Chicago Boss: A Rough Introduction

By Evan Miller

Version 0.6.9

Table of Contents

I. Introduction

If you want to keep a secret, tell it to a Swede. Born in Stockholm over 20 years ago, Erlang is the most advanced open-source server platform in existence, but it seems almost no one knows about it. Erlang can handle hundreds of thousands of simultaneous connections; it can spawn millions of simultaneous processes in under a second; server code can be upgraded, in production, without any interruption of service; and errors are handled in such a way that server crashes are extremely uncommon.

What is not to like? Why isn't the entire world programming in Erlang? Well, Erlang is a functional language, which means that to implement any algorithms, you must use recursion instead of familiar "for" and "while" loops. Unlike every major scripting language, there is no built-in syntax for dictionaries or hash maps. And to actually write a functioning server, you must learn an additional layer of magic known as OTP. These barriers, in my opinion, have prevented Erlang from gaining much traction outside of Scandinavia.

But Chicago Boss changes all that. It makes Erlang accessible to hackers who just want to write a reliable website in a nifty language. Boss uses code generation to get around the historic hash-map quandary, and takes care of all the OTP business so that you can focus on writing the features you need for your website. As for the supposed burdens of functional programming, I find that recursion is rarely necessary in workaday web programming; I would guess that 99% of server application code simply shuttles data to and from a database, so in the course of building a website, the pedestrian procedural programmer will hardly miss his "do/while" loops.

If you are an experienced web programmer, you'll probably enjoy all the amenities that CB has to offer: an advanced ORM with support for database associations, sharding, and caching; lightning-fast templates compiled down to Erlang bytecode; automatic recompiling and in-browser error reporting; simple directives for reloads and redirects; routes for constructing URLs and handling requests; full frameworks for sending and receiving email; a built-in message queue; a framework for writing and running functional tests; and a first-of-its-kind event system for monitoring the data model.

In the end, by combining the Erlang platform with its own innovations, Chicago Boss makes websites a delight to develop and a joy to deploy. Boss applications can be written in the same time or less as equivalent Rails applications, and they will almost never crash or leak memory. Since the underlying networking is all asynchronous, you can easily write concurrent services, such as chat, that previously were only possible in callback-based frameworks (such as Nginx, Node.js, Twisted, or Perlbal).

The importance of this advancement cannot be overstated. It is now feasible for a very small team to develop and operate a database-driven, highly interactive, heavily trafficked website with very little capital outlay. Although Chicago Boss can't tell you how to acquire users, the rest of this manual will show you everything you need to do to handle their requests and (with luck) fulfill their desires.

II. Architecture

Chicago Boss is really just a compiler chain and a run-time library for Erlang web applications. Since compilers are scary, we'll focus on the run-time library so that you understand the basics of how requests are fulfilled.

A single Chicago Boss server hosts one or more CB applications. A CB application is a set of controllers, views, models, and URL routes assigned to a base URL in the CB server configuration. You might deploy a blog application to `/blog`, a wiki application to `/wiki`, and a main website application to `/`. Applications can redirect to one another and access each other's data models and message queues. This application transparency is useful for creating reusable components that interact with one another, but where security is a consideration, you should probably run untrusted applications in standalone servers.

A single CB applications might have several parts:
* Web controllers, which are given information about an HTTP request and decide what to do with it
* Web views, which render can data returned by a controller
* Models, which provide an abstraction layer over the database (if any)
* Email controllers and views
* Initialization scripts
* Test scripts
* A route configuration file

But wait... there's more! The CB server starts a number of services which are available to all applications residing in that server:
* A URL router (BossRouter)
* A session storage layer (BossSession)
* A database connection and caching layer (BossDB and BossCache)
* A message queue (BossMQ)
* A model event system (BossNews)
* An email server (BossMail)

For the serious website developer, a "server" might actually consist of multiple Erlang nodes linked together. In this case all of the nodes in the cluster defer to a "master node" (specified in the configuration) for

shared services, such as the database cache, the message queue, and incoming email. The rest of the nodes can then handle regular HTTP traffic or perform other tasks. In these configurations you will want a proxy server, such as Nginx, to distribute incoming requests across the cluster. Since the Erlang VM is multi-threaded, an Erlang node should correspond to a physical machine. Running more than one node on a machine, while possible, is a waste of resources.

Incoming HTTP requests are first parsed by Mochiweb or Misultin (depending on your server configuration). If parsing succeeds, a SimpleBridge request object is handed to the CB server and Boss gets to work.

First, BossRouter parses the requested URL and figures out which application should process the request. The URL is then mapped to a *controller* and an *action.* A controller consists of a set of actions, which might have names like "create", "edit", and "delete"; each action corresponds to a function with the same name in the controller module.

But before any action is invoked, Boss checks to see if authorization is required by executing a special authorization function in the requested controller. The authorization function has full access to the database and session information, and can perform a redirect if authorization fails (for example, to a log-in page).

If authorization succeeds, the action is invoked; it is passed the HTTP method (such as GET or POST), a list of tokens in the URL, and (optionally) the return value from the authorization function. Because controllers are parameterized modules, controller actions also have the SimpleBridge request object and the current session ID available in scope as module parameters.

The controller action might do any of a number of things to fulfill a request, such as pull information from the database, wait on a message from the message queue, or send an email. The return value from the action determines how the CB server will respond to the client. In the normal case, a data structure for populating an HTML template will be returned, but the return value can also direct the CB server to generate JSON, redirect the request to another URL, return an error, or perform other actions.

After the controller action has returned, the next stage of processing is to generate a response. Normally, the response will be generated from an ErlyDTL template with the same name as the controller action. (ErlyDTL is a very fast, pure-Erlang implementation of the Django Template language.) Because templates have access to the data model, additional database fetches might be performed in the HTML generation process. As a result, there is no need for the controller to pre-fetch all of a record's associations. For example, the controller might simply retrieve a "blog post" record, and the template can fetch for itself additional information about the blog post's author's employer's second cousin's daughter's cat, or whatever.

But the generated response is not returned to the client immediately; controllers can define a special post-processing function to transform (or cache, or log) the output before sending it on to the client. This post-processing phase might be used to add a timestamp or perform compression on the output.

After post-processing, the client finally receives the response, and with any luck will make another request.

The workflow for handling incoming email is similar, but less involved. There is only one email controller per application, and return values are ignored. There is also no routing file for email; the recipient name is always mapped to the name of a controller function, so sending mail to "post@domain.com" will invoke the "post" function. The controller function is passed the sender's email address and a parsed version of the email body, and can do with them as it pleases (probably stick a version of it into the database). If security is an issue, an authorization function can authorize the sender's email address and IP address before receiving the email and invoking the handler function.

But enough about architecture; let's get on to the building.

III. Getting Started

To get going with Chicago Boss web development, you will need a machine that has Erlang installed, version R13A or later. The machine should also have a terminal, a web browser, and an editor that makes you feel warm and mushy.

1. Hello, world

Download the Chicago Boss 0.6.9 source code from here:

http://www.chicagoboss.org/

The first task is to open the archive, compile the code, and create a new project.

```
tar xzf ChicagoBoss-0.6.9.tar.gz
cd ChicagoBoss-0.6.9
make
make app PROJECT=cb_tutorial
```

`cb_tutorial` will be the name of the new project. You can name it something else, but the project name must start with a lowercase letter, and contain only lowercase letters, digits, and underscores.

The second task is to enter the new project directory.

```
cd ../cb_tutorial
```

Go ahead, take a look around (I'll wait). Some items of interest that you'll see:

> `boss.config` - The configuration file, where you can do useful things like connect to a real database, set up caching, or define the port to listen on. For this tutorial we'll be using a fake in-memory database, listening on port 8001, and not caching a darn thing.

> `cb_tutorial.app.src` - This is processed to produce a `.app` file, which helps the OTP fairies manage dependencies and upgrade your application in production. Edit the file to give your project an accurate description and a version number other than `0.0.1`.

> `ebin/` - The home of compiled modules as well as the `.app` file. It is not referenced in development, but before starting the production server you

need to populate it with `make`. You can always clear out this directory with `make clean`.

`start-dev.sh` - Run this to start a development server, which automatically recompiles code for you, pretty-prints error messages in the browser, and includes an interactive console. You can terminate the development server by typing `q()`.

`start.sh` - Run this to start a production server, which runs as a background process. You can terminate the production server by sending `kill` to processes named `epmd` and `beam.smp`. (Erlang has poor support for UNIX signal handling, but there's nothing a `-9` can't take care of.)

`priv/` - A directory of junk that doesn't belong anywhere else. The name "priv" makes it sound somehow elite or exclusive, but it's just init scripts, language files, routing tables, static images, CSS, and related bric-a-brac.

`src/` - The future home of your source code. Peek inside! You'll see:

> `controller/` - controllers. It's empty now, but it's the first place we'll go when we want to start writing the application.
>
> `model/` - models for the ORM. All files in here are compiled with a special model compiler, so don't put any library modules in here.
>
> `lib/` - where the library modules go. In development these are potentially recompiled with every request, so if you're using a large 3rd party library it's best to keep it outside the project directory and include the path in your server start scripts.
>
> `view/` - templates. You'll need to create a subdirectory for each controller. Don't miss the special `view/lib/` directory, where you can put reusable template components.

`log/` - Error logs galore. CB will maintain a symlink (`log/boss_error-LATEST.log`) that always points to the most recent log file.

`include/` - Header files. Often they provide record definitions for library modules found in `src/lib/`, but you might put application-wide macros and definitions here.

All right, enough looking around, let's start the development server.

```
./start-dev.sh
```

You will see many messages fly by the screen, such as this one:

```
=PROGRESS REPORT==== 27-Nov-2011::18:54:17 ===
          supervisor: {local,sasl_safe_sup}
             started: [{pid,<0.40.0>},
                       {name,alarm_handler},
                       {mfargs,{alarm_handler,start_link,[]}},
                       {restart_type,permanent},
                       {shutdown,2000},
                       {child_type,worker}]
```

To this day I have no idea what that means, but I am assured by various credible sources that it is harmless. Anyway, below all the messages, you will see an Erlang shell prompt that looks like this:

```
(wildbill@blackstone)1>
```

This is the server's shell. `wildbill` is just a silly default name that I chose for the development node; if you prefer, you can change it in `start-dev.sh` to something less masculine. `blackstone` is the name of my computer; on your computer you will see something else after the `@` sign.

You can type Erlang statements directly into the server shell; here you have complete access to the server state and the loaded code libraries. Just remember to end any statements with a period. You can always halt the server with the `q().` command.

We are now ready to write the obligatory "Hello, world!" application. In the project directory, open a file (in another terminal, or with a desktop text editor) called `src/controller/cb_tutorial_greeting_controller.erl`. This file will be the `greeting` controller of the new `cb_tutorial` application. (The names of controller modules always take the form `<application name>_<controller name>_controller`.)

To get started, type (or paste) this into it:

```
-module(cb_tutorial_greeting_controller, [Req]).
-compile(export_all).
```

```
hello('GET', []) ->
    {output, "Hello, world!"}.
```

Now point your web browser to http://localhost:8001/greeting/hello. You should see a familiar message.

We've written a short controller with one action, called "hello". You can create other actions in the same controller. Each action will have its own URL of the form /<controller name>/<action name>. If the URL contains additional slash-separated tokens beyond the action name, these will be passed as a list to the controller action in the second argument. The first argument, you might have guessed, is an atom specifying the HTTP method that initiated the request: 'GET', 'POST', 'PUT', or 'DELETE'.

This controller module is a *parameterized module*, as indicated by the parameter list ([Req]) in the -module directive. Although Erlang is a functional language, parameterized modules add a little OO sauce so that we don't have to pass the same values all the time to all the functions of a particular module. In the case of CB controllers, every function will have access to the Req variable, which has a lot of useful information about the current request. We'll learn more about parameterized modules later.

Controller actions can return several values. The simplest is {output, Value}, and it returns raw HTML. We can also use {json, Values} to return JSON:

```
hello('GET', []) ->
    {json, [{greeting, "Hello, world!"}]}.
```

I've changed the string to a *proplist*, which is just a list of key-value tuples. Proplists are ubiquitous in Erlang programming, so I hope you like them. The json return value tells CB to transform the given proplist into honest-to-goodness JSON for consumption by the client. Refresh your browser to see the result.

The json return value is convenient for creating JSON APIs with Chicago Boss. Of course, if we wish to generate HTML, it will be convenient to use a template.

To use templates, we will need to change the controller code, and to create an actual template file. First, alter the controller file and replace json with ok:

```
hello('GET', []) ->
    {ok, [{greeting, "Hello, world!"}]}.
```

The variable list will now be passed to the associated template, if there is one. There's not one, so we need to create it. Open a new file `src/view/greeting/hello.html` (you may need to create a `greeting` subdirectory), and type or paste this into it:

```
<b>{{ greeting }}</b>
```

That is a snippet of Django template language, which has become something of a *lingua franca* in Erlang web development. The language is simple, and it is easy for non-Erlang programmers to understand and write. Anyway, refresh the browser to see the greeting in bold.

To see an emphatic greeting, try this:

```
<b>{{ greeting|upper }}</b>
```

Here we've used a *filter*, whose function should become obvious once you refresh the browser. Filters offer a cornucopia of formatting functionality, and they can be chained together in the same way that UNIX commands are chained: simply insert a pipe character ("|") between each filter name. Some other filters to try on your own time are `length`, `slugify`, `title`, and `wordcount`.

To summarize, we have seen that controllers can return three types of tuples:

- `{output, Value}` - Send `Value` as raw HTML
- `{json, Values}` - Format `Values` (a proplist) as JSON
- `{ok, Values}` - Pass `Values` (a proplist) to the associated template

Well, that was fun. We have said hello to the world, but what shall we do when the world says hello to us? Most web applications will need to store or retrieve data. Having touched on the View and Controller, we now visit the Model.

2. Hello, database

For dealing with databases, Chicago Boss includes a special querying syntax, a full ORM, and a handy set of database adapters. To get started, we'll create a simple model file and see what we can do with it. Open a new file called `src/model/greeting.erl`, and type or paste this into it:

```
-module(greeting, [Id, GreetingText]).
-compile(export_all).
```

This is the model for "greeting" objects. It is really a parameterized module with two parameters (`Id` and `GreetingText`). All models must have `Id` as their first parameter, but subsequent parameters can be whatever you like.

This looks like an ordinary Erlang module, but it is harboring a number of secret superpowers. It is time for a little detour into the hidden wonders of the Chicago Boss ORM, called BossRecord.

Refresh the browser (so that this model file is properly compiled and loaded), then try this in the server shell:

```
> Greeting = greeting:new(id, "Hello, world!").
{greeting,id,"Hello, world!"}
```

The `new` function just returns a new instance of `greeting`. I passed in `id` as the first argument because that tells BossDB to generate a new ID number when the time comes; if I wanted a particular ID, such one that included my basketball jersey number, I could have specified it.

As you can see above, the `greeting` instance is really just a tuple that includes the module name followed by all of the passed-in parameters. When you call a function of a parameterized module (other than `new`), the run-time system just binds values from that tuple to the module's parameter list before executing the function. Even though it sort-of looks like object-oriented programming, passing around the `greeting` instance really is just passing around all of its parameter values. Like other Erlang variables, the values are immutable, so we don't need to worry about locks, side effects, or other perils of object orientation.

It's time to get to know the model. Try this:

```
> Greeting:greeting_text().
"Hello, world!"
```

"But wait just a gosh-darn minute," you might be thinking, "I don't remember putting a `greeting_text/0` function in *my* `greeting` module! In fact, I don't remember putting *any* functions in there. Am I getting Alzheimer's? Where am I?"

Don't worry, you are not losing your mind (and you are sitting at your computer). Before loading model modules into the system, the Chicago Boss

compiler surreptitiously attaches extra functions to them. Here are a couple of others to try. I'll let you figure out what they do.

```
> Greeting:attributes().
[{id,id},{greeting_text,"Hello, world!"}]

> Greeting:attribute_names().
[id,greeting_text]
```

This one is useful too, but it might not behave exactly as you expect:

```
> Greeting:set(greeting_text, "Good-bye, world!").
{greeting,id,"Good-bye, world!"}
```

Erlang variables are immutable, so calling `set/2` returns a new record without altering the old one in any way. Try this to see what I mean:

```
> Greeting.
{greeting,id,"Hello, world!"}
```

See? The `Greeting` variable remains alive and full of hope.

These generated functions come in mighty handy in day-to-day Chicago Boss programming. Without them, you would need to call `proplists:get_value/2`, `proplists:delete/2`, and similar verbosities. But the most important function of them all, the function without which all the other functions would be useless and nothing except error messages would ever persist to disk, is this:

```
> Greeting:save().
{ok,{greeting,"greeting-1","Hello, world!"}}
```

Try it. Congratulations! You just saved your first record to the database, and the `id` atom was replaced with a real-live identification string. Note that generated IDs in Chicago Boss take the form `"model-number"` (here it is `"greeting-1"`). By including the model name in the ID, Chicago Boss can guarantee that the IDs are unique across all of the different models. (This simple fact greatly simplifies the API design.)

You might be wondering how you are going to remember all of these useful functions that are clandestinely attached to each BossRecord. Can you keep a secret? This is a secret you shouldn't even tell your spouse (particularly if he or she is a Rails programmer). Open up a browser and visit http://localhost:8001/doc/greeting. Quick, close it before anyone sees. You just caught a glimpse of

the Chicago Boss's automatic EDoc, which will tell you about all the functions generated for each model. In the Chicago Boss community we call this feature "/doc", for reasons you are left to ponder in private.

Anyway, back to the shell. I've been avoiding mentioning this, but we have a slight problem. After saving the greeting, the `Greeting` variable is still bound to the unsaved version:

```
> Greeting.
{greeting,id,"Hello, world!"}
```

Oh dear. It would seem that the saved record is "lost" because we didn't bind anything to the return value of `save/0`. What is a budding CB programmer to do?

Not to worry -- it's time to turn to BossRecord's best friend since childhood, the API with an attitude, Mr. BossDB.

BossDB is a library for querying the database. It is unusual in that it will accept a language-integrated syntax that is illegal outside of Chicago Boss projects (in fact, it is illegal in the Chicago Boss server shell). For now we will use the more verbose syntax that will not cause us any trouble with the Erlang interpreter.

To find the record that we saved, it is best to cast a wide net. We use `boss_db:find/2`.

```
> boss_db:find(greeting, []).
[{greeting,"greeting-1","Hello, world!"}]
```

Whew! The saved record hasn't gone anywhere. We can retrieve it anytime.

`boss_db:find/2` is the simplest way to query the database; it takes a model name and a list of conditions and returns a list of results. We didn't want to impose any search conditions, so the list here is empty.

But try a few search conditions, just for fun:

```
> boss_db:find(greeting, [{id, 'equals', "greeting-1"}]).
> boss_db:find(greeting, [{greeting_text, 'matches', "^Hell"}]).
> boss_db:find(greeting, [{greeting_text, 'gt', "Hellespont"}]).
> boss_db:find(greeting, [{greeting_text, 'lt', "Hellespont"}]).
```

See if you can figure out what each does. There are many more query operators available besides `equals`, `matches`, `gt`, and `lt`, some 18 in total. You will want to spend some time with the API documents to learn more about them.

Of course, if we know the record ID, we don't need to mess with search conditions and query operators at all. We can use the very handy `boss_db:find/1`, like this:

```
> boss_db:find("greeting-1").
{greeting,"greeting-1","Hello, world!"}
```

The function returns either a record with the provided ID, or the atom `undefined`. Note that we didn't have to formally specify the name of the model, since it is embedded in the ID string.

Before we forget that we're working on a web application, let's get back to the source code and figure out how to take advantage of BossRecord and BossDB on our website.

3. A database-driven website

Open up a new template `src/view/greeting/list.html`. We'll use it to display all the greetings in the database. (By the way, if you use Vim as your text editor, use `:setf htmldjango` to get proper syntax highlighting.)

```html
<html>
<head>
  <title>{% block title %}Greetings!{% endblock %}</title>
</head>
<body>
{% block body %}
<ul>
{% if greetings %}
  {% for greeting in greetings %}
    <li>{{ greeting.greeting_text }}
  {% endfor %}
{% else %}
    <li>No greetings!
{% endif %}
</ul>
{% endblock %}
</body></html>
```

In the code above, we've used a number of Django template tags that may be unfamiliar to you, but they should be easy to figure out. The `{% if %}` tag tests to see if a variable is defined and not `false`. The `{% for %}` tag iterates over a list, and the `{% block %}` tag is used in template inheritance; we'll explore template inheritance shortly. Anyway, with this template in place, point your browser to http://localhost:8001/greeting/list to see the greeting you created in the console formatted in beautiful HTML.

Gotcha! There's nothing there. We need to create a controller action to retrieve the greetings from the database. Open the controller (`src/controller/cb_tutorial_greeting_controller.erl`) and add a `list` action:

```
list('GET', []) ->
    Greetings = boss_db:find(greeting, []),
    {ok, [{greetings, Greetings}]}.
```

By now this controller code should be comprehensible; the first line queries the database for all greetings, and the second sends them all to the template. Refresh the browser, and you'll see your HTML-formatted greeting at long last.

Next we will create a form for submitting new greetings. Add this to the bottom of the `body` block of `src/view/greeting/list.html`:

```
<p><a href="{% url action="create" %}">New greeting...</a></p>
```

Here we've used a new tag, the `url` tag. It takes a key="value" argument list for constructing a URL, and must always have an `action` key.

Then create a new template, `src/view/greeting/create.html`:

```
{% extends "greeting/list.html" %}
{% block title %}A new greeting!{% endblock %}

{% block body %}
<form method="post">
Enter a new greeting:
<textarea name="greeting_text"></textarea>
<input type="submit">
</form>
{% endblock %}
```

Using the `{% extends %}` tag, this template inherits from the `list.html` template we created earlier. With it, the `create.html` template inserts its own content into the `title` and `body` blocks defined in `list.html`; in this way, we don't need to redefine the HTML `<head>`, `<title>`, and `<body>` tags in every template. As you might imagine, template inheritance is an essential strategy when creating and maintaining anything but a one-page website.

Anyway, refresh the browser, and click the link to make sure that the form we created is there. Even though we won't be winning any design awards, we need to process the form, so add a `create` action to the controller:

```
create('GET', []) ->
    ok;
create('POST', []) ->
    GreetingText = Req:post_param("greeting_text"),
    NewGreeting = greeting:new(id, GreetingText),
    {ok, SavedGreeting} = NewGreeting:save(),
    {redirect, [{action, "list"}]}.
```

The first clause handles GET requests, and simply renders the template without an variables.

The second clause handles POST requests. It first pulls the "greeting_text" parameter from the form using `Req:post_param/1`. Recall that `Req` is a parameter of the controller module (in fact, the only parameter). The action then creates a new greeting with the greeting text and saves it to the database. Finally, the function returns a `{redirect, Location}` directive, which we have not encountered before. This directive performs an HTTP 302 redirect to the specified location, which can be either a URL string or a proplist with an `action` key (just like the `{% url %}` tag in the template). In this example, we redirect to the `list` action.

We now have a database-driven website. Submit a greeting or two, and marvel at the power of the Internet.

It is likely that our application will occasionally display a greeting that does not have any merit, so next we will add a delete button to our application.

Add the following form to the bottom of the `body` block of the `list` template (`src/view/greeting/list.html`):

```
<form method="post" action="{% url action="goodbye" %}">
Delete:
```

```
<select name="greeting_id">
{% for greeting in greetings %}
<option value="{{ greeting.id }}">{{ greeting.greeting_text }}
{% endfor %}
</select>
<input type="submit">
</form>
```

There should not be any surprises in this code snippet. Next, we need to process the form so that it works. Add the following code to the bottom of the controller (`src/controller/cb_tutorial_greeting_controller.erl`):

```
goodbye('POST', []) ->
    boss_db:delete(Req:post_param("greeting_id")),
    {redirect, [{action, "list"}]}.
```

The function `boss_db:delete/1` takes a record ID and deletes the associated record from the database, just like that. Use it with caution!

With that we have a delete feature. Now you have a form for getting rid of some of the less interesting greetings that you created before.

Of course, we are far from finished. Currently, anyone can store any greeting, no matter how long or how short. Next we will add validation code to the `greeting` model, and return an appropriate error message to the user if validation fails.

4. Simple validation

To ensure that greetings are non-empty and tweetable, add the following code to the `greeting` model (`src/model/greeting.erl`):

```
validation_tests() ->
    [{fun() -> length(GreetingText) > 0 end,
        "Greeting must be non-empty!"},
     {fun() -> length(GreetingText) =< 140 end,
        "Greeting must be tweetable"}].
```

`validation_tests/0` is an optional function that should return a list of `{TestFun, FailureMessage}` tuples. If any of the functions returns `false`, the pending save operation is aborted, and the call to `save/0` returns `{error, FailureMessageList}`.

Try submitting an empty greeting. You should see something like this:

Error:

```
{{{{cb_tutorial_greeting_controller,['Req']},create,2},
  {line,{17,25}},
  {match,{error,["Greeting must be non-empty!"]}}}},
 [{cb_tutorial_greeting_controller,create,3},
  {boss_web_controller,execute_action,5},
  {boss_web_controller,process_request,5},
  {timer,tc,3},
  {boss_web_controller,handle_request,3},
  {mochiweb_http,headers,5},
  {proc_lib,init_p_do_apply,3}]}
```

Following the error message, look at line 17 of the controller (`src/controller/cb_tutorial_greeting_controller.erl`). That line is:

```
{ok, SavedGreeting} = NewGreeting:save(),
```

As indicated by the error message, we are getting a `match` error. The function is returning `{error,["Greeting must be non-empty!"]}`, but we are trying to match it to `{ok, SavedGreeting}`, so the process crashes (in the harmless, Erlang sense of the word "crash," of course).

Next, instead of crashing, we'll modify this part of the controller to match potential validation errors, and present them to the user in the template. Modify the controller accordingly:

```
create('POST', []) ->
    GreetingText = Req:post_param("greeting_text"),
    NewGreeting = greeting:new(id, GreetingText),
    case NewGreeting:save() of
        {ok, SavedGreeting} ->
            {redirect, [{action, "list"}]};
        {error, ErrorList} ->
            {ok, [{errors, ErrorList}, {new_msg, NewGreeting}]}
    end.
```

If saving succeeds, we perform a redirect as before. But if validation fails, we send a list of errors to the template along with the greeting that failed to save.

Now modify the top of the view (`src/view/greeting/create.html`) to show any errors in red, and to display the greeting that was previously entered so that the user does not need to retype anything:

```
{% if errors %}
<ul>
    {% for error in errors %}
    <li><font color=red>{{ error }}</font>
    {% endfor %}
</ul>
{% endif %}

<form method="post">
Enter a new greeting:
<textarea name="greeting_text">{% if new_msg %}
{{ new_msg.greeting_text }}{% endif %}</textarea>
<input type="submit">
</form>

...
```

(Note that the code inside the `textarea` tag should be on one line if you don't want the template to insert any spurious whitespace into the form.) Now try entering in an exceedingly long greeting into the form. It can't be saved!

5. Save hooks

Save hooks can execute a callback before or after a record is created, updated, or deleted. But save hooks can also be used to modify a record before it is actually saved, or to abort the pending save operation after it has passed validation. Just to get a feel for things, we'll write a hook that replaces an obscene word with a clean word in our application before it is saved to the database.

Open `src/model/greeting.erl`, and add this function:

```
before_create() ->
    ModifiedRecord = set(greeting_text,
                        re:replace(GreetingText,
                                "masticate", "chew",
                                [{return, list}])),
    {ok, ModifiedRecord}.
```

We are using the (generated) `set/2` function of this module to replace the value of the `greeting_text` attribute with something else. In particular, we are using the `re` module to replace instances of a certain Latinate word that appear in `GreetingText` with its Anglo equivalent. (For this and other applications, the `re` module is a good one to know.)

Here we returned `{ok, ModifiedRecord}`. We also could have returned `ok` if we wanted to continue with the save as originally planned. Alternatively, if the greeting contained something so dirty that no amount of expurgating could possibly redeem it, we could have aborted the save operation by returning `{error, "Do you kiss your mother with that mouth?"}`, or something to that effect. Note that save hooks execute *after* model validation, so it is possible to slip something shady past the validation police using a save hook.

Go ahead and try submitting a greeting through your form with the forbidden word. You will see that it is properly sanitized.

There are other save hooks available, and the curious reader is encouraged to explore the API documentation on them. But the impatient reader will want to get to the fun stuff described in the next section.

***

By now we've covered the basics of programming a database-driven website with Chicago Boss: submitting a form, creating a record, ensuring it is not crazy before saving it to the database, retrieving records for display, deleting a record if it has no redemptive qualities, and restricting privileges based on IP address. These are fairly mundane functions that ought to be familiar to anyone who has done web server programming in any language. In the next section, we're going to leverage Erlang's strengths to implement features that in other languages would be difficult, if not impossible.

IV. Getting Fancy

So far we have explored features of Chicago Boss that are available in most MVC web frameworks. In this section, we'll explore the features that make Chicago Boss unique: the message queue, and the model event notification system. Together, they make it possible to have a web page that updates itself in real-time while consuming very few server resources.

1. BossMQ

Chicago Boss ships with a message queue service called BossMQ. The service consists of named *channels* which follow a publish/subscribe architecture; any Erlang process can publish or subscribe to any channel, and Erlang term can be sent as a message. Channels need not be explicitly created or destroyed; they are created on demand for publishers or subscribers, and automatically destroyed after a certain (configurable) amount of time. BossMQ runs in clustered configurations just as well as a single-machine setup.

The two most important functions in the BossMQ API are `boss_mq:push/2` (i.e., publish) and `boss_mq:pull/2` (i.e., subscribe). `push` is non-blocking, but `pull` may block until a message is available on the channel. A non-blocking version of `pull` is also available; it is called `poll`.

To get a feel for the BossMQ API, try this in the server shell:

```
> boss_mq:push("test-channel", "Who's there??").
```

That pushes a message onto the channel named "test-channel". The channel names can be any valid string.

You should see a PROGRESS REPORT in the console that looks like this:

```
=PROGRESS REPORT==== 1-Dec-2011::16:12:10 ===
          supervisor: {<0.304.0>,bmq_channel_sup}
             started: [{pid,<0.305.0>},
                       {name,mq_channel_controller},
                       {mfargs,
                           {bmq_channel_controller,start_link,
                               [[{max_age,60},
                                 {supervisor,<0.304.0>},
                                 {channel,"test-channel"}]]}},
                       {restart_type,permanent},
```

```
                    {shutdown,2000},
                    {child_type,worker}]
```

That PROGRESS REPORT just indicates that the "test-channel" channel was automatically created for us.

Below (or perhaps above) the PROGRESS REPORT is the return value:

```
{ok,1322777530216837}
```

The large number there is a timestamp assigned to the message that we pushed. We'll see later that message timestamps are an important part of applications designed to use BossMQ so that no messages are missed and that duplicate messages are not received.

Now we can perform a `pull` to receive the message that we just sent. Try this:

```
> boss_mq:pull("test-channel").
```

You should see a return value like:

```
{ok,1322778236795192,["Who's there??"]}
```

The second element of the return tuple is another timestamp; this timestamp references the time of the pull. The third element is a list of messages currently on the channel. Here, we have only one message -- the one that we pushed a few moments ago.

Pulls are non-destructive, so many clients can receive the same message. Try performing another pull to see what I mean.

Once you are satisfied that messages aren't being deleted upon receipt, try this:

```
> boss_mq:pull("test-channel", now).
```

The second argument is a timestamp; any returned messages will have been sent *after* that timestamp. By passing in `now`, we are indicating that we only want *new* messages, that is, we are not interested in the messages currently sitting on the channel.

We now have a problem. The call to `pull/2` blocks until a new message is available. Since it blocks, we can't type anything into the server shell. In order

to regain control of the console, that is, in order for `pull/2` to return, we need to send a message to the "test-channel" channel from another process.

Open `src/controller/cb_tutorial_greeting_controller.erl`, and add a new action:

```
send_test_message('GET', []) ->
    TestMessage = "Free at last!",
    boss_mq:push("test-channel", TestMessage),
    {output, TestMessage}.
```

Point your browser to http://localhost:8001/greeting/send_test_message. You should see the test message, and instantly regain control of the console.

Messaging is a powerful tool that allows for complex, real-time interaction among many services. In the next few sections, we'll use messaging with save hooks to create a page that updates itself whenever a new greeting is added to the database.

2. About long-polling

The strategy that we will use to update a web page is long-polling, also known as Comet. If you are unfamiliar with long-polling, the basic strategy is this: first, the web browser issues a request to the server asking for new information. But the server doesn't return a response immediately; instead, it holds on to the request until new information actually becomes available, and only sends a response when it has something interesting.

This strategy is in contrast to short-polling, where the client issues a request to the server every few seconds and the server returns an answer immediately every time. There are two disadvantages of short-polling compared to long-polling: it uses more bandwidth and CPU resources (since the server needs to parse more requests), and it introduces a delay in delivering new information.

But historically, short-polling has been preferred over long-polling in web programming because long-polling ties up a process while the long-poll request is pending. In a scripting language such as Ruby, the memory overhead can be several megabytes for every connected client. With more than a handful of people connected to the server, the server would quickly run out of resources.

It is possible to write long-pollers in C or scripting languages in an asynchronous (event-driven) environment, such as Nginx (C), Twisted (Python),

EventMachine (Ruby), or Node.js (JavaScript). However, the difficulty with these frameworks is that a callback function must be provided for every network request that occurs as part of the long-poll. Compared to synchronous code, asynchronous code quickly becomes difficult to understand and maintain.

Erlang solves this dilemma. It is perfectly suited not just for *handling* long-poll requests, but for *writing, understanding, and maintaining* long-poll handlers. Erlang is "the best of both worlds" in this very specific sense: *your* code will appear to be synchronous, but under the hood, all of the networking is event-driven and asynchronous. Erlang combines the best aspects of synchronous and asynchronous programming, and there is no other language like it in the world.

With Chicago Boss, we can write long-poll handlers as we would any other handler, no callbacks or other hoop-jumping required. In fact, Chicago Boss treats long-poll requests *exactly* as it does other requests, so we can put a long-poll endpoint anywhere we like in the application.

As mentioned, BossMQ will sit in the middle of our long-poll design. The basic strategy will be:

- Whenever activity of interest occurs, a message is placed in a known channel on the message queue

- Sometime later, a client sends an HTTP request

- The server asks BossMQ if there are any messages available in the channel of interest

    - If yes, the messages are returned immediately

    - If no, the request blocks until a message is put onto the message queue

        - As soon as new database activity occurs, a message goes into the message queue. The blocking call returns, the controller action logic continues, the server returns a response to the client, and the client updates the page.

It sounds complicated, but we will need only a few pieces of code to achieve it. First, we need code to push the update into the message queue; second, we need code to pull updates off of the message queue; and finally, we need JavaScript code to inject those updates into the web page. We will address these parts in order.

3. Implementing real-time updates with save hooks

First, open up `src/model/greeting.erl` and add the following save hook:

```
after_create() ->
    boss_mq:push("new-greetings", THIS).
```

In a parameterized module (such as the `greeting` model), the variable `THIS` refers to the current module instance; the code above simply pushes a copy of each newly-created greeting to the "new-greetings" channel.

Next, create a new action in the controller (`src/controller/cb_tutorial_greeting_controller.erl`) to pull messages from the "new-greetings" channel and return them as JSON to the client:

```
pull('GET', [LastTimestamp]) ->
    {ok, Timestamp, Greetings} = boss_mq:pull("new-greetings",
        list_to_integer(LastTimestamp)),
    {json, [{timestamp, Timestamp}, {greetings, Greetings}]}.
```

The timestamp is important to the design, because it lets us retrieve older messages in case there is a chronological gap in the client's sequence of long-polls. Note that in line 2 a new timestamp is returned by `boss_mq:pull/2` along with the list of messages; this timestamp will be used as an input to the *next* long-poll executed by the client, so we encode it in the JSON response.

To get things going, we will need an initial list of greetings, as well as an initial timestamp, which can be retrieved from `boss_mq:now/1`. Create a new "live" action in the controller, like this:

```
live('GET', []) ->
    Greetings = boss_db:find(greeting, []),
    Timestamp = boss_mq:now("new-greetings"),
    {ok, [{greetings, Greetings}, {timestamp, Timestamp}]}.
```

This action will populate a template for a live-updating page; JavaScript in that page will subsequently issue requests to the `pull` action above. Note that `boss_mq:now/1` requires the name of the channel because channels might reside on different Erlang nodes whose clocks could be slightly out of sync.

With that, the server code for real-time updates is finished! Now we just need a template with some JavaScript to perform the actual polling and to insert new

greetings into the web page. Create a new file `src/view/greeting/live.html`, and add the following as its contents:

```html
<html><head>
  <title>Fresh hot greetings!</title>
  <script src="http://ajax.googleapis.com/ajax/libs/jquery/
1.7.0/jquery.min.js"></script>
  <script>
    function listen_for_events(timestamp) {
        $.ajax("/greeting/pull/"+timestamp, { success:
          function(data, code, xhr) {
            for (var i=0; i<data.greetings.length; i++) {
              var msg = data.greetings[i].greeting_text;
              $("#greeting_list").append("<li>"+msg);
            }
            listen_for_events(data.timestamp);
        } });
    }
    $(document).ready(function() {
      listen_for_events({{ timestamp }});
    });
  </script>
  </head>
  <body>
    <ul id="greeting_list">
      {% for greeting in greetings %}
        <li>{{ greeting.greeting_text }}
      {% empty %}
        <li>No greetings!
      {% endfor %}
    </ul>
  </body>
</html>
```

This template creates an initial list of greetings with the provided template variables in the "greeting_list" `ul` element. When the document is loaded, it executes `listen_for_events()`, which takes as its only argument the timestamp that will be passed to the `pull` action. An AJAX request is issued, and when it returns (possibly much later), the DOM is updated with the new greetings, and `listen_for_events` is called again with the updated timestamp.

Taking these three pieces together (the `after_create` save hook, the `pull` action, and the `live` template), we have a page that stays synchronized with the database as new greetings are created!

To test the feature, open a window for http://localhost:8001/greeting/create and a separate window for the new page, http://localhost:8001/greeting/live. For maximum effect, put the windows side by side, invite a few coworkers over to your computer, and dim the lights. Create a greeting in the first window, and BAM! It will appear instantaneously in the second window.

The implementation here is a good start, but having notification logic in our data model may seem a bit, well, unseemly. In the next few sections, we'll learn about BossNews, a notification system that will let us act on changes to the data model without contaminating our Platonic model code with the temporal world of message queues.

4. BossNews

Chicago Boss introduces *model events* to database-driven server programming. Model events are something of a foreign subject to most server programmers, whose idea of an event is a file descriptor that is ready for reading or writing. Model events refer to *callbacks that are executed when there is a change to application state*; they can be used to ensure that some action is taken whenever some data of interest changes, *regardless of which code path actually performed the change.*

Model events are similar to, but distinct from, save hooks.

- Save hooks require you to write code to detect what actually changed in an update. With model events, we specify the particular changes we are interested in and receive a structured event that describes the change (for example, `{"account-1", first_name, "Jon", "John"}`).

- Save hooks execute in the caller's thread, and so functions that block in the save hook will block the caller. If we wish to send an email, for instance, the caller must wait while the email is generated and sent. Model events, on the other hand, execute in a separate process and are suitable for potentially long-running tasks.

- Save hooks are a static bit of code embedded in the model object. Model events, on the other hand, can be dynamically added or deleted anywhere in the application, and can be made to expire after a given amount of time. They are "disposable" and thus inexpensive from a code-maintenance point of view. For example, it is easy to use model

events to implement unimportant features on your website without ever touching important model code.

Once you use model events in a server environment, you will feel depressed using a web framework that doesn't have them.

Chicago Boss's model event system is known as BossNews. To get to know the API, try this in the server shell:

```
> {ok, WatchId} = boss_news:watch("greetings", fun(_, _) ->
        error_logger:info_msg("Did you see that??~n") end).
```

Here we've used the `boss_news:watch/2` function to watch a particular *topic string* (here, `"greetings"`). The provided callback will be executed whenever a record enters or leaves the "greetings" set, that is, when a greeting is created or deleted. In Chicago Boss lingo, we say that the function "creates a watch". It returns `{ok, WatchId}`; the `WatchId` is an integer which can be used to cancel the watch later on when we are tired of receiving messages.

Go ahead and try it out by creating a new greeting in the console:

```
> NewGreeting = greeting:new(id, "Test greeting").
{greeting, id, "Test greeting"}.
```

Nothing so far. Now save it.

```
> NewGreeting:save().

=INFO REPORT==== 28-Nov-2011::17:01:47 ===
Did you see that??
{ok,{greeting,"greeting-8","Test greeting"}}
```

The greeting is saved, and our callback is executed. Here we have most of the power of save hooks without even *touching* the code base; we've done everything in the console. As you might guess, watches are highly useful for debugging a system in production.

Go to the web browser and try creating and deleting greetings. You should see similar `INFO REPORT` messages in the show up in the console.

Before these get too annoying, we can cancel the watch thus:

```
> boss_news:cancel_watch(WatchId).
```

If you create or delete greetings now, no `INFO REPORTS` will appear in the console (besides the usual request logging).

The callback function that we provided above took two arguments, but I didn't tell you what those arguments were. Well, you're old enough now. The first argument is the name of the event: `created` or `deleted`. The second argument is simply a copy of the record that was created or deleted.

Try a more sophisticated callback, such as this:

```
> {ok, WatchId2} = boss_news:watch("greetings",
        fun(Event, Record) ->
        error_logger:info_msg("Didn't you hear? ~p was ~p~n",
                                [Record:id(), Event]) end).
```

(Note that we bind the result to something other than `WatchId`, since that variable is already bound.)

Create and delete records to confirm that the callback works, and cancel the watch (with `boss_news:cancel_watch/1`) when the novelty has worn off.

BossNews provides a rich API for receiving these and other events; you are encouraged to explore the API docs to see the full power of BossNews for yourself. For now, we are going to put BossNews to work for us on the website we've been building.

5. Implementing real-time updates with BossNews

We'll now replace the save-hook implementation of our real-time updates with BossNews. To start, comment out the save hook in src/model/greeting.erl:

```
% after_create() ->
%    boss_mq:push("new-greetings", THIS).
```

Now we'll set up a watch directly in the console. Type this in the server shell:

```
> {ok, WatchId3} = boss_news:watch("greetings",
        fun(created, NewGreeting) ->
            boss_mq:push("new-greetings", NewGreeting);
          (deleted, OldGreeting) ->
            boss_mq:push("old-greetings", OldGreeting) end).
```

That will set up a watch that adds newly created and newly deleted greetings to separate channels on the message queue.

Go ahead and make sure it works by visiting http://localhost:8001/greeting/live, and then creating a new greeting in the console. We now have a live-updating system without the use of save hooks.

This is not a very permanent solution, unless we wish to type that command into the console every time the server starts. Instead, we can add it to a server start-up script, `priv/init/cb_tutorial_01_news.erl`. Change the `init/0` function to this:

```
init() ->
    {ok, WatchId} = boss_news:watch("greetings",
       fun(created, NewGreeting) ->
            boss_mq:push("new-greetings", NewGreeting);
          (deleted, OldGreeting) ->
            boss_mq:push("old-greetings", OldGreeting)
       end),
    {ok, [WatchId]}.
```

The `init/0` function simply needs to return `{ok, ListOfWatchIDs}`. This list of created watch IDs is needed so that the `stop/1` function knows which watches to cancel when the server shuts down or reloads the script.

Now in the console, we can cancel the watch that we previously created in the console, and read in the script so that a new watch is created in its place:

```
> boss_news:cancel_watch(WatchId3).
ok
> boss_web:reload_init_scripts().
ok
```

With that, the feature is finished. We've taken out the save hook, and the BossNews observer will be set up on server start-up. Try it out by stopping and restarting the server.

```
> q().
```

```
./start-dev.sh
```

Since we've been using an in-memory database for this tutorial, restarting the server will lose any existing greetings. But if everything went as planned, new

greetings will automatically appear on our live-updating page. For extra credit, see if you can implement a "live delete" where deleted greetings are instantly excised from the browser.

<p style="text-align:center">***</p>

Although few persons besides your spouse and mother will be interested in your database of greetings, the implications of the code we've written in this section are quite exciting. With BossNews and BossMQ at your disposal (and BossDB and BossRecord at your back), you'll be able to write production-ready features that you may previously have thought were the exclusive province of large Internet companies. If you're writing a social network, you can now implement chat and status updates. If you're writing auction or exchange website, you will be able update the current bids in real-time. If you're writing an internal inventory system, you can be sure that no one in the company ever sees an out-of-date view of the system. All in just a few lines of Erlang.

The best part, of course, is that the Erlang/OTP platform is solid as a rock. As long as you do a little capacity planning and follow Erlang best practices, you'll be able to sleep without worrying about the application server crashing (that's what database servers are for, right?). All the features that we have covered here, by the way, will work "out-of-the-box" on one machine or a cluster of a hundred machines.

One last feature worth mentioning is that BossNews is deeply integrated with BossDB's cache system; if you turn on caching, there is no need to write cache-related save hooks or cache-invalidation logic. The cache always stays in sync with the database using model event notifications.

There is much more to learn about Chicago Boss, but it is best to stop here. If you've had fun so far, check out the wiki, read the API docs, sign up for the mailing list, and get involved in our creative and active developer community. It's a great time to be an Erlang web programmer.