Chicago Boss: A Rough Introduction

By Evan Miller

DRAFT 11/27/2011

Table of Contents

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 productive.

1. Hello, world

Download the Chicago Boss 0.6.6 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-0.6.6.tar.gz
cd ChicagoBoss-0.6.6
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 start the application server.

```
cd ../cb_tutorial
./start-dev.sh
```

You will see many messages fly by the screen, and finally an Erlang shell prompt. This is the server's shell. You can type Erlang statements directly into it, if you like, but all of the real work will happen in the text editor and web browser.

You are now ready to write the obligatory "Hello, world!" application. In the project directory, open a file called `src/controller/cb_tutorial_message_controller.erl`; this will be the `message` controller of the new `cb_tutorial` application. (The names of controller modules always take the form `<application name>_<controller name>_controller`.) Type (or paste) this into it:

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

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

Now point your web browser to http://localhost:8001/message/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'.

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, [{message, "Hello, world!"}]}.
```

Refresh your browser to see the result.

This method 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, [{message, "Hello, world!"}]}.
```

Now open a new file `src/view/message/hello.html` (you may need to create the "message" subdirectory), and type or paste this into it:

```
<b>{{ message }}</b>
```

Refresh the browser to see the greeting in bold.

To summarize, we have seen that controllers can return three types of tuples:
- `{output, Value}` - Send Value (an I/O list) as raw HTML
- `{json, Values}` - Format Values (a proplist) as JSON
- `{ok, Values}` - Pass Values (a proplist) to the associated template

Chicago Boss templates use the Django template language, which was popularized by the Python web framework of the same name. The language is simple, and it is easy for non-Erlang programmers to understand and write.

Most web applications will need to store or retrieve data at some point; having touched on the View and Controller, we now visit the Model.

2. Hello, database

Open a new file called `src/model/message.erl`, and type or paste this into it:

```
-module(message, [Id, MessageContents]).
-compile(export_all).
```

This is the model for "message" objects. It is really a parameterized module with additional methods attached to it by the Chicago Boss compiler. All models must have `Id` as their first parameter, but subsequent parameters can be whatever you like.

We will now store messages in the database and retrieve them.

First, alter the template (`src/view/message/hello.html`) to iterate over existing messages in the database, and then display a form for creating new messages:

```
<ul>
{% if messages %}
  {% for msg in messages %}
    <li>{{ msg.message_contents }}
  {% endfor %}
{% else %}
    <li>No messages!
{% endif %}
</ul>

<form method="post">
Enter a new message:
<textarea name="message_contents">
</textarea>
<input type="submit">
</form>
```

Refresh the browser to verify that a form is being displayed.

Next, alter the controller (`src/controller/cb_tutorial_message_controller.erl`) to retrieve and store messages:

```
hello('GET', []) ->
    Messages = boss_db:find(message, []),
    {ok, [{messages, Messages}]};
hello('POST', []) ->
    MessageContents = Req:post_param("message_contents"),
    NewMessage = message:new(id, MessageContents),
    {ok, SavedMessage} = NewMessage:save(),
    {redirect, [{action, "hello"}]}.
```

We now have a database-driven website. Try submitting a message into the form to verify that it works.

You will probably want to understand all of the lines of code above. The first clause in the controller handles GET requests, and will render the template. The second clause handles POST requests, and will process the incoming form.

In line 2 we've introduced the `boss_db:find/2` function, which queries the database. It takes the model name as the first argument, and a list of conditions as the second argument. We passed in an empty list because we did not want to restrict the returned set in any way.

Line 5 creates a new message, but does not actually save it to the database. Arguments to the `new` function correspond to the parameters that we defined in `src/model/message.erl`.

The first argument is `id`, which tells BossDB that an ID should be generated by the database. We could also specify the ID, for example, if we wished to overwrite an existing record.

The second argument to `message:new/2` is the message contents. We pull this value from the `Req` object, which the keen reader will recall is defined as a parameter to the controller itself. The `Req` object contains a lot of useful information about the current request. Besides GET and POST parameters, the `Req` object contains cookies, the client's IP address, and more.

In line 8, we save the new record to the database. The `save/0` function returns either the saved record, or a list of validation errors. We have not specified any

model validation conditions, so we can assume for the time being that the object is successfully saved.

Finally, the function returns a `{redirect, Location}` directive, which we have not encountered before. This directive performs an HTTP 302 redirect to the specified action. In this example, we redirect to the same URL.

In order to complete our message-board application, we will want to delete unwanted messages from the database.

Add the following form to the bottom of the template (`src/view/message/hello.html`):

```
<form method="post" action="{% url action="goodbye" %}">
Delete:
<select name="message_id">
{% for msg in messages %}
<option value="{{ msg.id }}">{{ msg.message_contents }}
{% endfor %}
</select>
<input type="submit">
</form>
```

Next, we need to process the form. Add the following code to the bottom of the controller (`src/controller/cb_tutorial_message_controller.erl`):

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

Using the form, delete one of the messages you created to confirm that the delete feature works as expected.

3. Simple authentication

You probably do not want just anyone to delete messages from your database. Next we will implement simple authentication based on the client's IP address.

Authentication in Chicago Boss applications occurs in a special function called `before_/1` that optionally appears in each controller. The function's only argument is the name of the action to be authenticated.

Add the following code to the controller (`src/controller/cb_tutorial_message_controller.erl`):

```
before_(_) ->
    {ok, Req:peer_ip() =:= {127, 0, 0, 1}}.
```

This function will return `{ok, true}` if the client is currently sitting at your computer, and `{ok, false}` otherwise. If actions in the controller take a third argument, this return value will be passed in. So to ensure that only local users can delete messages, modify the goodbye function to take this third argument:

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

Now, non-local users will receive an error if they attempt to delete a message.

To avoid confusion, it probably makes more sense to hide the delete form from non-local users. Modify the `hello` function so that the templates have access to user's authentication status:

```
hello('GET', [], IsLocal) ->
    Messages = boss_db:find(message, []),
    {ok, [{messages, Messages}, {is_local, IsLocal}]};
hello('POST', [], IsLocal) ->
    ...
```

Next, modify the template (`src/view/message/hello.html`) to hide the form in the event that the user is non-local:

```
{% if is_local %}
<form method="post" action="{% url action="goodbye" %}">
Delete:
<select name="message_id">
{% for msg in messages %}
<option value="{{ msg.id }}">{{ msg.message_contents }}
{% endfor %}
</select>
<input type="submit">
</form>
{% endif %}
```

Now we have a non-trivial service running: a database-driven web page with basic user authentication. Try accessing the page from another computer to confirm that this feature works.

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

4. Simple validation

Add the following code to the `message` model (`src/model/message.erl`):

```
validation_tests() ->
    [{fun() -> length(MessageContents) > 0 end,
        "Message must be non-empty!"},
     {fun() -> length(MessageContents) =< 140 end,
        "Message must be 140 characters or fewer!"}].
```

This code ensures that no message can be stored to the database unless it is between 1 and 140 characters long.

Next we need to modify the controller and templates to present the validation errors (if any) to the user. Modify the controller (`src/controller/cb_tutorial_message_controller.erl`) accordingly:

```
hello('POST', [], IsLocal) ->
    MessageContents = Req:post_param("message_contents"),
    NewMessage = message:new(id, MessageContents),
    case NewMessage:save() of
        {ok, SavedMessage} ->
            {redirect, [{action, "hello"}]};
        {error, ErrorList} ->
            {ok, [{errors, ErrorList}, {new_msg, NewMessage}]}
    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 message that failed to save.

Now modify the top of the view (`src/view/message/hello.html`) to show any errors in red, and to display the message 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>
{% else %}
<ul>
{% if messages %}
  {% for msg in messages %}
    <li>{{ msg.message_contents }}
  {% endfor %}
{% else %}
    <li>No messages!
{% endif %}
</ul>
{% endif %}

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

...
```

Try entering in an exceedingly long message into the form to confirm that the validation code works as expected.

5. Notifications

To stay informed about new messages in the database, we will take advantage of Chicago Boss's event system, called BossNews. Rather than add notification code to the controller or to the model, we will create a script that sets up event listeners at server startup. These listeners will fire callbacks whenever an event of interest occurs in the database (for example, when a message is created or deleted).

Change the `init/0` function of `priv/init/cb_tutorial_news.erl` to the following:

```
init() ->
    {ok, WatchId} = boss_news:watch("messages",
        fun(created, NewMessage) ->
                error_logger:info_msg("New message! ~p~n",
                    [NewMessage:message_contents()]);
            (deleted, OldMessage) ->
                error_logger:info_msg("Message go poof! ~p~n",
                    [OldMessage:message_contents()])
         end),
    {ok, [WatchId]}.
```

The event listeners are called *watches*. The `boss_news:watch/2` function creates a watch from a topic string and a callback function. The two arguments to the callback are the event name and additional information about the event; for the `created` and `deleted` events above, the event information simply consists of the record that was created or deleted.

The `init/0` function above should return `ok` along with a list of watch IDs that were created; these watches will be destroyed when the script is re-read (presumably, the script will create new watches in their stead).

This script will be read at server startup, but we can also invoke it manually. In the terminal screen that has the server shell, type the following command:

```
boss_web:reload_news().
```

If that function successfully returns `ok`, the database is now being watched for new records. Try creating a new message through the website you built, and then check the server shell to confirm that the notification system works as expected.

Besides keeping administrators informed about activity on the website, the BossNews system can be used to create real-time updates on web pages so that they seem to be more alive than traditional web pages. In the next section we will couple BossNews with a message queue to create a web page that updates itself the same instant that relevant database activity occurs.

6. Real-time updates

Chicago Boss ships with a message queue system called BossMQ. Messaging occurs via *channels*, which are identified by strings. There is no limit to the

number of subscribers on a particular channel. In practice, this means that we can broadcast updates to many users at once.

A real-time update system has three components: 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.

First, open the news script (`priv/init/cb_tutorial_news.erl`) and change it to the following:

```
init() ->
    {ok, WatchId} = boss_news:watch("messages",
        fun(created, NewMessage) ->
            error_logger:info_msg("New message! ~p~n",
                [NewMessage:message_contents()]),
            boss_mq:push("new-messages", NewMessage);
           (deleted, OldMessage) ->
            error_logger:info_msg("Message go poof! ~p~n",
                [OldMessage:message_contents()])
         end, []),
    {ok, [WatchId]}.
```

Before we forget, go ahead and reload the news script in the server shell:

```
boss_web:reload_news().
```

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

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

The call to `boss_mq:pull/2` will block until new messages are available; the technique is known as a long poll. (Note that in most languages, an indefinitely blocking call like this would result in significantly degraded server performance, but Erlang's internal design ensures that our server will keep running smoothly, even with a large number of clients waiting for new messages.)

To ensure that no messages are missed by a particular client, this pull action requires a timestamp that specifies the time of the earliest message we want to receive. To get things going, we will need an initial timestamp, which can be retrieved from `boss_mq:now/1`. Add a new "live" action like thus:

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

With that, the server code for real-time updates is finished; now we just need to add JavaScript to the template to perform the actual polling and to insert new messages into the web page. Create a new file `src/view/message/live.html`, and add the following as its contents:

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

To test the feature, open a window for http://localhost:8001/message/hello and a separate window for the new page, http://localhost:8001/message/live. Create a message in the first window and confirm that it automatically appears in the second.

If you thought that was fun, see if you can implement a "live delete" where deleted messages are instantly excised from the browser.

7. Users and log-ins

Unless you are making a website for purely for personal consumption over the loopback interface, you will probably want a log-in system of some kind. This section demonstrates how to implement one.

The first task is to come up with a name for your user model. Despite the obvious temptation, you should not name your user model `user`. It is an unfortunate accident of history that the engineers at Ericsson AB chose the name `user` for a privileged system library; as a consequence, you are forced to choose something else. One alternative is to call the model `account`; another to is to prefix the model name with the application name, for example, `cb_tutorial_user`. For the remainder of the tutorial, I will save the both of us a few keystrokes and call the user model `account`.

Create a new file called `src/model/account.erl`, and insert the following:

```
-module(account, [Id, Email, PasswordHash]).
-compile(export_all).
```

Our users will use their email addresses to log in. Developers familiar with authentication systems will correctly guess that `PasswordHash` will be used to check a user's password without storing the actual password in the database. The following functions will do the job, so add them in:

```
hash_for_password(Password) ->
    PasswordSalt = mochihex:to_hex(erlang:md5(Email)),
    mochihex:to_hex(erlang:md5(PasswordSalt ++ Password)).

check_password(Password) ->
    hash_for_password(Password) =:= PasswordHash.
```

We will see how to use these functions later.

Before leaving the account model, trivial validation we should perform on new accounts is to check that the email address contains an @ symbol. Add this code to the account model:

```
validation_tests() ->
    [{fun() -> string:chr(Email, $@) > 0 end,
        "Email must contain an @ symbol!"}].
```

Next we will create a workflow for logging in and logging out of the website; afterwards, we will go back and require a log-in to view a page that we previously made.

Create a new file `src/controller/cb_tutorial_account_controller.erl`. Use the following as its initial contents:

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

Note the new `SessionID` parameter, which is automatically generated for each request and available to all of the controller functions. We will use the session ID along with the password-checking code above to implement a login action and a logout action:

```
login('GET', []) ->
    {ok, [{redirect, Req:header(referer)}]};
login('POST, []) ->
    Email = Req:post_param("email"),
    Password = Req:post_param("password"),
    Redirect = Req:post_param("redirect"),
    Result = case boss_db:find(account, [email = Email]) of
        [Account] ->
            case Account:check_password(Password) of
                true ->
                    boss_session:set_session_data(SessionID,
                                                  "account",
                                                  Account:id()),
                    true;
                false -> false
            end;
         _ -> false
    end,
    case Result of
        true -> {redirect, Req:post_param("redirect")};
        false ->
```

```
            {ok, [{redirect, Redirect}, {email, Email},
              {errors, ["Invalid email/password combination"]}]}
    end.

logout('GET', []) ->
    boss_session:remove_session_data(SessionID, "account"),
    {redirect, "/"}.
```

The essential strategy will be to accept GET requests to the login action and embed the referring URL in the log-in form. If the log-in attempt succeeds, we store the user's account ID in the current session (with `boss_session:set_session_data/3`) and send them on their merry way (with `{redirect, Req:post_param("redirect")}`). If the attempt fails, we will display the form again, this time filled in with the email address that the user previously submitted. Logging out consists of removing the account ID from the current session (using `boss_session:remove_session_data/2`) and redirecting the user to the home page.

Next we add an action for creating a new account. Note that no action should not be named `new`, because `new` is used as the instantiation function of parameterized modules. This is the second Great Temptation of Chicago Boss programming (the first was naming the user model `user`), and must be avoided with equal resolve. Choose `create` instead:

```
create('GET', []) ->
    ok;
create('POST', []) ->
    Email = Req:post_param("email"),
    Password1 = Req:post_param("password1"),
    Password2 = Req:post_param("password2"),
    ValidationTests = [{length(Password1) >= 6,
            "Password must be at least 6 characters!"},
         {Password1 =:= Password2,
            "Passwords didn't match!"},
         {boss_db:count(account, [email = Email]) =:= 0,
            "An account with that email exists"}],
    ValidationFailures = lists:foldr(fun
            ({true, _}, Acc) -> Acc;
            ({false, Message}, Acc) -> [Message|Acc]
        end, [], ValidationTests),
    Account = boss_record:new(account, [{email, Email}]),
    ErrorList = case ValidationFailures of
        [] ->
            PasswordHash = Account:hash_for_password(Password1),
```

```
            Account1 = Account:set(password_hash, PasswordHash),
            case Account1:save() of
                {ok, SavedAccount} ->
                    boss_session:set_session_data(SessionID,
                        "account", SavedAccount:id()),
                    [];
                {error, List} -> List
             end;
        _ -> ValidationFailures
    end,
    case ErrorList of
        [] -> {redirect, [{controller, "message"},
                  {action, "hello"}]};
        _ -> {ok, [{errors, ErrorList}, {new_account, Account}]}
    end.
```

The strategy here is to validate the form, and if it succeeds, then attempt to create the account. Remember that we defined an additional validation test in the account model, so we make sure to pick up that error as well in the final `ErrorList`.

The code above is somewhat complicated by the fact that we are not storing the password as submitted; instead, we first instantiate `Account`, and then set its `password_hash` field using the `hash_for_password/1` function defined earlier. Since Erlang variables are immutable, the account with the password hash is stored in `Account1`, which we then try to save. If the saving succeeds, we get a third account variable called `SavedAccount`; it differs from the previous two in that it will have an ID generated by BossDB. We store that ID in the session with `boss_session:set_session_data/3`, at which point the account has been created and the user is now logged to the website.

The final piece of the puzzle is to create the HTML forms to go along with the controller we just created. To conserve keystrokes, we will move the error presentation code into its own template. In the file `src/view/lib/list_errors.html`, we have:

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

We can now use this template in both the log-in template and the account-creation template. Create a `src/view/account` directory, and then:

In the file `src/view/account/login.html`:

```
{% list_errors errors=errors %}

<form method="post">
Email address: <input name="email"
    value="{% if email %}{{ email }}{% endif %}"><br>
Password: <input name="password" type="password"><br>
<input type="submit">
</form>
<a href="{% url action="create" %}">Create an account</a>
```

In the file `src/view/account/create.html`:

```
{% list_errors errors=errors %}

<form method="post">
Email address: <input name="email"
     value="{% if account %}{{ account.email }}{% endif %}"><br>
Password once: <input name="password1" type="password"><br>
Password again: <input name="password2" type="password"><br>
<input type="hidden" name="redirect" value="{{ redirect }}">
<input type="submit">
</form>
```

The essential functionality is finished. All that remains is to password-protect the pages we created previously. Open the `message` controller from before (`src/controller/cb_tutorial_message_controller.erl`), and change the module declaration to take advantage of the session ID:

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

Next, change the `before_/1` function to:

```
before_(_) ->
    case boss_session:get_session_data(SessionID, "account") of
        undefined ->
            {redirect, [{controller, "account"},
                {action, "login"}]};
        AccountID ->
```

```
            {ok, Req:peer_ip() =:= {127, 0, 0, 1}}
    end.
```

Nothing else in the controller needs to be changed. Try navigating to http://localhost:8001/message/hello to verify that our brand-new account system successfully protects the page. For extra credit, add a message to the page that indicates who is currently logged in.

8. Database associations

So far we have two unconnected models (`account` and `message`). In this section we will link them together.

The relationship will be one-to-many; an account can have many messages, but each message will belong to only one account.

To establish the association, alter the top of message model (`src/model/message.erl`) to the following:

```
-module(message, [Id, MessageContents, AccountId]).
-compile(export_all).
-belongs_to(account).
```

Each message will now have a function `account/0` for accessing its associated account. To make use of this association in the application, we need to make slight alterations to the view and controller.

First, change the `hello` action of the message controller (`src/controller/cb_tutorial_message_controller.erl`):

```
hello('POST', [], IsLocal) ->
    MessageContents = Req:post_param("message_contents"),
    AccountID = boss_session:get_session_data(SessionID,
                                              "account"),
    NewMessage = message:new(id, MessageContents, AccountID),
    case NewMessage:save() of
        ...
    end.
```

Next, change the "hello" view to display the email address of the account associated with each message. To simplify things, we'll also take this

opportunity to use the `list_errors` template that we created for the account system.

```
{% if errors %}
{% list_errors errors=errors %}
{% else %}
<ul>
{% if messages %}
  {% for msg in messages %}
    <li>{{ msg.message_contents }}
        {% if msg.account %}--{{ msg.account.email }}{% endif %}
  {% endfor %}
{% else %}
    <li>No messages!
{% endif %}
</ul>
{% endif %}
...
```

New messages will now have the creator's email address displayed after the message contents. Try it out.

Just to round things out, we will now create an account page where you can view all the messages associated with a particular account.

First we need to establish the reciprocal relationship from one account to many messages. Open the account model (`src/model/account.erl`) and add this line before the function definitions:

```
-has({messages, many}).
```

The account model will now have a function called `messages/0` that retrieves that account's messages.

Next, we add a `view` action to the `account` controller (`src/controller/cb_tutorial_account_controller.erl`):

```
view('GET', [Id]) ->
    Account = boss_db:find(Id),
    {ok, [{account, Account}]}.
```

The call to `boss_db:find/1` is new to our application. `boss_db:find/1` retrieves a record with a given ID. As you might imagine, it is highly useful.

Next, we create a template for the new action. Create a file `src/view/account/view.html`, and put this into it:

```
<h1>{{ account.email }}</h1>
All messages:
<ul>
{% for msg in account.messages %}
    <li>{{ msg.message_contents }}
{% endfor %}
</ul>
```

Finally, we will create a link to this page from each displayed message. Open `src/view/message/hello.html`, and change:

```
{{ msg.account.email }}
```

to

```
<a href="{% url controller="account" action="view"
id=msg.account.id %}">{{ msg.account.email }}</a>
```

There is some subtlety in the `url` tag as we have used it above. To generate a URL, the `id` argument is matched to the `[Id]` argument in the controller action. The ID will thus be appended to the URL as a token (such as `/account/view/account-1`). If the controller action instead took `["foo", Id, "bar"]` as its second argument, the URL generated by the `url` tag would be `/account/view/foo/account-1/bar` in order to match it. Arguments to the `url` tag that cannot be matched to variables in the controller action will be appended as a query string (such as `?foo=bar`).

In any event, the new feature is ready for use. Reload http://localhost:8001/message/hello and click on your email address to see it in action.

9. Developer exhortation

Well, that's all for now. This tutorial has left out all the best parts of Chicago Boss: the elegant testing framework, the compact query language, the email templates and processors, the event-driven caching strategy, the seamless internationalization (i18n) support, and all the Erlang features that make CB a breeze to operate in a production environment. But it's 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 developer community.