# Build It with Nitrogen

## The Fast-Off-the-Block
## Erlang Web Framework

Lloyd R. Prentice
*&*
Jesse Gumm

# Praise

This book is exactly what I wish existed when I started with Nitrogen and Erlang for the web. Takes you by the hand and explains Erlang, Nitrogen, databases and much more. Everything comes together to make your own fast dynamic web site. Erlang and the web are a match made in heaven.
**Marc Worrell, creator, Zotonic CMS**

...love it. "Love it" is probably an understatement.
**Francesco Cesarini, Erlang Solutions**

Nitrogen simplifies development of web applications, making simple things easy and difficult things manageable. This book does the same. Projects present Nitrogen concepts and features in an iterative workflow that matches the way developers work. Developers will appreciate its many examples, its streamlined presentation, and numerous references to related material.
**Steve Vinoski, Yaws maintainer**

...a delightful and informative book that delivers the promise of the real-time web, combining colorful examples with battle-tested Erlang technology.
**Evan Miller, creator Chicago Boss**

An endless stream of real-world code samples on how to develop solid Erlang web applications with the Nitrogen framework. A real pleasure to read.
**Frank Müller, Erlang programmer**

I felt like the hero in an adventure story joining the characters and facing the challenges as a team. The fact that I learned about Nitrogen and web security was almost an after thought. I even learned how to extend Nitrogen with plugins! Serious learning wrapped in a fresh, fun package that kept me reading to the end.
**Alain O'Dea, Infrastructure and Security Manager, Sequence Bio**

. . . goes into great detail about Nitrogen and Erlang/OTP. Examples are simple yet functional with fictional dialogues bringing humor along the way. Nitrogen seems to be good for writing highly interactive Web applications entirely controlled from the server-side.
**Loïc Hoguin, creator Cowboy webserver**

Very fun book. Too many programming books are dry. Yours is just a joy to read.
**Alex Popov Jr.**

Cover design: Joseph DePinho, DePinho Graphic Design

Printed in the United States of America.

To Laurie, love of my life.

*–Lloyd*


To Jackie, my best half,

and to the memory of Paul Soik,

who cultivated me as a young programmer.

*–Jesse*


To Rusty Klophaus,

the late, great Joe Armstrong,

and other giants of Open Source.

*–LRP & JG*

# Forward

In the name of productivity, programmers will spend hours speeding up a five-minute task.[1]

Nitrogen is no exception. It began as a tool to speed up the development of a web application for a startup I envisioned back in 2008, and I spent a foolish amount of time building it. Happily, my efforts to blur the lines between front-end and back-end web development in Erlang resonated with others, and the framework quickly became more interesting than the startup. Soon, I shifted 100% of my focus to building Nitrogen, unsure of where that would lead.

My efforts proved fruitful. Nitrogen found a place in the toolboxes of other developers, and opened some great personal and professional doors for me, as well as gave me front row seats to the Erlang community during the the beginning of it's renaissance—I was fortunate to attend conferences and have conversations with Erlang heavyweights like Francesco Cesarini, Richard Carlsson, Ulf Wiger, Mickaël Rémond, and the late, great, great, great Joe Armstrong.

On a side note, that experience burned a crucial professional lesson into my head: "follow the traction."

During those early years as Nitrogen took shape, I received feedback and pull requests from countless generous developers, including both Lloyd Prentice in April 2009 and Jesse Gumm in June 2009. The Erlang community is notoriously positive, and each interaction helped push Nitrogen in a better direction.

Eventually, my work on Nitrogen came to an end. In June 2011, I reached out to Jesse asking if he would take over as the Nitrogen project lead. That was my first time handing off ownership of an open source project and, I imagine, Jesse's first time accepting ownership of one. Neither of us had any clue of The Right Way to do what we were doing. But luckily, Jesse agreed to sign up for

---

[1]Case in point: `https://xkcd.com/1319/`

this adventure. On a side note, Jesse was by then hard at work building his startup BracketPal.com. If you want something done, ask a busy person.

As of September 2019, Jesse has guided the Nitrogen project longer than I have, and has been the model of an open source project leader. He has corralled contributors, issued regular releases to stay current with the latest best practices, and maintained an active user base, all while following the convention of the existing code as much as possible and generously deflecting credit toward wherever it is most deserved, whether that is to me or other contributors.

This book is the next step in Jesse's journey as project lead, and Nitrogen's journey as a web development framework. Jesse and Lloyd have teamed up to put together a real treasure of a guide—something informative, practical, and entertaining.

This book is ambitious. It covers not only Nitrogen and Erlang, but also a bit of OTP, databases, git, web design, and software engineering best practices. And somehow, it does this well by building up from simple concepts in a sort of "Socratic dialogue"—if Socrates had a "California surfer" sense of humor.

Most importantly, the lessons in the book are the result of real experience running Nitrogen in a real production environment. Jesse's startup, BracketPal.com, was built on Nitrogen and was recently acquired by SportsEngine, an NBC company. In that light, the book addresses tech-debt and trade-offs clearly and carefully.

When I began hacking on Nitrogen I hoped for nothing beyond scratching my own development itch. Now I have the joy of reading a book about technology that I helped create, and the pleasure of learning a thing or two in the process. I hope you enjoy reading this book as much as I have, and I hope that learning about Nitrogen sparks in you the desire to build something great.

*Rusty Klophaus*
Creator of Nitrogen
rusty.io

# Authors Note

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

Here's what we came up with:

- You are comfortable with the Unix/GNU Linux command line.

- You have experience building websites; 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.

- But we do assume that you have a recent version of Erlang installed on your computer. Check out Chapter 2 if not.

- Our last assumption is that you're passionate about building reliable, scalable web applications using high-productivity tools.

- Note to wizards: No condescension intended if we cover stuff you know.

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*

# Contents

# Before we begin

- You'll find many code listings throughout this book. Computer display is in normal type; the commands you enter are in **bold**.

- Some URLs and directory paths are too long to fit on one line. 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 the two lines when you paste them into your browser or use them in code.

- We assume that you are editing code with Vim but, of course, any competent text editor will do the job. That's why we occasionally use ZZ at the end of an editing sequence to signify save and quit.

- You'll create many Nitrogen modules throughout this book. While we haven't done so for space reasons and to avoid annoying repetition, we strongly urge you to start each module with a comment block. Here's an example:

  ```
  %%%------------------------------------------------
  %%% @author Lloyd R. Prentice and Jesse Gumm
  %%% @copyright 2019 Lloyd R. Prentice and Jesse Gumm
  %%% @doc Associates directory admin page
  %%% @end
  %%%------------------------------------------------
  ```

  Erlang has a wonderful built-in documentation tool called EDoc that will pick up and pretty print this information and much more.[2]

---

[2] http://erlang.org/doc/apps/edoc/chapter.html

*Contents*

- In the same vein, we've omitted in-line documentation of functions and type-specs.[3] But, when you feel ready to write robust, maintainable, production-quality code, thorough documentation of functions and type-specs should be standard operating procedure.

- CAUTION: In many exercises you will have several windows open on your monitor. These might include one or more UNIX or Linux shells, an editor such as Vim, and an Erlang Shell. We've provided breadcrumbs in the header of each listing to alert you as to which window or terminal you'll be working in or starting in: ($\Rightarrow$ `$`) denotes a UNIX shell; ($\Rightarrow$ `vim`) denotes Vim[4] or other editor[5] of your choice; and ($\Rightarrow$ `>`) denotes an Erlang shell.

- As you enter project code module by module, function by function, it'll be all too easy to skip a step or introduce typos. The more code you enter without testing, the more likely bugs will compound. The more bugs, the greater the debugging hassle and frustration you'll experience when it comes time to view your work in the browser.

- You'll learn how to compile Erlang modules and view your work in the browser in Chapters 2 and 3. Both steps are dead easy, but make a note of the sequence. You'll save time in the long run if you compile and test frequently.

- After you enter each function, we encourage you to confirm that the module you are working on will compile successfully. Some functions require dependencies. In this case compilation will fail. But the error message will clearly show what's missing and any other bugs you may have inadvertently entered.

---

[3]http://erlang.org/doc/reference_manual/users_guide.html

[4]Vim (https://www.vim.org) is the preferred editor of your authors.

[5]Other common editors are Emacs (https://www.gnu.org/software/emacs/), the most popular editor for Erlang, and Nano (https://www.nano-editor.org/), a simple, and easy-to-use editor.

## 0.1. New to Erlang?

Welcome!

If Erlang is *terra incognito*, prepare for a rewarding journey. But be advised. Erlang may seem strange at first. Yet, as many explorers before you have discovered, Erlang will significantly expand your programming horizons.

For fast fly-over, take time now to review Appendix A—*Erlang from top down.*

We've listed informative resources in Appendix B. Joe Armstrong's *Programming Erlang*, 2nd edition would be a high-payoff investment.

We point to invaluable web resources thrughout this book.

Here are a few tips to help you kick off on the right foot:

- You can find an overview of Erlang data types here:

  https://erlang.org/doc/reference_manual/data_types.html

- Erlang is a functional programming language that uses single-assignment for variables. This means that unlike procedural languages, once a variable is assigned (or "bound," in Erlang parlance), it cannot be re-assigned; e.g. X = X + 1 is meaningless and would throw a run-time error. But never fear, single assignment has distinct advantages and poses few obstacles to fluent development.

- Variables all start with either an upper-case letter or an underscore.

- Assigned variables that are unused will throw warnings unless they start with _.

- _ is called an "anonymous" variable. It never has a value assigned to it, and but matches everything. It's designed for situations where a variable is required, but its value can be ignored.

- An atom is a literal, that is, a constant with a name. An atom begins with a lower-case letter or, if it contains characters other than alphanumeric characters, underscore (_), or @, then it is enclosed in single quotes ('). e.g. undefined, node@server2, 'This is not a string', '$end_of_file'.

*Contents*

- Functions within the same module are called with
  `<function name>(Args)`. e.g. `get_value(Arg1, Arg2)`

- Functions from other modules are called with
  `<module name>:<function name>(Args)`. e.g. `lists:member(X, Items)`

- The number of arguments required by a function is called "arity," denoted
  as `<function name>/N` where `N` is the arity. Two functions with the same
  name but different arity are entirely different functions.

- Functions are defined in modules. The `-export` attribute exposes functions
  to other modules; e.g. `-export([start/0])`. The attribute
  `-compile(export_all)` exports all functions defined in the module.

- Finally, Erlang code is loosely structured like a series of sentences:
  - Individual instructions end with a comma (`,`)
  - Separate clauses end with a semi-colon (`;`)
  - A function ends with a period (`.`)

In our experience the Erlang community is smart, creative, gracious, and
helpful. When stumped, reach out:

`https://www.erlang.org/community/mailinglists`

# Part I.

# Frying Pan to Fire

# 1. You Want Me to Build What?

**BOSSMAN:** 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 availability.[1]

Hard to learn? Excellent resources in Appendix B. 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. They're 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.

---

[1] https://en.wikipedia.org/wiki/High_availability

# 2. Enter the Lion's Den

**JESSE:** 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 Ubuntu 18.04. Dual-core, gig of RAM. Company up the road traded up so Bossman picked these puppies up for 50 bucks apiece. Bossman likes to stay lean-and-mean. Truth—the dude's a cheapskate.

Yes, we could use Kubernetes, 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 20.04, a six-core processor, and sixteen 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.

Developing web applications boils down to managing a jumble of languages and network protocols.

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 conveying digital bits cunningly structured by HTML, CSS, and JavaScript to convey natural language, sound, and images both still and moving.

The server responds to some babel of computer languages—HTTP/HTTPS, HTML, CSS, and JavaScript—to transcribe natural language, sound, and images into cunningly structured digital bits and marshal them into the web of data pipes called the Internet, where, with luck and brilliant engineering, they can reach the client's browser.

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 harnesses 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 you have that much less to think about when you craft your awesome 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 deploys Erlang records to structure data, Erlang functions to execute logic and embed JavaScript, and the Erlang development platform to organize and abstract 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, the simplest solution would be to download the "Standard" binary package from Erlang Solutions here:

```
https://www.erlang-solutions.com/resources/download.html
```

If you're feeling saucy, you could follow the instructions from the excellent Erlang resource, Adopting Erlang, which is a rather comprehensive guide to building on common platforms:

```
https://adoptingerlang.org/docs/development/setup/
```

If you really want to get into the nitty-gritty of installation and building Erlang, you can also follow official documentation, but we generally recommend this only for advanced users.

```
http://erlang.org/doc/installation_guide/INSTALL.html
```

Sometimes installing Erlang from source or via your system's package manager can be problematic (for example, the default erlang on Ubuntu is notoriously incomplete). So we recommend either downloading the "Standard" distribution from Erlang Solutions, or following the instructions from AdoptingErlang.org.

Once you have Erlang running on your system, Nitrogen is super easy to compile. Let's clone it from GitHub; make a test project called testproj:[1]

---

**Listing 2.1 ( ⇒ $) Clone and Make**

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

---

[1]If Git is not already installed, download it here: https://git-scm.com/downloads

*2. Enter the Lion's Den*

All that text scrolling up the terminal? That's GNU `make` and Rebar 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`.

`https://www.lifewire.com/make-linux-command-unix-command-4097054`

Looks like we're done:

---

**Example 2.1   Start Nitrogen**

```
...
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:

---

**Listing  2.2   (⇒ $) Start console**

```
 /nitrogen$ cd ../testproj
 /testproj$ bin/nitrogen console
...
Erlang/OTP 19 [erts-8.3.5] [source] [64-bit] [smp:3:3]
[async-threads:5] [hipe] [kernel-poll:true]
...
Eshell V8.3.5  (abort with Ĝ)
...
(testproj@127.0.0.1)1>
```

---

The 1> prompt 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`**

As me Cockney-speaking mates would put it, Bob's your uncle! "WELCOME TO NITROGEN" in your very own browser.

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'm baaack!

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

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

---

**Listing  2.3   (⇒ $) Explore site directory**

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

---

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

Let's look into the templates directory:

**Listing 2.4** ( ⇒ $) **Explore templates directory**

```
~/testproj/site$ cd templates
~/testproj/site/templates$ ls -l
bare.html
mobile.html
```

A peek in on `bare.html` will reveal a standard HTML file. We're partial to Vim around here[2], but you can develop Nitrogen applications in whatever code editor you prefer:

**Listing 2.5** ( ⇒ $) **Review default template**

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

As you see, `bare.html` is loading a bunch of JavaScript files and style sheets in the head section. You've built websites, so there's nothing here that you haven't seen before.

But, in the body section, we see:

**Example 2.2** **Snippet of body from template**

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

And there, my friend, is Nitrogen's secret sauce. We'll unveil the tantalizing mysteries by-and-by.

---

[2] Most of the Erlang community prefers Emacs, but the authors are oddballs and prefer Vim. There are Vim Erlang extensions available emulate Emacs `erlang-mode` for autoindentation, error detection, etc. See page 17 for how to configure Emacs and Vim to work with Nitrogen source code in a friendly way.

Close out the templates directory, bop into the ebin[3] directory, and list it:

**Listing 2.6 ($\Rightarrow$ $) Review ebin directory**

```
~/testproj/site/templates$ cd ../ebin
~/testproj/site/ebin$ ls -l
index.beam
mobile.beam
nitrogen.app
nitrogen_app.beam
nitrogen_main_handler.beam
nitrogen_sup.beam
```

Note nitrogen.app.

App files are a very big deal in Erlang.[4] The instance in the ebin directory was automatically generated by nitrogen.app.src in the src directory. We'll get to that in a moment.

Next note all the *.beam files.

Say the greybeards, beam stands for Bogdan's Erlang Abstract Machine.[5]

Like Forth and Java, Erlang runs on a virtual machine. Erlang source compiles down to *.beam files and the *.beam files (normally just referred to as beams) execute on the 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 the src directory:

---

[3]With the release of Nitrogen 3 (which will use Rebar 3), this directory will actually be located deep in the _build directory. The full path would be something like _build/default/rel/APPNAME/lib/APPNAME-X.Y.Z/ebin

[4]http://erlang.org/doc/man/app.html

[5]https://en.wikipedia.org/wiki/BEAM_(Erlang_virtual_machine)

---

**Listing 2.7  (⇒ $) Review src directory**

```
~/testproj/site/ebin$ cd ../src
~/testproj/site/src$ ls -l
actions
elements
index.erl
mobile.erl
nitrogen_app.erl
nitrogen.app.src
nitrogen_main_handler.erl
nitrogen_sup.erl
~/testproj/site/src$
```

---

Compare the content of `nitrogen.app.src` with `nitrogen.app` in the `ebin` directory. App files provide metadata that tell the Erlang compiler where to find resources that the application needs, such as start and stop functions.

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.

Note also how the `*.erl` files in the `src` directory have doppelgängers in the `ebin` directory.

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

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

Squint while you eyeball the `*.erl` files. The structure and names 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://erlang.org/doc/design_principles/applications.html

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`:

**Listing 2.8** (⇒ `vim`) **View index.erl**

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

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

Hot diggity! Here's the code that produced the Nitrogen home page that's displayed in our browser!

But hey—time to refuel. We'll tackle the Bossman gig after lunch. But real quick before we head out, let's talk about editors.

## 2.4. Editors

Nitrogen officially supports Emacs and Vim and provides extensions or modules for them. Other editors are sure to work just fine, but you won't benefit from custom Nitrogen extensions.

Why custom extensions for Nitrogen? Nitrogen page modules are just Erlang modules. Good Erlang code doesn't require much nesting, which makes standard Emacs `erlang-mode` and its variants very useful. Erlang code formatted with `erlang-mode` might look something like this:

**Example 2.3   Standard Erlang indentation**

```
MyList = [
         some_list_item,
         another_item,
         holy_moly_even_more_items_wow
         ],
```

This is fine until you start writing Nitrogen code. Nitrogen elements often result in several levels of nesting due to the way Erlang records abstract HTML code.

17

As a result, with standard Erlang-mode, you might end up with something like this:

---

**Example 2.4    Nitrogen with standard Erlang indentation**

```
Elements = #panel { body=[
                          #span { text="Hello, World!" }
                          ]},
```

---

Looks bad. And that's just one level of nesting. Imagine how horrible this code will look with two or three levels of nesting.

Fortunately, we have helper code to help out. Here's how `nitrogen-mode` for Erlang formats the code:

---

**Example 2.5    Nitrogen-recommended Indentation**

```
Elements = #panel{body=[
    #span{text="Hello, World!"}
]},
```

---

### 2.4.1.  Emacs

Nitrogen provides Nitrogen-mode for Emacs. You can find the installation instructions in:

`~/nitrogen/support/nitrogen-mode.`

For Nitrogen-mode to work, your page must have the following line at the top:

---

**Example 2.6    Specifying nitrogen-mode for Emacs**

```
%% -*- mode:  nitrogen -*-
```

---

## 2.4.2. Vim

Vim's support for Nitrogen formatting is simple to configure.

From the base of the Nitrogen directory, run:

**Listing 2.9   Installing Vim config for Nitrogen**

```
...nitrogen$ make install-vim-script
```

This installs simple Vim rules into your `~./vimrc` file. These prevent the auto-indenting of Vim's Erlang extensions from applying to Nitrogen files, but retain syntax highlighting and other goodies.

To apply these rules, add the following to the top of your Nitrogen pages:

**Listing 2.10   Specify Vim modeline for Nitrogen**

```
%% vim:  ft=nitrogen
```

---

**Mixing Erlang and Nitrogen Files**

The mode lines for Vim or Emacs will serve you well on your Nitrogen page modules. But there's no need for them on non-Nitrogen Erlang modules. Best to leave them off non-Nitrogen modules.

---

# Part II.

# If You Can't Run, Dance!

# 3. nitroBoard I

**JESSE:** Specs? From Bossman? You kidding? Typical client—expects developers to be mind-readers. But no worry. We'll brainstorm.

When visitors drop into front office, what do they need to see on the Welcome Board?

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.

Do we need a visitors' database?

Yeah, we do, but simple simple.

OK, what else?

Hmmm—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. Kill, Baby, Kill!

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

Why? It's hogging port 8000. We'll need that puppy by and by.

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:

---

**Listing 3.1** ($\Rightarrow$ >) **Kill the server**

```
...
Eshell V8.3.5 (abort with ^G)
1> q().
ok
2> jesseBob:~/testproj/site/src$
```

---

Promising. Now refresh your browser:

---

**Example 3.1 Is it dead yet?**

```
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 Erlang docs to discover others. Don't forget that every Erlang shell command must be terminated with a period before the command will execute.

Of course, in the event the Erlang virtual machine or shell becomes non-responsive—like say you accidentally type an infinite loop into the shell—there's always the tried-and-true, kill-it-with-fire method: CTRL+C.

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

Before we create a new project we need to decide whether to compile a full or slim release and on which webserver. When we first installed the Nitrogen demo, we entered the following command:

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

Result: Erlang created a new project, configured as a full release on inets, Erlang's built-in webserver. The release included the Erlang Runtime System, known affectionately in the Erlang world as ERTS, and all else required to run the demo. Tar up a full release, ship it to another 'ix system, untar it, and it will run without the bother of installing Erlang separately.

Here's bedtime reading for fun and profit:

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

Turns out, if Erlang is installed on the target system, you won't need to include the full ERTS in your release. In this case you can compile what's called a slim release.

We'll go for slim here.

Added bonus: Nitrogen offers a selection of webservers: Yaws, Cowboy, MochiWeb, Webmachine, and inets.[1] Since we expect nitroBoard to experience very light loads, we'll stick with inets. Also, let's call this project nb[2] for nitroBoard. Thus, we enter:

**Listing 3.2** ($\Rightarrow$ \$) **Make a new project for nitroBoard**

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

---

[1] http://yaws.hyber.org/
   https://github.com/ninenines/cowboy
   https://github.com/mochi/mochiweb
   https://github.com/Webmachine/webmachine
   http://erlang.org/doc/man/inets.html
[2] Full code for nb is here: https://github.com/BuildItWithNitrogen/nitroboard1

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/
```

Ask nicely and Rusty will bring you up to speed on our Git workflow in Appendix C. When you've finished your app, you can post it on GitHub:

---

**Listing 3.3** ( ⇒ $ ) **Initialize Git**

```
~/nb$ git init
Initialized empty Git repository in /home/jess/nb/.git
~/nb$ ls -al
...
bin
BuildInfo.txt
do-plugins.escript
etc
slim-release
git
lib
log
Makefile
plugins.config
rebar
rebar.config
releases
site
```

---

Way cool!

Much to be learned in the `nb` directory—in particular, the `bin` and `site` directories 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, where Git stores version records.

For now, let's bring up the demo source and morph 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:

**Listing 3.4 (⇒ >) Start Nitrogen**

```
~/nb$ bin/nitrogen console
...
(nb127.0.0.1)1>
```

There's ye olde Erlang shell. It should look familiar. 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 up for cosmetic surgery.



Now, *in a new terminal*, that is, the Unix shell with the $ prompt, open up the site directory. You now have two terminals open. We'll work in the Unix shell:

---

**Listing 3.5** (⇒ $) **Back to the site directory**

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

   ebin
   include
   src
   static
   templates
```

---

This should also look familiar.

And so, Nurse Ratched, 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 provides an index page that we can morph into our welcome board. Indeed, you're looking at it in your browser. From your working terminal bop into the src directory and open up index.erl:

---

**Listing 3.6** (⇒ $) **Open index.erl in your editor**

```
%% -+- mode: nitrogen -*-
~/nb/site$ cd src
~/nb/site/src$ vim index.erl
```

---

The index.erl file is a plain vanilla Erlang module. Cast your eye on the function inner_body/0. We'll make minor changes to it in a moment. But first, shift attention back to the Erlang shell and enter the following:

**Listing 3.7** ( ⇒ >) **Start sync:go/0**

```
(nitrogen127.0.0.1)1> sync:go().
Starting Sync (Automatic Code Compiler / Reloader)
...
ok
(nitrogen127.0.0.1)1>
```

What happened here?

The Erlang function sync:go/0[3] tracks and automatically compiles the changes 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:

**Example 3.2** ( ⇒ $) **inner_body/0**

```
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{},
        ...
```

---

[3]Sync is an application for automatically recompiling and loading changes to Erlang code. 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

## 3. nitroBoard I

Make the following changes:

---

**Listing 3.8** ($\Rightarrow$ `vim`) **Revise index.erl**

```
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.
      ",
      #h1{ text="Erlingo!  WEBSPINNERS" },
      #h1{ text="WELCOME!" },
      #h2{ text="Rusty Klophaus" },
      #h2{ text="Jesse Gumm" },
      #p{},
      #button { id=button, text="Click me!",
                postback=click},
      #p{},
      "
      Run <b>./bin/dev help</b> to see some useful developer commands.
      ",
      #p{},
      "
      <b>Want to see the ",
      #link{text="Sample Nitrogen jQuery Mobile Page",
               url="/mobile"},"?</b>
      "

   ].
```

---

Save your changes, cast your eye on the Erlang shell, and behold the helpful handiwork of `sync`:

**Example 3.3 index.erl recompiled**

```
=INFO REPORT==== 17-May-2019::14:32:48 ===
/home/jesse/nb/site/src/index.erl:0: Recompiled.
=INFO REPORT==== 17-May-2019::14:32:48 ===
index: Reloaded! (Beam changed.)
```

Refresh your browser and, by gum![4] To the delight of our eyes, your changes have been compiled!
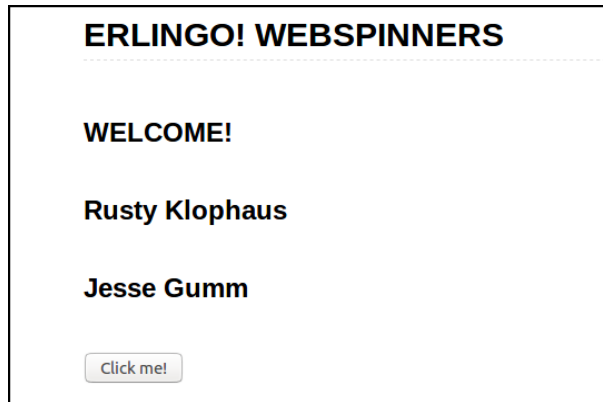


OK, we admit, this screen ain't pretty. But it gives us a canvas to work on.

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—rather, it provides the source code for building an HTML page.

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

---

[4] "by gum," not to be confused with *Gumby*, which is a totally different thing.

**Example 3.4   index.erl**

```
%% -{*}- mode: nitrogen -*-
%% vim: ft=nitrogen

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

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

The first two lines are comments. The first comment enables nitrogen-mode for Emacs; the second tells vim that this is a "nitrogen" filetype. Percent symbols at the beginning of the line give it away.

The next three lines are attributes signaled by hyphens at the start of the line. 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 `.erl` extension. This is also mandatory in Erlang. Note also that the 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. Every function in `index.erl` is exported so can be called from other modules.

If we wished to export a subset of functions and keep others private, we'd use:

   `-export([<export1/n1>, <export2/n2>,...]).` [5]

The third attribute, `-include_lib`, imports the `wf.hrl` file from `nitrogen_core`. The `nitrogen_core` library is the engine that powers Nitrogen. It's included as a library of your main application. Usually, `*.hrl` will import one or

---

[5]It's preferred practice in Erlang to use the attribute `-export` to clearly identify functions that can be exported. `-compile(export_all)` is convenient for development and testing, so we'll use it here. But it would be wise to replace it with `-export` before releasing the module for production. Indeed, using `-export` explicitly can clue you into dangling functions - functions that are no longer called within a module. A Nitrogen page must export `main/0`, but most will also export `title/0`, `body/0`, `event/1`, as well as any functions that must be available to postbacks or template callouts.

more record definitions. The `nitrogen_core/include/wf.hrl` file specifically includes all the built-in Nitrogen elements.

Now note that `index.erl` has five functions:

```
main/0
title/0
body/0
inner_body/0
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 will gain a fair understanding of each function's purpose. 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?**

In Erlang we sometimes call functions by "arity;" e.g., `body/0`, `sync:go/0`, etc. Arity means "the number of arguments passed to a function." So `sync:go/0` takes zero arguments. It can also be called as `sync:go()`. The function `lists:map/2` takes two arguments. It can be called as `lists:map(SomeFunction, SomeList)`. You can assign functions to variables with in format:

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

The first phrase assigns `my_module:some_fun/3` to the variable `MyFunction`. The second phrase demonstrates how `MyFunction` can then be invoked in exactly the same way as `my_module:some_fun/3`.

---

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 for a typical page:

1. User hits a URL.

2. URL is mapped to a module.

3. Nitrogen framework calls `module:main()`.

4. `module:main()` typically returns a `#template{}` element.

5. The `#template{}` is sent to Nitrogen's rendering engine.

6. The template includes raw HTML mixed with callouts back into the page module (in the form of `[[[page:some_function()]]]`).

7. Those functions return other Nitrogen elements.

8. Those elements are also run through Nitrogen's rendering engine, converting all elements into HTML and Javascript.

9. This process continues until all elements have been converted to HTML and Javascript.

10. Nitrogen sends the rendered output (HTML and Javascript) to the browser.

Woohoo! We're rockin' 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's an extension, assumes a static file:

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

- Nitrogen replaces slashes with underscores:

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

- Nitrogen tries the longest matching module:

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

- If a module is not found, go to `web_404.erl` if it exists.

- The underlying platform handles static files that aren't found (not yet generalized).

This suggests that nitroBoard will have at least three pages:

```
index.erl
visitors_admin.erl
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
```

## 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 an Erlang list in a list in a list:

```
[[[module:function(Args)]]]
```

But don't be fooled. They're pure Nitrogen—`module:function(Args)` is an Erlang function call. It returns a result and plugs into the template. A call to the module `page` refers to the current page.

## 3.7. Elements

A Nitrogen element is simply HTML or an Erlang record that renders to HTML. Here's what you need to know about Erlang records:

- When compiled, a record is a plain vanilla Erlang tuple. But the record definition provides the compiler with enough information to enable the programmer to address fields in the tuple by name.

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

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

- Tuples may be defined with any number of fields, but once defined, the number of fields, that is, length, cannot be increased or decreased. Problem is, if a tuple gets too long, you get confused as to which chunk of data goes into which field.

- An Erlang record is, arguably, a hack to solve this problem. In source, an Erlang record is defined as:

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

More here:

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

So, back to Nitrogen elements. 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` – Set the name of an element.

- `actions` – Add Actions to an element. Actions will be 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.
```

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

- Layout (templates, page grid, etc.)

- HTML (links, lists, images, etc.)

- Forms (fields, buttons, dropdowns, etc.)

- jQuery Mobile elements

- RESTful Form Elements

- More (drag and drop elements, wizards, spinners, progress bars, etc.)

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
```

Every Nitrogen element can be mapped into:

1. An Erlang tuple

2. HTML

Give it a shot. In your infinite free time, trace through the Nitrogen source code. See if you can map element source code into respective tuples and HTML. You can find source code here:

```
https://github.com/nitrogen/nitrogen_core/tree/master/src/elements
```

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

## 3.8. Actions

A Nitrogen action can be either 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 that a few elements contain helper attributes called `click` (most notably `#link{}` and `#button{}`), so you don't have to use the actions attribute for extremely common cases:

```
#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 affects.

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" } ]
```

Here are two function calls that return identical results. When you click the Submit button in either case, the label will pulsate. Note the differences in syntax:

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

As you see, you can specify the `trigger` and `target` directly in `wf:wire/N`. This can take several 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)`

### 3.9.1. The big picture

1. Elements make HTML.

2. Actions make JavaScript.

3. An action can be wired using the action's 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—scope out those bloodshot eyes. Up all night?

Marketing sent us a corporate logo file. It's called erlingologo.png. I've taken the liberty of storing it in `~/nb/site/static/images`.[6]



---

[6]You can clone the Erlingo! logo here: `http://builditwith.com/images/erlingologo.png`. Or, you can make your own *.png image and fake it. But be sure to copy the logo image into .../nb/site/static/images.

*3. nitroBoard I*

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. Indeed, that's exactly what templates are for—to save source duplication.

So let's do it.

In your Unix shell, open bare.html:

---

**Listing  3.9   (⇒ $) Edit the template**

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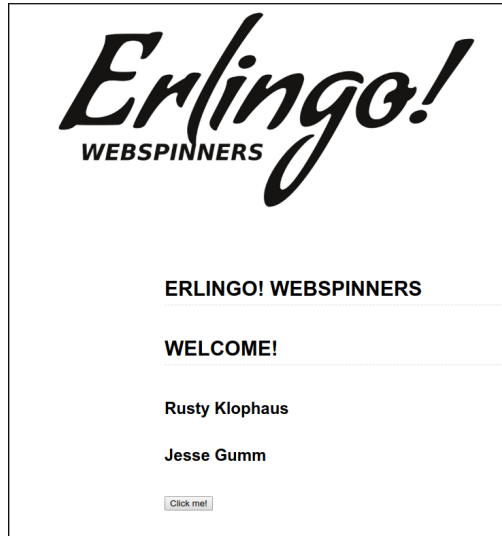~/nb/site/src$ cd ../templates
~/nb/site/templates$ vim bare.html
```

---

And, after the opening <body> tag, add the following code:

---

**Listing  3.10   (⇒ vim) Add HTML to the template**

```
<body>
    [[[page: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>
```

---

So, good to go?

Not quite. We need to make slight changes to our homepage (`index.erl`). If, by chance, you closed your Erlang shell, re-open it and enter:

~/nb$ **bin/nitrogen console**

Don't forget to start `sync` so that we can take advantage of Erlang's sweet auto-code reloading:

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

Now, in the UNIX terminal, open `index.erl`:

**Listing 3.11** (⇒ $) **Edit index.erl yet again**

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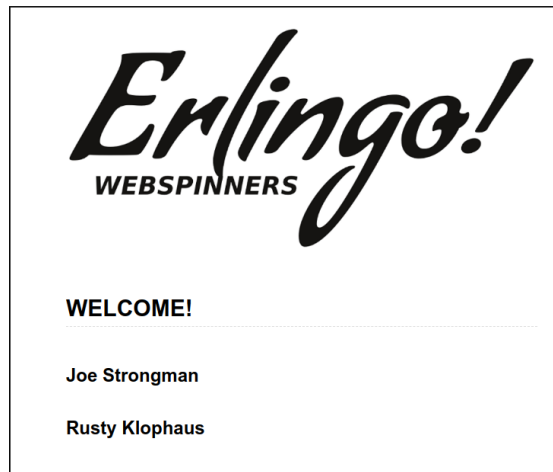~/nb/site/templates$ cd ../src
~/nb/site/src$ vim index.erl
```

Next, morph `index.erl` for real. Delete `event(...)` and `inner_body()` and revise `body()` to look like this:

---
**Listing 3.12** (⇒ `vim`) **Revise body()**

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

Save and call up your browser:

> **localhost:8000**



And—Bingo!

That was easy. With a touch of CSS we could reposition the welcome line to make it more attractive, but we'll do that later. Let's focus on 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. This suggests 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 before 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
```

### 3.11.1.  Visitor record

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

---

**Listing 3.13** (⇒ $) **New include file**

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

---

Insert the following line:

---

**Listing 3.14** (⇒ `vim`) **Our very first record**

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

---

Then 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 the file `nb.hrl`, which 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 enables us to create and retrieve visitor records. Let's do this in a new module called `visitors_db.erl`:

---

**Listing 3.15** (⇒ $) **Open visitors_db.erl**

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

---

Now insert the following:[7]

---

[7]It's good practice to include header comments at the beginning of every module. At minimum, show author, copyright, and a brief note specifying the function of the module. Where we've left them out, it's simply to save you a bit of typing.

> **Listing 3.16** (⇒ `vim`) **visitors_db.erl**
>
> ```
> -module(visitors_db).
> -include("nb.hrl").
> -compile(export_all).
> ```

You may be wondering why we didn't include `wf.hrl`, that is:

```
-include_lib("nitrogen_core/include/wf.hrl"),
```

Well, we're not coding a Nitrogen page. We're coding a normal Erlang module for retrieving items from the database, so no need to include Nitrogen element record definitions—wouldn't hurt if we did, but why bother?

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

> **Listing 3.17** (⇒ `$`) **Fire up Erlang shell**
>
> ```
> ~/nb/site/src$ cd ../
> ~/nb/site$ erl -pa ebin
> . . .
> Running Erlang
> Eshell V6.0  (abort with ^G)
> 1>
> ```

The `erl -pa ebin` command opens the Erlang shell and puts `ebin` in the code path. To work with records in the shell, first read them in:

> **Listing 3.18** (⇒ `>`) **Read record definitions into the shell**
>
> ```
> Eshell V6.0  (abort with ^G)
> 1> rr(visitors_db).
> [visitor]
> ```

The `rr` function reads all record definitions included in the `visitors_db.erl` module. Here's how we can examine the definition of visitor:

**Listing 3.19** (⇒ >) **Examine the definition of a record**

```
2> rl(visitor).
-record(visitor,{date,time,name,company}).
ok
```

This raises a question: How should we format date and time?

Erlang has a calendar library that can help. The following sequence of Erlang shell commands points the way:

**Listing 3.20** (⇒ >) **The calendar library**

```
3> {Date, Time} = calendar:local_time().
{{2019,5,20},{16,6,57}}
4> Date.
{2019,5,20}
5> Time.
{16,6,57}
```

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`. The variable `Date` is represented as a tuple: `{Year, Month, Day}`. The variable `Time` as: `{Hour, Minute, Second}`.[8]

Moving on, let's create a record:

---

[8]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

**Listing 3.21** (⇒ >) **Instantiate a record in the shell**

```
6> V1 = #visitor{date=Date, time=Time,
     name="Jesse James", company="Erlingo!"}.
#visitor{date = {2019,5,30}, time = {11,25,54},
         name = "Jesse James", company = "Erlingo!"}
```

We can retrieve data from this record in three ways.
By field:

**Listing 3.22** (⇒ >) **Retrieve record attribute by field name**

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

Through pattern matching:

**Listing 3.23** (⇒ >) **Use pattern matching to retrieve record attribute**

```
8> #visitor{date=Date1, time=Time1, name=Name,
         company=Company} = V1.
9> Date1.
{2019,5,30}
10> Time1.
{11,25,54}
11> Name.
"Jesse James"
12> Company.
"Erlingo!"
```

Or, with the Erlang BIF element/2[9]

---

[9]Built-in Function—Note that you'll rarely need to retrieve data from a record with element/2.

---

**Listing 3.24** (⇒ >) **Use element/2 to retrieve a record attribute**

```
13> Company2 = element(#visitor.company, V1).
"Erlingo!"
```

---

The BIF `element/2` retrieves the Nth element of a tuple. So `element(2, {a,b,c})` would return `b`. Further, the compiler takes `#visitor.company` and converts it to the integer that denotes which element of `#visitor` is represented by the company parameter. That integer is passed along with the record `V1` to the `element(N, Record)` function to retrieve the appropriate value from that field. It's not typically done, but every once in a while, you'll find a need for it.

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

```
https://stackoverflow.com/questions/52713006/what-is-
the-need-for-immutable-persistent-data-structures-in-
erlang
```

But where should we store our records? Erlang delivers just the ticket—`Dets`.

## 3.12. Persistence

`Dets` provides disk-based 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 and specify the key position and table type:

---

**Listing 3.25** (⇒ >) **Open Dets**

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

---

So what's {type,bag}? Check the Dets man page.
Now we can play:

---

**Listing 3.26** (⇒ >) **Play with Dets**

```
15> dets:insert(visitors, V1).
ok
16> dets:lookup(visitors, Date).
[#visitor{date = {2019,5,30}, time = {11,25,54},
     name = "Jesse James", company = "Erlingo!"}]
17> V2 = #visitor{date=Date,time={12,20,11},
name="Rusty Scupper",company="Erlingo!"}.
18> dets:insert(visitors,V2).
ok
19> V3 = #visitor{date={2019,6,1}, time={14,0,0},
          name="Joe Strongman"}.
#visitor{date = {2019,6,1},
        time = {14,0,0},
        name = "Joe Strongman",
        company = undefined}
```

---

*Excelente!*
Let's see who's coming in on May 30, 2019:

---

**Listing 3.27** (⇒ >) **More lookups**

```
20> Visitors = dets:lookup(visitors, Date).
[#visitor{date = {2019,5,30},
        time = {11,25,54},
```

---

```
          name = "Jesse James",
          company = "Erlingo!"},
 #visitor{date = {2019,5,30},
          time = {12,20,11},
          name = "Rusty Scupper",
          company = "Erlingo!"}]
```

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

Close out the database for now.

**Listing 3.28** (⇒ >) **Close Dets database**

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

### 3.12.1. Format visitor data

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

The May 30, 2019 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:

**Listing 3.29** (⇒ >) **Instantiate a record**

```
22> V4 = #visitor{date={2019,5,21}, 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 them—so we can format the name like this:

**Listing 3.30** (⇒ >) **Format name from our visitor record**

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

Ah, puzzled by what's going on in line 25?

You can see that it's a list that includes a variable, a string, and another variable. So what good does that do us?

A bunch!

It's what's known in Erlang circles as an `IO List`[10] (or just "iolist").

    https://prog21.dadgum.com/70.html

When Erlang sends an `IOList` through standard output or a network socket it will conveniently concatenate all the terms, converting them ultimately to a stream of bytes. 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

---

[10]We talk about `IO Lists` in detail on page **??**

So let's look at how to do this with our list of visitors using an Erlang list compre-
hension and an anonymous function[11] assigned to the variable FormatVisitor:

**Listing 3.31** ($\Rightarrow$ >) **Format Visitors**

```
26> FormatVisitor=fun(V) ->
      [V#visitor.name,"-",V#visitor.company]
    end.
#Fun<erl_eval.6.54118792>
27> FormattedVisitors = [FormatVisitor(V) || V <-
      Visitors].
[["Jesse James"," - ","Erlingo!"],
 ["Rusty Scupper"," - ","Erlingo!"]]
```

Read the list comprehension right-to-left and it should be transparent. Note the
result—a list of two IOLists.

### 3.12.2. Visitors database

Reopen visitors_db.erl in the UNIX terminal:

**Listing 3.32** ($\Rightarrow$ $) **Edit visitors_db.erl**

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

And now, let's add open_visitors_db/0:

**Listing 3.33** ($\Rightarrow$ vim) **Open visitors_db.erl**

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

---

[11]https://www.tutorialspoint.com/erlang/erlang_funs.htm

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

We'll also need to close the database:

**Listing 3.34** (⇒ `vim`) **close visitors_db.erl**

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

close_visitors_db() ->
   ok = dets:close(visitors).
```

We'll definitely need put and get functions. Here's put:

**Listing 3.35** (⇒ `vim`) **put function**

```
close_visitors_db() ->
   ok = dets:close(visitors).

put_visitor(Record) ->
    open_visitors_db(),
    ok = dets:insert(visitors, Record),
    close_visitors_db().

%% @doc Enter VIP visiting today and store in db
put_vip(Name, Company) ->
    {Date, Time} = calendar:local_time(),
    Record = #visitor{date=Date, time=Time,
        name=Name, company=Company},
    put_visitor(Record).
```

And here's get:

**Listing 3.36** ($\Rightarrow$ `vim`) **get function**

```
      name=Name, company=Company},
    put_visitor(Record).

get_visitors(Date) ->
    open_visitors_db(),
    List = dets:lookup(visitors, Date),
    close_visitors_db(),
    List1 = lists:sort(List),
    List1.
```

And now let's top it off with utility functions:

**Listing 3.37** ($\Rightarrow$ `vim`) **Utility functions**

```
    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 @doc comments to create beautiful documentation. To find out how, check out:

> http://www.erlang.org/doc/apps/edoc/chapter.html

Also note that the function format_name/1 employs pattern matching to deal with three possible user inputs:

- name only

- company only

- name and comany

---

**Return of the Function**

By now, you should have noticed that there is no return keyword as one would expect in a procedural or object oriented programming language. Erlang's functional nature means that the final term evaluated in a function is that function's return value. So that means there is no need for a return keyword.

---

### 3.12.3. Visitors admin

We now need a form to enter visitors into our visitors database. Save visitors_db.erl and create a file called visitors_admin.erl:

**Listing 3.38** (⇒ $) **Visitors admin**

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

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

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

The next few functions may look familiar from the earlier work you've done on `index.erl`:

**Listing 3.39** ($\Rightarrow$ vim) **Nitrogen page functions**

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

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

title() -> "Visitor Admin".

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

Note that body/0 calls inner_body/0:

**Listing 3.40** ($\Rightarrow$ vim) **inner_body/0**

```
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}
    ]}),
```

Forms are easy to create with Nitrogen. Note that we've added our first validator and wired it with `wf:defer/3` rather than `wf:wire/3`. In the case of a page refresh or redraw, the `wf:defer` function assures that form is redrawn before the validators are attached.

Another validator:

**Listing 3.41** (⇒ vim) **Validate date entry**

```
                    unless_has_value=company}
    ]}),
    wf:defer(save, date1, #validate{validators=[
        #is_required{text="Date is required"}
    ]}),
```

And next, a list of Nitrogen elements:

**Listing 3.42** (⇒ vim) **Nitrogen elements**

```
        #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"}
    ].
```

A time dropdown function:

### Listing 3.43   (⇒ vim) **Time dropdown**

```
 #button{id=save, postback=save, text="Save"
   ].

time_dropdown() ->
    Hours = lists:seq(8,17), %% 8am to 5pm
    #dropdown {id=time, options=
        [time_option({H,0,0}) || H <- Hours]}.
```

Helper functions for time_dropdown/0. Note pattern matching:

### Listing 3.44   (⇒ vim) **Time dropdown helper functions**

```
    #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] = re:split(Date, "/", [{return,list}]),
    {wf:to_integer(Y),
    wf:to_integer(M),
    wf:to_integer(D)}.
```

Let's not forget that we need event functions:

**Listing 3.45** ($\Rightarrow$ `vim`) **Vistors admin** `event` **function**

```
    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()).
```

And, of course, we need to save visitor data:

---

**Listing 3.46** ($\Rightarrow$ `vim`) **Save function**

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

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

---

**`wf:defer?`**

It's worth mentioning that `wf:defer/N` is considered a sibling to `wf:wire/N`, the other sibling being `wf:eager/N`.

Both functions wire actions to the browser. The difference here is that actions wired with `wf:defer` will execute after actions wired with `wf:wire`. Because our code is destroying and redrawing the form after every save (with the `wf:update` call), we need to rewire the validators.

---

Note also the function `time_dropdown/0`. Look closely at the list comprehension. What's that about? We leave that to your brilliance.[12]

The function `time_option/1` also demonstrates pattern matching on function parameters. And more, it introduces a new Erlang concept— guard sequences.

---

[12]Hint: http://www.erlang.org/doc/programming_examples/list_comprehensions.html

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. You'll learn much.

---

Note: We'll show you how to create a custom element for picking time in Chapter **??**. Stick with us. Should be fun.

---

There's yet another thing to observe in `visitors_admin.erl`—event/1. Here's our very first Nitrogen event handler in the wild. You'd be correct to surmise that `event/1` is part of the process that validates data entry and posts data back to the server.

Note first that `event/1` is pattern matching on two Erlang atoms—save and `confirm_ok`. How do we know? The first `event/1` function is terminated with a semicolon.

Here's more on pattern matching:

> `http://erlang.org/doc/reference_manual/patterns.html`

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. More here:

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

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

Give yourself a gold star if you said event(confirm_ok).

Extra credit: What is event(confirm_ok) doing for us? Check out these links to learn more:

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

We have one more crucial task. We need to display visitors-of-the-day on our welcome page. Save open `index.erl` and make the following changes to `body/0`:

---

**Listing 3.47** ( ⇒ vim) **Revise `index.erl`**

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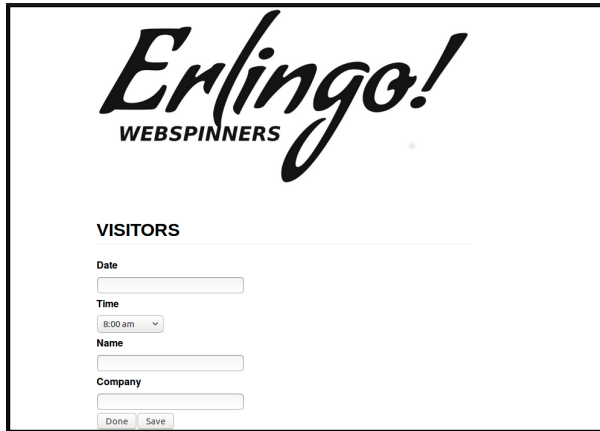~/nb/site/src$ vim index.erl

body() ->
   Visitors = visitors_db:get_visitors(date()),
   [
      #h1{ text="WELCOME!" },
      #h2 { text="Joe Strongman" },
      #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. Save and open the browser to:

**localhost:8000/visitors/admin**

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

`localhost:8000`



## 3.13. Styling

Welcome screen is ugly, you say? Couldn't agree more. Let's tweak CSS to see what we can do to:

---

**Listing 3.48** (⇒ $) **Opening** `site/static/css/style.css`

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

---

Now, change the following in the h1 declaration:

---

**Listing 3.49** (⇒ vim) `style.css`

```
h1 {
    ...
    font-size: 1.875em 3em;
    line-height: 1.066667em;
    margin-top: 1.6em 0.2em;
    margin-bottom: 1.6em 0.6em;
```

---

And add the following new declarations:

---

**Listing 3.50** (⇒ vim) `style.css`

```
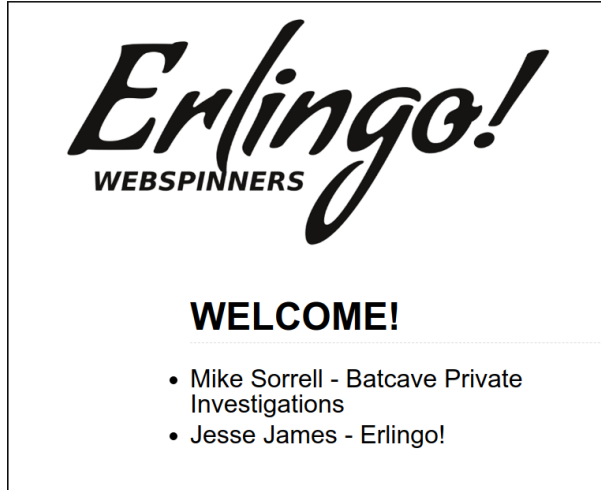    margin-bottom: 0.6em;
}

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

---

Save and refresh your browser:

**http://localhost:8000**



Not too shabby, eh?

## 3.14. Debugging

If the form fails to show up in the admin page, the problem is most likely in visitors_admin.erl. If the entry to the appointment form fails to show up, prepare for a stint of debugging: Check code in visitors_db.erl and visitors_admin.erl for typos. Also test exported functions in those modules by running them in the Erlang shell. Here's how:

---

**Listing 3.51   ( ⇒ >) Test exported functions in the shell**

```
28> l(visitors_db).
```

---

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

---

**Listing  3.52  (⇒ >) What's in our database?**

```
29> 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.

Now would be a good time to commit your hard work to Git:

---

**Listing  3.53  (⇒ $) Commiting to Git**

```
.../css$ ~/nb
.../nb$ git add .
.../nb$ git commit -m "First commit of nitroBoard I"
```

---

## 3.15.  What You've Learned

So, compadre, you've employed and seen in action 18 Nitrogen elements. You've created Nitrogen forms, used Nitrogen events, and wired Nitrogen validators. You've created an Erlang Dets database and learned a smattering of Erlang along the way.

Good day's work!

## 3.16.  Think and Do

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

# 4. Continue Your Adventure...

Thank you for reading this far in *Build It With Nitrogen*. If you enjoyed what you've read so far, continue your adventure by dropping a coin in the slot. That is, you can find *Build It With Nitrogen: The Fast-Off-The-Block Erlang Web Framework* for sale at:

## `BuildItWithNitrogen.com`

## 4.1. What You'll Learn in the Full Book

- How to build basic CRUD interfaces

- How to build highly dynamic interfaces

- How to work with the templates and the DOM

- How to work with logins and password hashing

- How to use OTP basics with gen_servers and supervisors

- How to use the databases: DETS, Mnesia, and PostgreSQL

- How to interact with external APIs (in our case, retrieving stock quotes)

- How to build your own behaviours and use those to produce Nitrogen pages more rapidly

- How to build custom Nitrogen elements and convert those into plugin libraries

*4. Continue Your Adventure...*

- The basics of web application security

- The basics of using other front-end frameworks (specifically Bootstrap)

- How to avoid potential maintenance traps

- The basics of Git

- Learn Erlang along the way

Once again, in case you've already forgotten where you can buy the book, the official website for *Build It With Nitrogen* can be found at:

# BuildItWithNitrogen.com