Introduction to Nitrogen

A step-by-step introduction to the
major features and concepts behind
the Nitrogen Web Framework.

Agenda

- Part 1: Install & Run Nitrogen
- Part 2: Nitrogen Pages
- Part 3: Nitrogen Elements
- Part 4: Nitrogen Actions
- Part 5: Nitrogen Postback Events
- Part 6: Session and Page State
- Part 7: Security
- Part 8: Validation
- Part 9: Comet
- Part 10: Extending Nitrogen
- Conclusion

Install & Run Nitrogen

- Install Nitrogen
- Run the Website
- A Tour Through the Files

Install Nitrogen

If you don't have Erlang Installed:

Download Nitrogen, unzip and **`cd nitrogen`**.

If you do have Erlang installed:

Pull the Nitrogen Source Code, then **`make rel_inets; cd rel/nitrogen`**.

Install & Run Nitrogen

Start Up

```
bin/nitrogen console
open http://localhost:8000
```

Install & Run Nitrogen

Shut Down

Press Control-C twice.

View the Directory

```
ls -l
```

Install & Run Nitrogen

Anatomy of a Nitrogen Project

BuildInfo.txt

From **`uname`**.

Makefile

Used by **`make`**.

bin/

Commands to start and stop system, plus developer tools.

etc/

Configuration settings.

site/

Contains the website files, templates, and Erlang modules.

log/

The logs.

doc/

Contains Nitrogen documentation.

erts-5.7.5/

Embedded Erlang.

releases/

Tells Erlang how to start the system.

lib/

Dependent libraries.

The `site/` Directory

The site directory should go under source control, it contains all of the information necessary to run the website.

Emakefile
: Used by make.erl to compile the system.

ebin/
: Compiled Erlang modules.

include/
: Include files for your website.

src/
: Erlang source files for your website.

static/
: Static files for your website.

templates/
: Template files for your website.

Install & Run Nitrogen

The `site/src/` Directory

Stores the Erlang source files for your application. By default it contains:

nitrogen_init.erl
: Runs once on Nitrogen startup.

nitrogen_PLATFORM.erl
: Holds the request loop depending on platform.

index.erl
: The default web page.

elements/
: By convention, custom alements are placed here.

actions/
: By convention, custom actions are placed here.

Install & Run Nitrogen

Exercise: Modify Your First Page

- Open **`site/src/index.erl`**
- Change "Welcome to Nitrogen" to "Welcome to My Website"
- From the Erlang Shell, run:

```
sync:go()
```

- Reload the page

Install & Run Nitrogen

Exercise: Compile in a Different Way

- Change to "Welcome to my ERL-TASTIC WEBSITE!" (or, you know, whatever)
- From a different terminal, run:

```
bin/dev compile
```

- Reload the page

Install & Run Nitrogen

Debug Statements

- Add **`?DEBUG`** to **`index.erl`**. Then compile and reload. What happens?
- Add **`?PRINT(node())`** to **`index.erl`**. Then compile and reload. What happens?

Install & Run Nitrogen

Emacs `nitrogen-mode`

```
(add-to-list 'load-path "PATH/TO/NITROGEN/support/nitrogen-mode")
(require 'nitrogen-mode)
```

Without **`nitrogen-mode`**:

```
#panel { id=my_panel, body=[
                            #panel { id=my_panel2, body=[
                                                          #label { text="Name" },
                                                          #textbox { id=my_textbox }
                                                         ]}
                           ]}
```

With **`nitrogen-mode`**:

```
M-x nitrogen-mode
#panel { id=my_panel, body=[
    #panel { id=my_panel2, body=[
        #label { text="Name" },
        #textbox { id=my_textbox }
    ]}
]}
```

Nitrogen Pages

- What is a Nitrogen Page?
- Dynamic Routing Explained
- Creating Your First Page
- How is a Page Rendered?
- Anatomy of a Template
- Experimenting With Templates

Nitrogen Pages

What is a Nitrogen Page

- A Page is an Erlang Module
- Each page should accomplish one store or piece of functionality.

  Some examples:

  - 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`**)

Nitrogen Pages

Dynamic Routing Explained

Dynamic routing rules:

1. If there is an extension, assume a static file.

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

2. Root page maps to **`index.erl`**
3. Replaces slashes with underscores.

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

4. Try the longest matching module.

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

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

Nitrogen Pages

Exercise: Create a New Page

- Generate the Page

```
bin/dev page my\_page
$EDIT site/src/my\_page.erl
```

- Replace the default body with:

```
body() -> "Hello World!".
```

- Remove the **`event/1`** function.
- Compile the page and load **`http://localhost:8080/my/page`**

Nitrogen Pages

How is a Page Rendered?

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

(This is the simple version. Complex version will come later.)

Nitrogen Pages

Anatomy of a Template

- HTML. The Page is slurped into the Template.
- Contains one or more callouts, ie:

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

- Contains a script callout for Javascript:

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

- The callouts look like Erlang, but they are not. They can only be of the form **`module:function(Args)`**. The 'page' module refers to the current page.

Nitrogen Pages

Experimenting With Templates

- Change the callout from **`page:body()`** to **`page:body1()`** in the default template and reload the page. What happens?
- Create another callout. What happens?
- What happens when you change **`page`** to be a specific module?
- Replace the module call with some arbitrary Erlang code. What happens?

Nitrogen Elements

- What is a Nitrogen Element?
- Add Elements to Your Page
- Nested Elements
- Documentation
- Anatomy of a Nitrogen Element

Nitrogen Elements

What is a Nitrogen Element?

An element can be either HTML, or some record that renders into HTML.

Change this:

```
body() -> "Hello World!".
```

To this:

```
body() -> #label { text="Hello World!" }.
```

Nitrogen Elements

What is a Nitrogen Element?

The **`#label{}`** element is rendered into:

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

View the rendered page source in your browser and search for "Hello World".

Nitrogen Elements

Why Nitrogen Elements?

Nitrogen elements serve two purposes:

1. Allow you to generate HTML within Erlang:
    - Avoid mixing languages == clearer code.
    - Fewer characters to type.
    - Checked at compile time.
2. Abstraction layer:
    - Avoid repeating common functionality.
    - Hide complexity in a module.

Nitrogen Elements

Nitrogen Element Examples

Try this on my_page.erl:

```
body() -> [
    #h1 { text="My Simple Application" },
    #label { text="What is your name?" },
    #textbox { },
    #button { text="Submit" }
].
```

Then compile, reload, and view source.

Nitrogen Elements

Nested Elements

Try a nested element:

```
body() ->
    #panel { style="margin: 50px;", body=[
        #h1 { text="My Page" },
        #label { text="Enter Your Name:" },
        #textbox { },
        #button { text="Submit" }
    ]}.
```

Nitrogen Actions

- What is a Nitrogen Action?
- Wiring an Action
- Conditional Actions with **`#event{}`**
- Postbacks

Nitrogen Actions

What is a Nitrogen Action?

An action can either be Javascript, or some record that renders into Javascript.

Add a Javascript alert to the **`#button{}`** element. Then recompile and run. What do you expect will happen?

```
body() ->
    [
        #button { text="Submit", actions="alert('hello');" }
    ].
```

Nitrogen Actions

What is a Nitrogen Action?

Do the same thing a different way.

```
body() ->
    [
        #button {
            text="Submit",
            actions=#alert { text="Hello" }
        }
    ].
```

Nitrogen Actions

Wiring an Action

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.

```
body() ->
    wf:wire(mybutton, #effect { effect=pulsate }),
    [
        #button { id=mybutton, text="Submit" }
    ].
```

Nitrogen Actions

Conditional Actions with `#event{}`

Put the **`#effect{}`** action inside of an **`#event{}`** action. This causes the effect to **only** get fired if the user clicks on **`mybutton`**.

```
body() ->
    wf:wire(mybutton, #event {
        type=click,
        actions=#effect { effect=pulsate }
    }),
    [
        #button { id=mybutton, text="Submit" }
    ].
```

Nitrogen Actions

Triggers and Targets

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

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

Try this:

```
body() ->
    wf:wire(#event {
        type=click, trigger=mybutton, target=mylabel,
        actions=#effect { effect=pulsate }
    }),
    [
        #label { id=mylabel, text="Make Me Blink!" },
        #button { id=mybutton, text="Submit" }
    ].
```

Triggers and Targets

You can also specify the **Trigger** and **Target** directly in **`wf:wire/N`**. It takes three forms:

```
% Specify a trigger and target.
wf:wire(Trigger, Target, Actions)

% Use the same element for both trigger and target.
wf:wire(TriggerAndTarget, Actions)

% 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.)
wf:wire(Actions)
```

Nitrogen Actions

Quick Review

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

Nitrogen Events

- What is a Postback?
- Your First Postback
- Event Properties
- More Event Examples
- Postback Shortcuts
- Modifying Elements

Nitrogen Events

What is a Postback?

A postback briefly transfers control from the browser to the Nitrogen server. It is initiated when an event fires with the **`postback`** property set. For example:

```
#event { type=click, postback=my_click_event }
```

The postback tag can be any valid Erlang term. You use this to differentiate incoming events.

Nitrogen Events

Your First Postback

First, let's use the postback to print out a debug message.

```
body() ->
    wf:wire(mybutton, #event { type=click, postback=myevent }),
    [
        #button { id=mybutton, text="Submit" }
    ].

event(myevent) ->
    ?PRINT({event, now()}).
```

Nitrogen Events

Postback Shortcuts

A few elements allow you to set the **`postback`** property as a shortcut to handle their most common events.

Element	Shortcut Event
`#button{}`	click
`#textbox{}`	enter key
`#checkbox{}`	click
`#dropdown{}`	change
`#password{}`	enter key

Nitrogen Events

Postback Shortcuts

A few elements allow you to set the **`postback`** property as a shortcut to handle their most common events.

The previous code, simplified:

```
body() ->
    [
        #button { id=mybutton, text="Submit", postback=myevent }
    ].

event(myevent) ->
    ?PRINT({event, now()}).
```

Nitrogen Events

More Event Examples

```
body() ->
    % 'mouseover', 'click', and 'mouseout' are standard Javascript
    % events.
    wf:wire(mybutton, [
        #event { type=mouseover, postback=my_mouseover_event }
        #event { type=click, postback=my_click_event }
        #event { type=mouseout, postback=my_mouseout_event }
    ]),
    [
        #button { id=mybutton, text="Submit" }
    ].

event(my_click_event) ->
    ?PRINT({click, now()});
event(OtherEvent) ->
    ?PRINT({other, MyEvent, now()}).
```

Nitrogen Events

More Event Examples

Generally, a postback is a good chance to read form elements. The **`wf:q(ElementID)`** function does this.

```
body() ->
    [
        #textbox { id=mytextbox, text="Edit this text." },
        #button { id=mybutton, text="Submit", postback=myevent }
    ].

event(myevent) ->
    Text = wf:q(mytextbox),
    ?PRINT({event, Text}).
```

Nitrogen Events

Modifying Elements

Here is where everything comes together: we are going to modify the page from within a postback event. Nitrogen uses **AJAX** to update parts of a page without updating the entire page.

```
body() ->
    #panel { style="margin: 50px;", body=[
        #button { id=mybutton, text="Submit", postback=click },
        #panel { id=placeholder }
    ]}.

event(click) ->
    wf:update(placeholder, [
        #h1 { text="Congratulations!" },
        #p { body="You have updated the page!" },
        #p { body=io_lib:format("~p", [now()]) }
    ]).
```

Nitrogen Events

More Page Manipulation

The **`wf`** module exposes many manipulation functions:

`wf:update/2`
: Update the contents of an element with another element(s).

`wf:insert_top/2`
: Insert a new element(s) at the beginning of another element.

`wf:insert_bottom/2`
: Insert a new element(s) at the bottom of another element.

`wf:replace/2`
: Replace an element with another element.

`wf:remove/1`
: Remove an element(s).

`wf:set/2`
: Set a textbox or checkbox value.

It also exposes many other generally useful utility functions: http://nitrogenproject.com/doc/api.html

Remembering State

- Page State vs. Session State
- Page State Example
- Session State Example

Remembering State

Page State vs. Session State

Nitrogen can store two kinds of state:

- **Page State**
  - Stored in a user's browser window.
  - Destroyed when the user closes the window or navigates to a different page.
  - Sent across the wire with each request.
- **Session State**
  - Stored in server memory.
  - Destroyed when the session expires or the Erlang VM dies.
  - Associated with the user's session by an HTTP cookie.
  - Useful place to store authentication

Remembering State

Page State

Using Page State:

```
% Set a state variable
wf:state(Key, Value)

% Get a state variable
wf:state(Key)
wf:state_default(Key, DefaultValue)
```

`Key` and **`Value`** can be any valid Erlang term.

Exercise: Modify my_page.erl to display a counter that gets incremented every time you press the 'Submit' button. The counter should reset when the user reloads the page.

Remembering State

Page State

```
body() ->
    #panel { style="margin: 50px;", body=[
        #button { id=mybutton, text="Submit", postback=click },
        #panel { id=placeholder, body="1" }
    ]}.

event(click) ->
    Counter = wf:state_default(counter, 1),
    wf:update(placeholder, [
        #panel { body=io_lib:format("~p", [Counter + 1]) }
    ]),
    wf:state(counter, Counter + 1).
```

Remembering State

Session State

Using Session State:

```
% Set a session state variable
wf:session(Key, Value)

% Get a session state variable
wf:session(Key)
wf:session_default(Key, DefaultValue)
```

`Key` and **`Value`** can be any valid Erlang term.

Exercise: Modify my_page.erl to display **TWO** counters. When the user presses the 'Submit' button, one counter should get incremented, the other counter should get doubled. The server should remember the counter even if the closes and then re-opens the browser.

Remembering State

Session State

```
body() ->
    #panel { style="margin: 50px;", body=[
        #button { id=mybutton, text="Submit", postback=click },
        #panel { id=placeholder1, body="1" },
        #panel { id=placeholder2, body="1" }
    ]}.

event(click) ->
    %% Increment the counter...
    Counter1 = wf:session_default(counter1, 1),
    wf:update(placeholder1, io_lib:format("~p", [Counter1 + 1])),
    wf:session(counter1, Counter1 + 1),

    %% Double the other counter...
    Counter2 = wf:session_default(counter2, 1),
    wf:update(placeholder2, io_lib:format("~p", [Counter2 * 2])),
    wf:session(counter2, Counter2 * 2).
```

Security

- Limiting Access to a Page
- Authentication and Authorization Functions
- Page Redirection Functions
- Creating a Secure Page

Security

Limiting Access to a Page

Nitrogen contains functions to help you build password protected websites:

- Nitrogen is built for role-based security. You set the roles for a current session, and check those roles later.

  For example, the user may have the **`friend`** and **`manager`** roles, but not the **`administrator`** role.

- Authentication/authorization info is stored in the session.

Security

Authentication and Authorization Functions

Functions to set the user/role:

```
% Get/set the current user for this session.
wf:user(), wf:user(User)

% Get/set whether the current session has the specified role.
wf:role(Role), wf:role(Role, IsInRole)
```

Security

Page Redirection Functions

Functions kick the user to a login page:

```
% Redirect the user to a different page.
wf:redirect(Url)

% Redirect the user to the login page.
wf:redirect_to_login(LoginUrl)

% Redirect the user back to the original page they
% tried to access.
wf:redirect_from_login(DefaultUrl)
```

Security

Creating a Secure Page - Step 1

Check for the **managers** role at the top of a page. If the user doesn't have the role, go to a login page.

```
main() ->
    case wf:role(managers) of
        true ->
            #template { file="./site/templates/bare.html" };
        false ->
            wf:redirect_to_login("/login")
    end.
```

Security

Creating a Secure Page - Step 2

Create a login page. For now, just create a button that, when clicked, grants the **`managers`** role to the user and redirects back.

```
body() ->
    #button { text="Login", postback=login }.

event(login) ->
    wf:role(managers, true),
    wf:redirect_from_login("/").
```

Security

Creating a Secure Page - Step 3

Update **`login.erl`** to prompt for a username and password.

```
body() ->
    #panel { style="margin: 50px;", body=[
        #flash {},
        #label { text="Username" },
        #textbox { id=username, next=password },
        #br {},
        #label { text="Password" },
        #password { id=password, next=submit },
        #br {},
        #button { text="Login", id=submit, postback=login }
    ]}.

event(login) ->
    case wf:q(password) == "password" of
        true ->
            wf:role(managers, true),
            wf:redirect_from_login("/");
        false ->
            wf:flash("Invalid password.")
    end.
```

Creating a Secure Page - Step 4

Create a way for the user to logout.

```
% Clears all user, roles, session state, and page state.
wf:logout()
```

Note: Placing this statement appropriately is left as an exercise for the reader.

Validation

- Overview of Nitrogen Validation
- Adding Some Validators

Validation

Overview of Nitrogen Validation

Nitrogen implements a validation framework, plus a number of pre-built validators, to allow you to declaratively validate your form variables.

Validation happens on both client side (using the LiveValidation library) and server side (in Erlang).

This is done to present a responsive front end to the user

Validation

Overview of Nitrogen Validation

The simplest validator is the **`#is_required{}`** validator. Tell your **`login.erl`** page to make sure the user enters both a username and a password.

```
body() ->
    wf:wire(submit, username, #validate { validators=[
        #is_required { text="Required." }
    ]}),
    wf:wire(submit, password, #validate { validators=[
        #is_required { text="Required." }
    ]}),
    #panel { style="margin: 50px;", body=[
        ...
```

Validation

Overview of Nitrogen Validation

We can get clever and use a validator to check that the user entered the correct password. The **`#custom{}`** validator runs on the server. (To make a custom client-side validator, use **`#js_custom{}`**.)

```
body() ->
    wf:wire(submit, username, #validate { validators=[
        #is_required { text="Required." }
    ]}),
    wf:wire(submit, password, #validate { validators=[
        #is_required { text="Required." },
        #custom {
            text="Invalid password.",
            function=fun(_, Value) -> Value == "password" end
        }
    ]}),
    #panel { style="margin: 50px;", body=[
        ...
```

Validation

Overview of Nitrogen Validation

Since we validate the password in the **`#custom`** validator, we can trust that the **`login`** event only fires when the password is correct. Change the **`login`** event to:

```
event(login) ->
    wf:role(managers, true),
    wf:redirect_from_login("/").
```

Comet

- What is Comet?
- Comet the Nitrogen/Erlang Way
- A Comet Counter
- Comet Pools
- Comet Pool Scope
- The Simplest Chatroom Ever Constructed

Comet

What is Comet?

Comet is the name for a technique where the browser requests something from the server, and the server doesn't respond until it has something useful to say.

This makes it useful for applications that need fast, out-of-band communication, such as chat rooms.

In other words, you don't need to keep hitting a "Get Messages" button. The server just pushes messages when they are available.

A big happy shout out to Tom McNulty for his innovative ideas on what Comet support could look like in Nitrogen.

Comet

Comet the Nitrogen/Erlang Way

Think of Comet like **`erlang:spawn/1`**:

- Start up a function.
- The function can manipulate the page using **`wf:update/2`** or any other page manipulation function.
- Output is queued until the function ends or calls **`wf:flush/0`**.
- The function acts like it is linked to the current user's page. It is killed when the user leaves the page (or receives **`{'EXIT', _, Message}`** if **`trap_exit`** is **`true`**.)

Comet

A Comet Counter

Update **`my_page.erl`** to count once per second.

```
body() ->
    wf:comet(fun() -> counter(1) end),
    #panel { id=placeholder }.

counter(Count) ->
    timer:sleep(1000),
    wf:update(placeholder, integer_to_list(Count)),
    wf:flush(),
    counter(Count + 1).
```

Comet

Comet Pools

You can tell a Comet function to start in a pool by providing a **`PoolName`**. The **`PoolName`** can be any Erlang term.

```
wf:comet(Fun, PoolName)
```

Now you can send messages to the pool. The messages will be received by other functions started in that comet pool.

```
wf:send(PoolName, Message)
```

Comet

Comet Pool Scope

So far, we've been creating **local** comet pools. Nitrogen also has the idea of **global** comet pools:

- **Local** comet pools are walled around the current page and the current user. If the user reloads the page, the comet process(es) goes away.
- **Global** comet pools exist to help you create multi-user applications. They pool is accessible by **all** pages and **all** users.

```
%% Create a global comet pool.
wf:comet_global(Function, PoolName)

%% Send a global comet message.
wf:send_global(PoolName, Message)
```

Comet

The Simplest Chatroom Ever Constructed

Here we're going to create a page that listens for some text, and sends it to the global comet pool. Connect with different browsers and chat to yourself.

```
body() ->
    wf:comet_global(fun() -> repeater() end, repeater_pool),
    [
        #textbox { id=msg, text="Your message...", next=submit },
        #button { id=submit, text="Submit", postback=submit },
        #panel { id=placeholder }
    ].

event(submit) ->
    ?PRINT(wf:q(msg)),
    wf:send_global(repeater_pool, {msg, wf:q(msg)}).

repeater() ->
    receive
        {msg, Msg} -> wf:insert_top(placeholder, [Msg, "<br>"])
    end,
    wf:flush(),
    repeater().
```

Extending Nitrogen

- Custom Elements
- Custom Actions
- Handlers

Extending Nitrogen

Custom Elements - Part 1

You can create custom elements to encapsulate other elements. There is no difference between a **custom** element and a **built-in** element, except where the actual files are stored.

Create a new custom element in **`site/src/elements/my_element.erl`**.

```
./bin/dev element my_element
```

Extending Nitrogen

Custom Elements - Part 2

An element has:

1. A **record** containing the properties of the element.
2. A **`reflect()`** function, providing a programattic way to get the properties of an element. If **`record_info(fields, RecordType)`** worked, this would not be necessary.)
3. A **`render_element(Record)`** function that emits HTML or other elements.

Extending Nitrogen

Custom Elements - Part 3

Let's make an element that displays a textbox and a button, logs the result of the textbox to the console, and then calls a method on the main page.

```
render_element(#my_element{}) ->
    TextboxID = wf:temp_id(),
    ButtonID = wf:temp_id(),
    wf:wire(ButtonID, #event {
        type=click,
        delegate=?MODULE,
        postback={click, TextboxID}
    }),
    [
        #textbox { id=TextboxID, text="Your text...", next=ButtonID },
        #button { id=ButtonID, text="Submit" }
    ].

event({click, TextboxID}) ->
    Text = wf:q(TextboxID),
    ?PRINT({clicked, TextboxID, Text}),
    PageModule = wf:page_module(),
    PageModule:my_element_event(Text).
```

Extending Nitrogen

Custom Elements - Part 4

Now, use the element on **`my_page.erl`**. Remember to move the element into **`include/records.hrl`** first!

```
body() ->
    #my_element {}.

my_element_event(Text) ->
    ?PRINT(Text).
```

For more examples, see the built-in elements under nitrogen/src/elements.

Extending Nitrogen

Custom Actions - Part 1

A custom **action** is like a custom **element**, except it should emit Javascript or other actions.

```
./bin/dev action my_action
```

Extending Nitrogen

Custom Actions - Part 2

Let's make a custom action that calls **`#alert{}`** with a specified string, but converted to all uppercase.

```
-record(my_action, {?ACTION_BASE(action_my_action), text}).

render_action(Record = #my_action{}) ->
    #alert { text=string:to_upper(Record#my_action.text) }.
```

Extending Nitrogen

Custom Actions - Part 3

Now, use the element on **`my_page.erl`**. Remember to move the action into **`include/records.hrl`** first!

```
body() ->
    wf:wire(#my_action { text="this is a message" }),
    #label { text="You should see an alert." }.
```

For more examples, see the built-in actions under nitrogen/src/actions.

Extending Nitrogen

Handlers - Part 1

Handlers are an attempt to formalize an approach for overriding core Nitrogen behavior.

Handlers exist for:

- Configuration
- Logging
- Process Registry
- Caching
- Session Storage
- Page State Storage
- User Identity
- Roles
- Routing
- Security

Extending Nitrogen

Handlers - Part 2

Handlers are initialized in the order described on the previous page. This means that any handler can access and override information defined by a handler that came before it.

For example, you could write a **`route_handler`** that behaved differently depending on the role of a user.

Extending Nitrogen

Handlers - Part 3

Let's make a **`security_handler`** handler that only allows the user to access modules beginning with the word "my".

```
-module(my_security_handler).
-behaviour(security_handler).
-export([init/2, finish/2]).
-include_lib("nitrogen/include/wf.hrl").

init(_Config, State) ->
    ?PRINT(wf:page_module()),
    case wf:to_list(wf:page_module()) of
        "my" ++ _ ->
            {ok, State};
        "static_file" ->
            {ok, State};
        _ ->
            wf_context:page_module(access_denied),
            {ok, State}
    end.

finish(_Config, State) ->
    {ok, State}.
```

Extending Nitrogen

Handlers - Part 3

Now, install the handler in **`nitrogen_inets.erl`**:

```
do(Info) ->
    RequestBridge = simple_bridge:make_request(inets_request_bridge, Info),
    ResponseBridge = simple_bridge:make_response(inets_response_bridge, Info),
    nitrogen:init_request(RequestBridge, ResponseBridge),
    nitrogen:handler(my_security_handler, []),
    nitrogen:run().
```

Conclusion

By now, you should have a basic understanding of how Nitrogen works, and know enough to be able to quickly grok the examples on http://nitrogenproject.com and apply them to your own pages.

Things not covered in this tutorial:

- Drag and Drop
- Sorting
- Binding
- More Effects
- File Uploads
- Javascript API
- Custom Valiators
- Handlers

Thanks

- Mailing List, Bugs, etc: http://nitrogenproject.com/community
- Follow me on Twitter: @rklophaus

Author: Rusty Klophaus

Date: 2010-12-06 08:54:01 EST

HTML generated by org-mode 7.01h in emacs 24