

The new file will look like this:

```
This is my file.
It contains some lines of text.
That last line was indeed a complete sentence.
This is a new line, it fears no man.
```

Let's commit these changes:

```
$ git add a
$ git commit
[master af8eeda] Merge branch 'dev'
```

Awesome, we've merged in those changes.

You've just done 95% of what you'll ever do with git.

Working With Remote Repositories

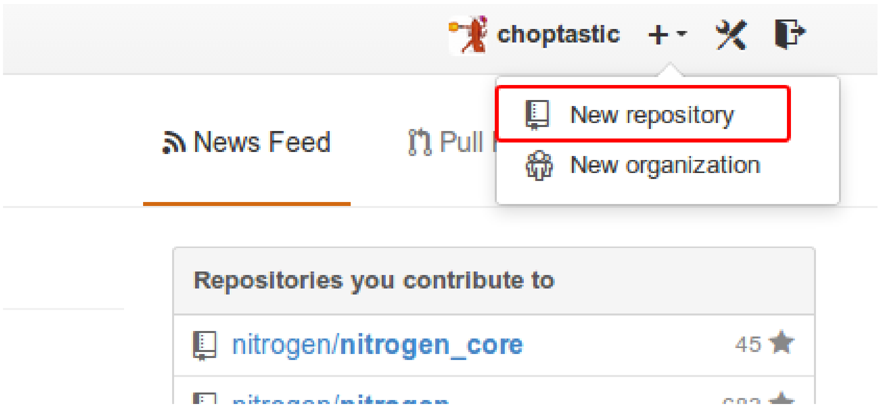
So we say that Git is a *distributed* version control system. What does that even mean?

Unlike non-distributed version control systems, git does not depend on a central repository, though it's actually quite common for projects to have a central repository (for example, Nitrogen's central repository is on Github). But the distributed nature of git means that everyone with a cloned repository has a complete local copy of the repository.

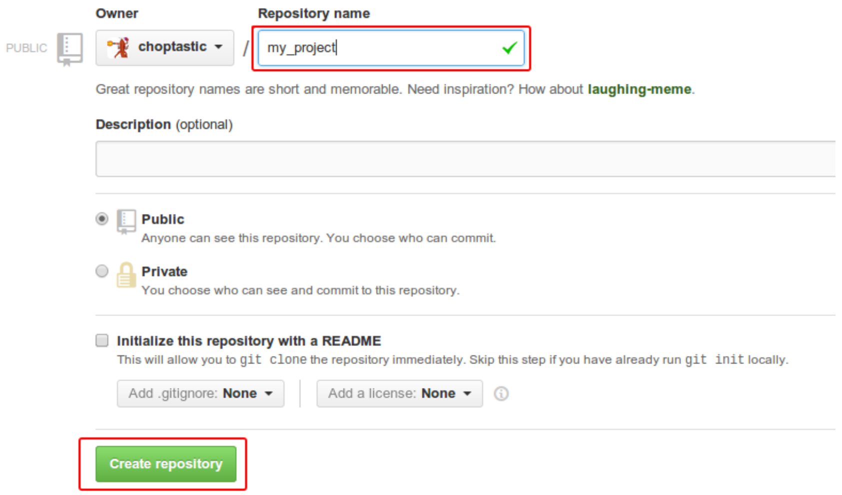
GitHub

For the sake of simplicity, we're going to set up a repository on GitHub.com.

Create an account on github.com and click the little + icon next to your username and click "New Repository"



Give your repo a name and click "Create Repository"


Owner

Repository name

PUBLIC choptastic / my_project ✓

Great repository names are short and memorable. Need inspiration? How about **laughing-meme**.

Description (optional)

[● Public] Anyone can see this repository. You choose who can commit.

Private You choose who can see and commit to this repository.

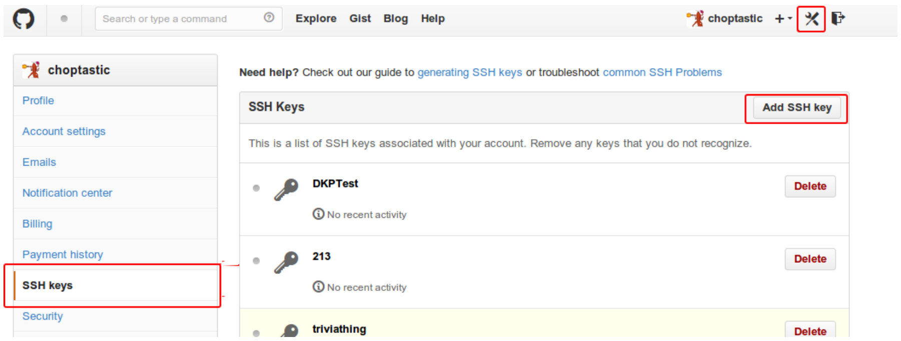
☐ **Initialize this repository with a README**
This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.

Add .gitignore: **None** | Add a license: **None**

Now that our remote repo is created. Before we push everything to Github, we need to make sure we have permissions to push to this repository. We need to tell Github how to validate us, so we need to generate SSH Keys and copy the private key to Github:

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
...
```

Then, on Github, click the little configuration icon on the menu bar, then click “SSH Keys” on the menu on the left, and finally click “Add SSH Key.”



Need help? Check out our guide to generating SSH keys or troubleshoot common SSH Problems

SSH Keys

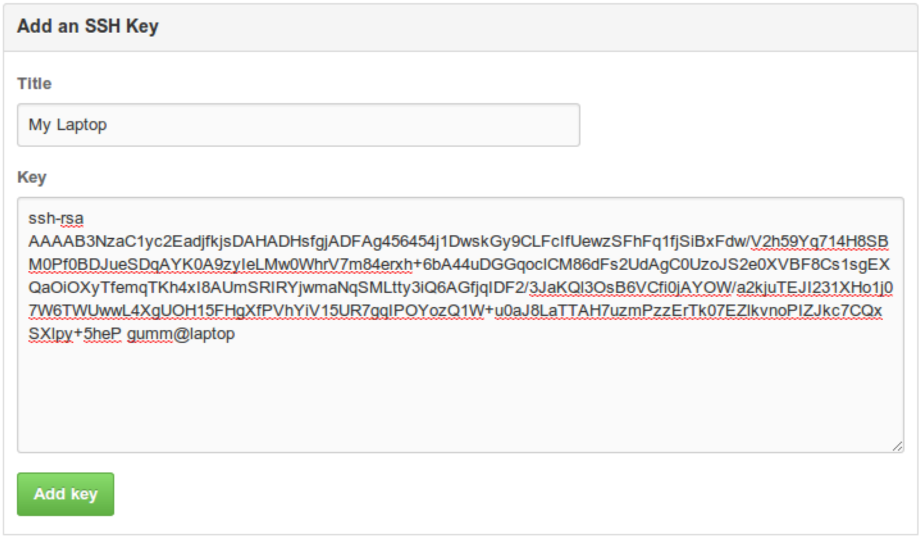
This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

DKPTest
No recent activity

213
No recent activity

triviathing

Give your key a descriptive name (like "My Laptop"), copy the contents of ~/.ssh/id_rsa.pub into the "Key" field and click "Add Key."



Add an SSH Key

Title: My Laptop

Key:

```
ssh-rsa
AAAAB3NzaC1yc2EadjfkjsDAHADHsfgjADFAg456454j1DwskGy9CLFcIfUewzSFhFq1fjSiBxFdw/V2h59Yq714H8SB
M0Pf0BDJueSDqAYK0A9zyleLMw0WhrV7m84erxh+6bA44uDGGqocICM86dFs2UdAgC0UzoJS2e0XVBF8Cs1sgEX
QaOiOXyTfemqTKh4xI8AUmSRIRYjwmaNqSMLtty3iQ6AGfjqIDF2/3JaKQI3OsB6VCfi0jAYOW/a2kjuTEJI231XHo1j0
7W6TWUwwL4XgUOH15FHgXfPVhYiV15UR7gqIPOYozQ1W+u0aJ8LaTTAH7uzmPzzErTk07EZlkvnoPIZJkc7CQx
SXIpy+5heP gumm@laptop
```

Pushing to and Pulling from Our Remote Repository

Great! Our account is all set up to push to our Github repository. Let's do that.

```
$ git remote add origin git@github.com:choptastic/
      my_project.git
$ git push origin master
Counting objects: 18, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (18/18), 1.68 KiB | 0 bytes/s,
      done.
Total 18 (delta 3), reused 0 (delta 0)
To git@github.com:choptastic/my_project.git
 * [new branch] master -> master
```

You might be wondering what "origin" is. "origin" is the default name for a remote repository in git. You can be connected to more than one remote repository, and indeed, that practice is generally recommended. So what we did was to push the "master" branch to the remote repository identified by "origin" (in this case, our Github repo).

And just like push, we can also pull recent changes from the repository with

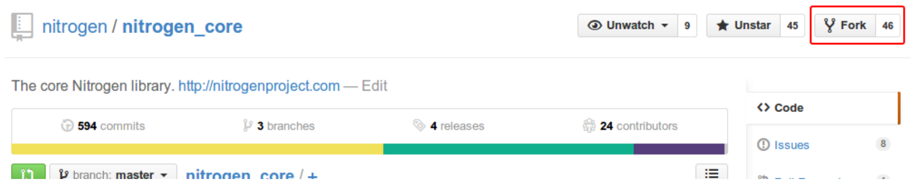
```
$ git pull origin master
From github.com:choptastic/my_project
  * branch           master     -> FETCH_HEAD
Already up-to-date.
```

This can be thought of like downloading and merging from a remote repository. And just like normal merging, if there are merge conflicts, it'll let you know.

Forking and Cloning

Let's say there's an open source project you wish to contribute to (like Nitrogen!). You're going to want to be able to work on it without requiring push access to the "official" repo. This means forking it.

Let's fork Nitrogen and work on it. As you may have noticed, the main Nitrogen repository is actually just a shell that brings together a handful of Erlang applications that becomes a Nitrogen deployment. The meat of Nitrogen is actually found in the nitrogen_core repository. Let's fork that. Head your browser to
https://github.com/nitrogen/nitrogen_core and click the "Fork" button.



This will create a fork of the nitrogen_core repo in your account, which you'll have full push access to.

My github username is "choptastic", so let's now clone my newly forked nitrogen_core repository.

```
$ git clone git@github.com:choptastic/nitrogen_core.git
```

That was easy.

Now let's make some kind of simple change and make our first contribution to improving Nitrogen.

Pull Requests

Let's say we've found a bug in Nitrogen and we want to fix it. In this case, we've found that the #panel{} element is lacking the type signatures for help with Dialyzer. Let's fix that.

But before we do that, we want to employ Good Open Source Practices, and create what's typically called a *topic branch*, which is just a fancy way to say that we're creating a branch that fixes a specific bug, or adds some new feature.

```
$ cd nitrogen_core
$ git checkout -b panel_type_specs
Switched to a new branch 'panel_type_specs'
```

Now that we have our topic branch all set up, let's fire up our editor and edit the source code for the `#panel{}` element. Here's what it currently looks like:

```
$ vim src/elements/layout/element_panel.erl
-module (element_panel).
-include_lib ("wf.hrl").
-compile(export_all).
reflect() -> record_info(fields, panel).
render_element(Record) ->
     Body = [
        wf:html_encode(Record#panel.text,
            Record#panel.html_encode),
        Record#panel.body
    ],
    wf_tags:emit_tag('div', Body, [
        {id, Record#panel.html_id},
        {class, Record#panel.class},
        {title, Record#panel.title},
        {style, Record#panel.style},
        {data_fields, Record#panel.data_fields}
    ]).
```

Let's make the following changes (changes bolded)

```
-module (element_panel).
-include("wf.hrl").
-export([
    reflect/0,
    render_element/1
]).
-spec reflect() -> [atom()].
reflect() -> record_info(fields, panel).
-spec render_element(#panel{}) -> body().
render_element(Record) ->
    Body = [
        wf:html_encode(Record#panel.text,
              Record#panel.html_encode),
        Record#panel.body
    ],
    wf_tags:emit_tag('div', Body, [
        {id, Record#panel.html_id},
        {class, Record#panel.class},
        {title, Record#panel.title},
        {style, Record#panel.style},
        {data_fields, Record#panel.data_fields}
    ]).
```

Our changes here are simple. We've switched from `-include_lib` to `-include` (a remnant from early Nitrogen versions), changed from `export_all` to using explicit exports (when writing API-type things, you want to use explict exports). And finally adding Erlang type specs, which not only help with documentation, by being able to seeal what a function is supposed to accept and return, and also with debugging using the Dialyzer tool (which analyzes every call in the system to find type inconsistencies before they're reach runtime). For the sake of this demonstration, though, **it's not critical to understand the nature of these fixes.** We're just demonstrating pull requests.

Let's save our changes and commit them.

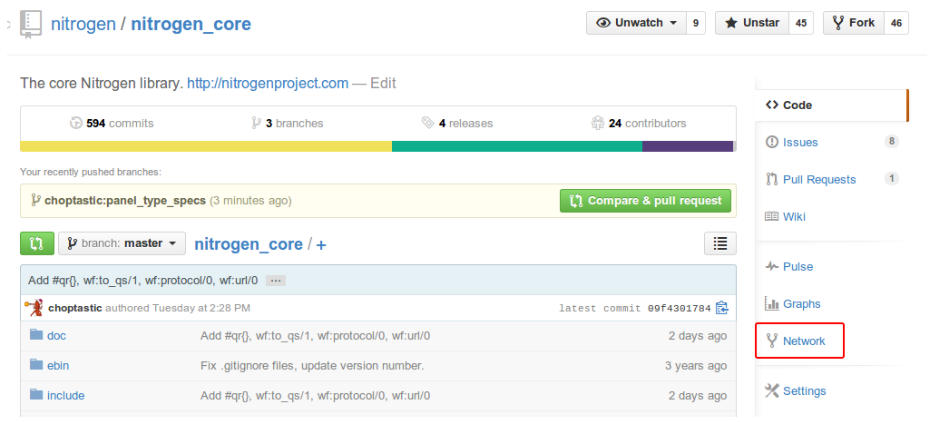
```
$ git commit -m "Add typespecs to #panel{}"
[panel_type_specs 7adc730] Add typespecs to #panel{}
 1 file changed, 9 insertions(+), 3 deletions(-)
```

Here we commit the changes, then push our topic branch off to Github.

```
$ git push origin panel_type_specs
Counting objects: 36, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 660 bytes | 0 bytes/s,
      done.
Total 6 (delta 5), reused 0 (delta 0)
To git@github.com:choptastic/nitrogen_core.git
 * [new branch]      panel_type_specs -> panel_type_specs
```

Fabulous, our topic branch is now there. Let's have a look, shall we?

Head to https://github.com/nitrogen/nitrogen_core and click the "Network" tab on the right.


nitrogen / nitrogen_core

Unwatch 9 | Unstar 45 | Fork 46

The core Nitrogen library. http://nitrogenproject.com — Edit

594 commits | 3 branches | 4 releases | 24 contributors

Your recently pushed branches:

choptastic:panel_type_specs (3 minutes ago) | Compare & pull request

branch: master | nitrogen_core / +

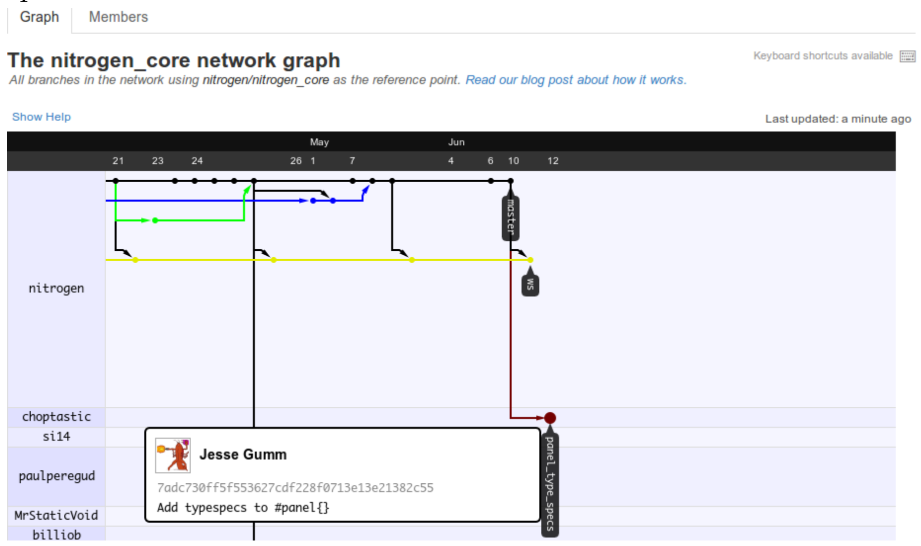
Add #qr{}, wf:to_qs/1, wf:protocol/0, wf:url/0 ...

choptastic authored Tuesday at 2:28 PM | latest commit 09f4301784

doc	Add #qr{}, wf:to_qs/1, wf:protocol/0, wf:url/0	2 days ago
ebin	Fix .gitignore files, update version number.	3 years ago
include	Add #qr{}, wf:to_qs/1, wf:protocol/0, wf:url/0	2 days ago

Clicking that will open up a view of the all commits and branches by all users who've forked our project and pushed their changes up to Github. Quite frankly, the Network view is my absolute favorite part of Github.

Here we'll see our branch, and by hovering over the commit dot on the graph.


Graph | Members

The nitrogen_core network graph

All branches in the network using nitrogen/nitrogen_core as the reference point. Read our blog post about how it works.

Keyboard shortcuts available

Show Help

Last updated: a minute ago

Jesse Gumm

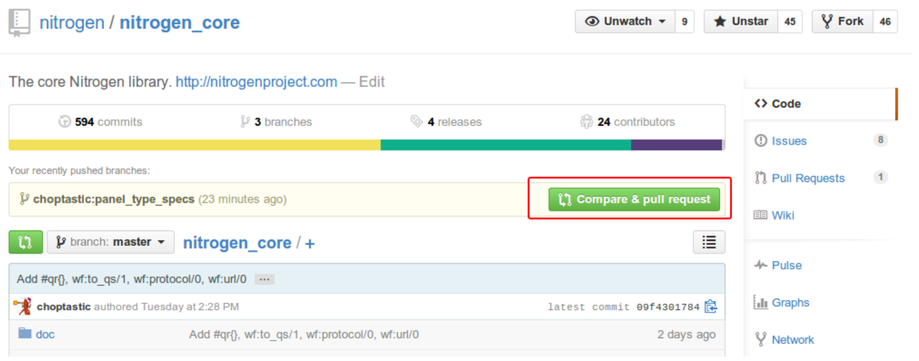
7adc730ff5f553627cdf228f0713e13e21382c55

Add typespecs to #panel{}

Sweet, it's there.

Now, let’s issue the pull request. Head back to the “Code” screen (or just the default page for the repo) - the quickest way is to click the big blue nitrogen_core link at the top of the screen.

Then you’ll see a highlighted row showing our topic branch (choptastic:panel_type_specs). Click the “Compare & pull request” button.

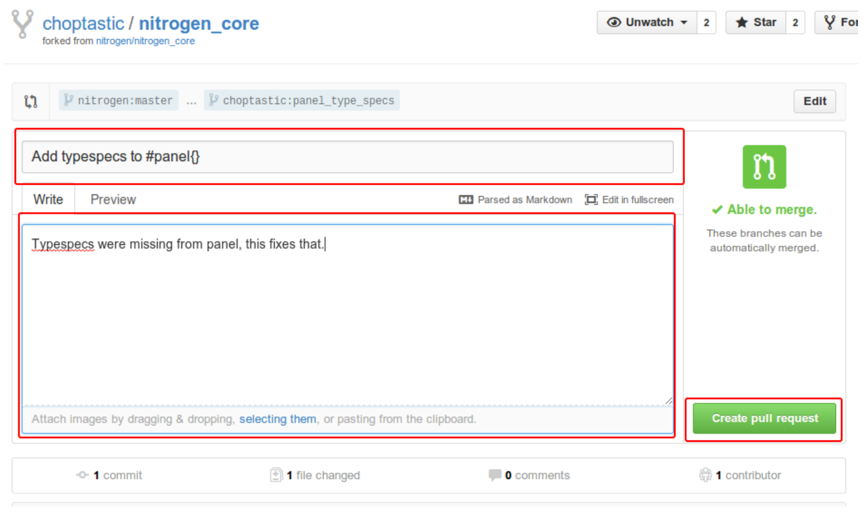


Add #qr{}, wf:to_qs/1, wf:protocol/0, wf:url/0

choptastic authored Tuesday at 2:28 PM latest commit 09f4301784

doc Add #qr{}, wf:to_qs/1, wf:protocol/0, wf:url/0 2 days ago

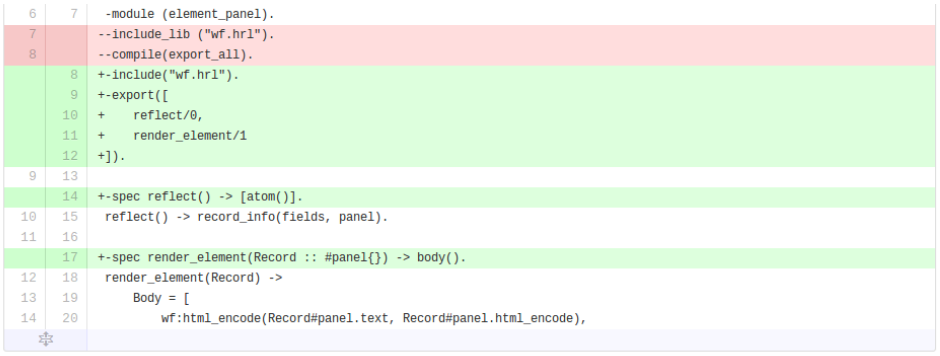
This will bring us to a pull request page to review and initate the pull request. Fill out the pull request as detailed as you feel is necessary. Typically, if this solves an open Github issue issue, or some conversation from the mailing list, it’s good form to provide a link to the relevant conversation, otherwise, explaining in detail your changes. Further, if your change represents a change in an API, it’s a good idea to modify the documentation accordingly. Every project is different in this regard, so make sure you read every project’s CONTRIB file. Most large projects have one.


choptastic / nitrogen_core
forked from nitrogen/nitrogen_core
Unwatch 2
Star 2
Fork
nitrogen:master ... choptastic:panel_type_specs
Edit
Write
Preview
Parsed as Markdown
Edit in fullscreen
Able to merge.
These branches can be automatically merged.
Attach images by dragging & dropping, selecting them, or pasting from the clipboard.
Create pull request
1 commit
1 file changed
0 comments
1 contributor

Add typespecs to #panel{}

Typespecs were missing from panel, this fixes that.

If you want to review your pull request before submitting it, just scroll down the page and you'll see a diff of all the changes. This will look like this:

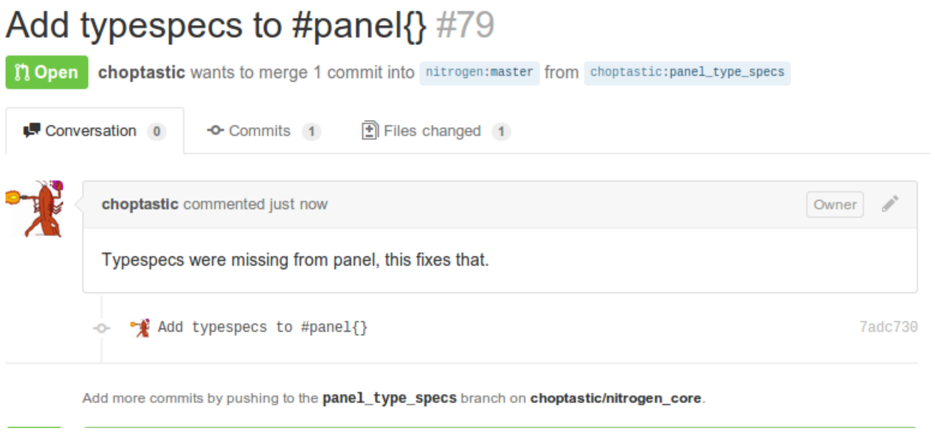


```
 -module (element_panel).
--include_lib ("wf.hrl").
--compile(export_all).
+-include("wf.hrl").
+-export([
+    reflect/0,
+    render_element/1
+]).

+-spec reflect() -> [atom()].
 reflect() -> record_info(fields, panel).

+-spec render_element(Record :: #panel{}) -> body().
 render_element(Record) ->
     Body = [
         wf:html_encode(Record#panel.text, Record#panel.html_encode),
```

When you feel confident that your changes are up to your satisfaction, click the "Create pull request" button.


Add typespecs to #panel{} #79

Open choptastic wants to merge 1 commit into nitrogen:master from choptastic:panel_type_specs

choptastic commented just now

Owner

Typespecs were missing from panel, this fixes that.

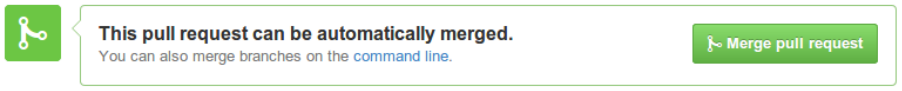
Add typespecs to #panel{} 7adc730

Add more commits by pushing to the **panel_type_specs** branch on **choptastic/nitrogen_core**.

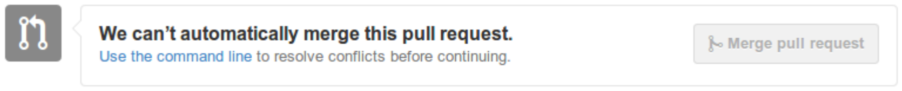
Congratulations, you've just submitted your first pull request! If a change is simple enough, the owner will likely just merge the changes.

As a user, you'll receive an email regarding any action related to a pull request, be it comments, or the owner closing, re-opening, or merging your pull request.

If you are a repository owner (or someone will push permissions to a repository), you'll be given the ability to act on pull requests, either by closing or merging them. If the pull request can be merged by Git without conflict, you'll be presented with a friendly green button:


This pull request can be automatically merged.
You can also merge branches on the command line.

However, if the pull request can not be cleanly merged (usually caused by the master branch changing the same line of code as one of the lines changed in your own pull request), then you would be presented with this message:


We can't automatically merge this pull request.
Use the command line to resolve conflicts before continuing.

Clicking the "use the command line" link in either box will give you instructions on manually completing the merge on the command line.

These commands, with a few additions, will look *strikingly familiar*. Indeed, being a remote branch means nothing.

While clicking the "Merge pull request" box is the simpler route when available, let's use the command line to merge our changes (to better illustrate what we're typing, I've bolded our commands).

```
$ git checkout -b choptastic-panel_type_specs master
Switched to a new branch 'choptastic-panel_type_specs'
$ git pull git://github.com/choptastic/
      nitrogen_core.git panel_type_specs
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From git://github.com/choptastic/nitrogen_core
 * branch            panel_type_specs -> FETCH_HEAD
Updating 09f4301..7adc730
Fast-forward
 src/elements/layout/element_panel.erl | 12 +++++++++---
 1 file changed, 9 insertions(+), 3 deletions(-)
```

Those two commands created a branch off master, then pulled the changes down from the remote repository, merging them into our current branch.

Now we want to merge those changes into master and push those changes off to Github.

```
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
$ git merge choptastic-panel_type_specs
Updating 09f4301..7adc730
Fast-forward
 src/elements/layout/element_panel.erl | 12 +++++++++---
 1 file changed, 9 insertions(+), 3 deletions(-)
$ git push origin master
```

Congratulations, you've merged your first pull request!

That just about covers the basics of working with Git. There are lots of resources out there for dealing with some of Git's warts, but the above describes 99% of how you'll work with Git.

10 Git recommendations that are so easy they'll blow your mind right out of your brain-hole!

Pardon the link-bait title! But because bullet points are so easy to grok, I'll throw out some brief recommendations for you when working in Git and contributing to Git-based Open Source Projects in general:

1. Keep commit titles short. Most projects say to keep the commit title *shorter* than 50 characters.

2. Keep extended commit messages under *80 characters per line.* You can make your extended messages as long as necessary, but some stricter open source authors will reject commits on this basis alone. As always, read the CONTRIB doc for your project.

3. Keep the indentation and code style consistent with the original project. If the project uses spaces instead of tabs, make sure you're doing the same. A project maintainer will be annoyed at having to

fix indentation when it's something that can be easily done by the pull request author.

4. If you accidentally commit something and realize there was a mistake in the commit message, you can modify that commit with `git commit --amend`

5. If you need to pull a single commit from somewhere else without doing a full merge, use `git cherry-pick X` where `X` is the commit hash. If it's from a remote repo, use `git fetch git://url/to/repo.git` first, then `git cherry-pick X`

6. Before merging a pile of potentially broken commits into master, it's usually a good idea to squash those broken commits all into a single working commit. You can do this with `git rebase -i X` where X is the commit to start with. Because rebasing can be relatively complex, and there are several guides for doing so online, feel free to search for "git rebase guide". Search engines are your friend.

7. Make your git life easier with tig[1], an ncurses-based interface for git.

8. Accidentally add a file to the index and don't actually want to commit it? Unstage that beast with `git reset HEAD <filename>.` Or use `tig status`, and press the `u` key (you did install tig after my recommendation, right?)

9. You can always edit your repository's configuration in the root of your project in `.git/config`. Incidentally, this `.git` directory is where the whole repository structure is kept. Feel free to spelunk.

10. If you use vim and would like to take advantage of git in the editor, Fugitive[2] is your friend.

[1]https://github.com/jonas/tig
[2]https://github.com/tpope/vim-fugitive

11. Notice that your current commit doesn't work, but you know a past commit does work, except that you don't know which commit first? Use `git bisect`[3]. This will do a sort of binary search of the commits for you. Letting you test each commit, and then telling git if it worked or not, then picking another commit until you've finally discovered exactly which commit first caused the bug. It's extraorinarily powerful, but beyond the scope of this appendix.

[3]A simple guide for `git bisect:` http://www.metaltoad.com/blog/beginners-guide-git-bisect-process-elimination

References

Index

About the Authors

Lloyd R. Prentice

Lloyd is a novelist and small-press book publisher. His first computer had a an S-100 bus and 8K of RAM. He has designed and developed more than 100 educational and consumer software products for major publishers. Web experience includes a soup-to-nuts application to support marketing and management of world-class technical conferences; six months in the making and ten years under non-stop revision and unrelenting deadlines. "You haven't lived," he says, "until several hundred conference attendees — mandarins of IT — log into your URL at the same time and bring your server down." Jesse has patiently taught him much that he wish he'd known at the time.

Jesse Gumm

Jesse wrote his first program in QBASIC for DOS: a two-player text-based fighter game. He's now an entrepreneur specializing in web application development. His current flagship product, BracketPal, is a sports league management system written in Nitrogen. In open source, he's the project leader of Nitrogen and its core dependency SimpleBridge, and he has a hand in a number of other Erlang projects: sync (automatic compilation), ChicagoBoss (MVC web framework), and qdate (date and timezone management). With Lloyd's help, he's learning how to write technical content that won't put the reader to sleep. Also, if you're ever in the Milwaukee area and looking for a beach volleyball partner, he's your guy.

Working Notes - Delete in Final Draft

```
git clone git://github.com/nitrogen/nitrogen
cd nitrogen
make slim_inets mv rel/nitrogen ~/nb
cd ~/wboard
git init
```

This will put the whole release and its included stand-alone erlang installation under source control.

If you want a smaller release that doesn't include the full release, you could do 'make slim_inets' in place of 'make rel_inets', then adding the whole release to source control is a little more manageable because it doesn't include the full erlang installation.

The basics are:

Making an HTML template, https://docs.google.com/presentation/d/19yw

Referencing your Template in in your erlang code: #template{file="pa

Working with simple nitrogen elements (#panel{}, #span{}, #button{}
http://nitrogenproject.com/demos/simplecontrols

Working with postbacks
http://nitrogenproject.com/demos/postback

Wiring page updates.
http://nitrogenproject.com/demos/ajax http://nitrogenproject.com/dem

After all that, probably circle back to describe the anatomy of a pa

And maybe end with a "bang" by showing how easy it is to do live-upd
http://sigma-star.com/blog/post/sync_panel

And you can try it with:

git clone git://github.com/nitrogen/NitrogenProject.com cd NitrogenP

That'll be a copy of the latest NitrogenProject.com code with my upd