

TU PRIMER SERVIDOR EN ERLANG CON SSE

UTILIZANDO COWBOY Y SUMO DB

Fernando Benavides (@elbrujoalcon)

Inaka Labs

November 28, 2013

inaka

HELLO WORLD!

- Soy programador desde que tenía 10 años 🤖
- Hago programación *funcional* hace 5 años, en *Erlang*
- Soy *Director of Engineering* en **Inaka**
- Me dedico a diseñar y, a veces, construir servidores
- **No soy** un programador *Ruby*
- Qué hago acá?

inaka

HELLO WORLD!

- Soy programador desde que tenía 10 años
- Hago programación *funcional* hace 5 años, en *Erlang*
- Soy *Director of Engineering* en **Inaka**
- Me dedico a diseñar y, a veces, construir servidores
- **No soy** un programador *Ruby*
- Qué hago acá?

HELLO WORLD!

- Soy programador desde que tenía 10 años
- Hago programación *funcional* hace 5 años, en *Erlang*
- Soy *Director of Engineering* en **Inaka**
- Me dedico a diseñar y, a veces, construir servidores
- **No soy** un programador *Ruby*
- Qué hago acá?

HELLO WORLD!

- Soy programador desde que tenía 10 años
- Hago programación *funcional* hace 5 años, en *Erlang*
- Soy *Director of Engineering* en **Inaka**
- Me dedico a diseñar y, a veces, construir servidores
- **No soy** un programador *Ruby*
- Qué hago acá?

HELLO WORLD!

- Soy programador desde que tenía 10 años
- Hago programación *funcional* hace 5 años, en *Erlang*
- Soy *Director of Engineering* en **Inaka**
- Me dedico a diseñar y, a veces, construir servidores
- **No soy** un programador *Ruby*
- Qué hago acá?

HELLO WORLD!

- Soy programador desde que tenía 10 años
- Hago programación *funcional* hace 5 años, en *Erlang*
- Soy *Director of Engineering* en **Inaka**
- Me dedico a diseñar y, a veces, construir servidores
- **No soy** un programador *Ruby*
- Qué hago acá? 🦎

RESUMEN

ESCENARIO

- Servidor con API tipo REST
- Clientes necesitan actualizaciones en *Real-Time*

SOLUCIÓN

- Se puede resolver con *Ruby*? **Sí**
- Existen otras soluciones? **Sí**

ERLANG

- Es **ideal** para este tipo de escenarios
- Trae *de fábrica*:
 - Concurrencia con procesos livianos
 - Comunicación por envío de mensajes
 - Supervisión de procesos
 - Escalabilidad horizontal *automática*

RESUMEN

ESCENARIO

- Servidor con API tipo REST
- Clientes necesitan actualizaciones en *Real-Time*

SOLUCIÓN

- Se puede resolver con *Ruby*? **Sí**
- Existen otras soluciones? **Sí**

ERLANG

- Es **ideal** para este tipo de escenarios
- Trae *de fábrica*:
 - Concurrencia con procesos livianos
 - Comunicación por envío de mensajes
 - Supervisión de procesos
 - Escalabilidad horizontal *automática*

RESUMEN

ESCENARIO

- Servidor con API tipo REST
- Clientes necesitan actualizaciones en *Real-Time*

SOLUCIÓN

- Se puede resolver con *Ruby*? **Sí**
- Existen otras soluciones? **Sí**

ERLANG

- Es **ideal** para este tipo de escenarios
- Trae *de fábrica*:
 - Concurrencia con procesos livianos
 - Comunicación por envío de mensajes
 - Supervisión de procesos
 - Escalabilidad horizontal *automática*

CONTENIDOS

En esta charla

- SSE
- Erlang *básico*
- Sumo DB *básico*
- Cowboy *básico*

Fuera de esta charla

- REST *avanzado*
- Erlang *avanzado* / OTP
- Sumo DB *avanzado*
- Elixir

CONTENIDOS

En esta charla

- SSE
- Erlang *básico*
- Sumo DB *básico*
- Cowboy *básico*

Fuera de esta charla

- REST *avanzado*
- Erlang *avanzado* / OTP
- Sumo DB *avanzado*
- Elixir

CONTENIDOS

En esta charla

- SSE
- Erlang *básico*
- Sumo DB *básico*
- Cowboy *básico*

Fuera de esta charla

- REST *avanzado*
- Erlang *avanzado* / OTP
- Sumo DB *avanzado*
- Elixir

CONTENIDOS

En esta charla

- SSE
- Erlang *básico*
- Sumo DB *básico*
- Cowboy *básico*

Fuera de esta charla

- REST *avanzado*
- Erlang *avanzado* / OTP
- Sumo DB *avanzado*
- Elixir

CONTENIDOS

En esta charla

- SSE
- Erlang *básico*
- Sumo DB *básico*
- Cowboy *básico*

Fuera de esta charla

- REST *avanzado*
- Erlang *avanzado* / OTP
- Sumo DB *avanzado*
- Elixir

CONTENIDOS

En esta charla

- SSE
- Erlang *básico*
- Sumo DB *básico*
- Cowboy *básico*

Fuera de esta charla

- REST *avanzado*
- Erlang *avanzado* / OTP
- Sumo DB *avanzado*
- Elixir

CONTENIDOS

En esta charla

- SSE
- Erlang *básico*
- Sumo DB *básico*
- Cowboy *básico*

Fuera de esta charla

- REST *avanzado*
- Erlang *avanzado* / OTP
- Sumo DB *avanzado*
- Elixir

CONTENIDOS

En esta charla

- SSE
- Erlang *básico*
- Sumo DB *básico*
- Cowboy *básico*

Fuera de esta charla

- REST *avanzado*
- Erlang *avanzado* / OTP
- Sumo DB *avanzado*
- Elixir

LA APLICACIÓN



Canillita

Provee una API sencilla con dos endpoints:

POST /NEWS

para publicar
noticias

GET /NEWS

para recibir
noticias

inaka

EJEMPLOS

POST /news

```
curl -vX POST http://localhost:4004/news \  
-H"Content-Type:application/json" \  
-d'{ "title": "Tu Primer Servidor Erlang con SSE",  
    "content": "@elbrujohalcon muestra su sistema para ..." }'
```

EJEMPLOS

POST /news

```
curl -vX POST http://localhost:4004/news \  
-H"Content-Type:application/json" \  
-d'{ "title": "Tu Primer Servidor Erlang con SSE",  
    "content": "@elbrujohalcon muestra su sistema para ..." }'
```

```
> POST /news HTTP/1.1  
> User-Agent: curl/7.30.0  
> Host: localhost:4004  
> Accept: */*  
> Content-Type:application/json  
> Content-Length: 50  
>  
< HTTP/1.1 204 No Content  
< connection: keep-alive  
< server: Cowboy  
< date: Fri, 08 Nov 2013 20:06:01 GMT  
< content-length: 0  
<
```

EJEMPLOS

GET /news

```
curl -vX GET http://localhost:4004/news
```

inaka

EJEMPLOS

GET /news

```
curl -vX GET http://localhost:4004/news
```

```
> GET /news HTTP/1.1
> User-Agent: curl/7.30.0
> Host: localhost:4004
> Accept: */*
>
```

```
< HTTP/1.1 200 OK
< transfer-encoding: chunked
< connection: keep-alive
< server: Cowboy
< date: Thu, 07 Nov 2013 14:31:10 GMT
< content-type: text/event-stream
<
```

```
event: old_news_flash
```

```
data: Nueva charla en la RubyConfAR!
```

```
data: La charla de @elbrujoahalcon esta por comenzar
```

inaka

EJEMPLOS

GET /news

```
< HTTP/1.1 200 OK
< transfer-encoding: chunked
< connection: keep-alive
< server: Cowboy
< date: Thu, 07 Nov 2013 14:31:10 GMT
< content-type: text/event-stream
<
event: old_news_flash
data: Nueva charla en la RubyConfAR!
data: La charla de @elbrujoalcon esta por comenzar

event: news_flash
data: Tu Primer Servidor Erlang con SSE
data: @elbrujoalcon muestra su sistema para ...
```


EJEMPLOS

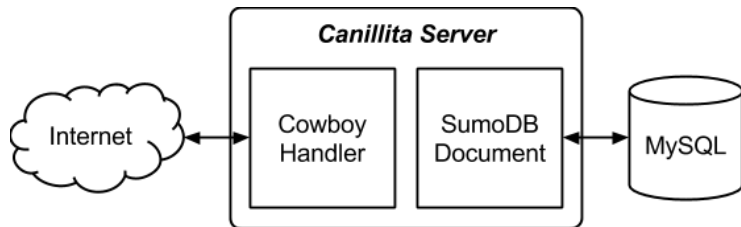
GET /news

```
< HTTP/1.1 200 OK
< transfer-encoding: chunked
< connection: keep-alive
< server: Cowboy
< date: Thu, 07 Nov 2013 14:31:10 GMT
< content-type: text/event-stream
<
event: old_news_flash
data: Nueva charla en la RubyConfAR!
data: La charla de @elbrujoalcon esta por comenzar

event: news_flash
data: Tu Primer Servidor Erlang con SSE
data: @elbrujoalcon muestra su sistema para ...

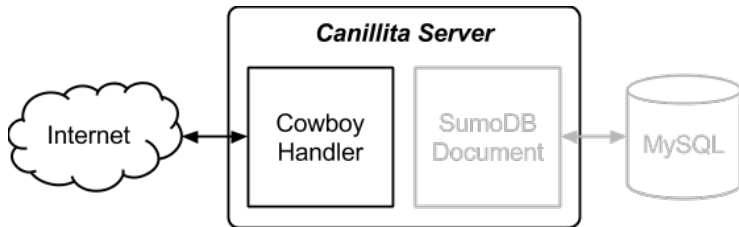
event: news_flash
data: Gran Concurrencia!
data: el publico observa esta diapositiva :P
```

ESQUEMA GENERAL



- canillita es una aplicación Erlang típica
 - Usa cowboy como web framework
 - Y sumo_db como motor de persistencia

ESQUEMA GENERAL



- canillita es una aplicación Erlang típica
- Usa cowboy como web framework
- Y sumo_db como motor de persistencia

SINTAXIS

Antes de continuar, un poco de sintaxis Erlang...

```
send_if_found(Key, Hash, Pid) ->  
    case proplists:get_value(Key, Hash) of  
        undefined -> do_nothing;  
        Value -> Pid ! {value, Value}  
    end.
```

SINTAXIS

```
send_if_found(Key, Hash, Pid) ->  
    case proplists:get_value(Key, Hash) of  
        undefined -> do_nothing;  
        Value -> Pid ! {value, Value}  
    end.
```

Ésta es la función `send_if_found`

SINTAXIS

```
send_if_found(Key, Hash, Pid) ->  
    case proplists:get_value(Key, Hash) of  
        undefined -> do_nothing;  
        Value -> Pid ! {value, Value}  
    end.
```

Tiene 3 argumentos, cada uno de ellos se asocia a una variable

SINTAXIS

```
send_if_found(Key, Hash, Pid) ->  
  case proplists:get_value(Key, Hash) of  
    undefined -> do_nothing;  
    Value -> Pid ! {value, Value}  
  end.
```

Ésta es una estructura de case. Cada cláusula comienza con
– > y culmina con ;

SINTAXIS

```
send_if_found(Key, Hash, Pid) ->  
    case proplists:get_value(Key, Hash) of  
        undefined -> do_nothing;  
        Value -> Pid ! {value, Value}  
    end.
```

En esta expresión se evalúa la función `get_value` del módulo `proplists`

SINTAXIS

```
send_if_found(Key, Hash, Pid) ->  
    case proplists:get_value(Key, Hash) of  
        undefined -> do_nothing;  
        Value -> Pid ! {value, Value}  
    end.
```

En esta cláusula se construye una *tupla* con el átomo `value` y el valor encontrado en el hash

SINTAXIS

```
send_if_found(Key, Hash, Pid) ->  
    case proplists:get_value(Key, Hash) of  
        undefined -> do_nothing;  
        Value -> Pid {value, Value}  
    end.
```

Y se envía esa tupla al proceso Pid

SINTAXIS

```
send_if_found(Key, Hash, Pid) ->  
    case proplists:get_value(Key, Hash) of  
        undefined -> do_nothing;  
        Value -> Pid ! {value, Value}  
    end.
```

Todas las cláusulas deben devolver algo, por eso ésta devuelve el átomo `do_nothing`

COWBOY HANDLER

canillita_news_handler

- Procesa requests HTTP
- Responde a POST /news utilizando un **REST handler**
- Responde a GET /news utilizando un **Loop handler**

COWBOY HANDLER

canillita_news_handler

- Procesa requests HTTP
- Responde a POST /news utilizando un **REST handler**
- Responde a GET /news utilizando un **Loop handler**

En la función init se determina qué handler utilizar:

```
init(_Transport, Req, _Opts) ->
  case cowboy_req:method(Req) of
    {<<"POST">>, _} ->
      {upgrade, protocol, cowboy_rest};
    {<<"GET">>, Req1} ->
      handle_get(Req1)
  end.
```

REST HANDLER

- `cowboy_rest` define múltiples funciones a implementar para procesar un request
- sólo se implementan las que se necesitan
- en nuestro caso:

REST HANDLER

- cowboy_rest define múltiples funciones a implementar para procesar un request
- sólo se implementan las que se necesitan
- en nuestro caso:

```
allowed_methods(Req, State) ->
  {[<<"POST">>], Req, State}.

content_types_accepted(Req, State) ->
  [{<<"application/json">>, handle_post}], Req, State}.

resource_exists(Req, State) ->
  {false, Req, State}.
```


REST HANDLER

- cowboy_rest define múltiples funciones a implementar para procesar un request
- sólo se implementan las que se necesitan
- en nuestro caso:

```
allowed_methods(Req, State) ->
  {[<<"POST">>], Req, State}.

content_types_accepted(Req, State) ->
  {[{<<"application/json">>, handle_post}], Req, State}.

resource_exists(Req, State) ->
  {false, Req, State}.
```

REST HANDLER

LA FUNCIÓN HANDLE_POST

```
handle_post(Req, State) ->
{ok, Body, Req1} = cowboy_req:body(Req),
case json_decode(Body) of
  {Params} ->
    Title =
      proplists:get_value(<<"title">>, Params, <<"News">>),
    Content =
      proplists:get_value(<<"content">>, Params, <<" ">>),
    NewsFlash = canillita_news:new(Title, Content),
    notify(NewsFlash),
    {true, Req1, State};
  {bad_json, Reason} ->
    {ok, Req2} =
      cowboy_req:reply(400, [], jiffy:encode(Reason), Req1),
    {halt, Req2, State}
end.
```

REST HANDLER

LA FUNCIÓN HANDLE_POST

```
handle_post(Req, State) ->
{ok, Body, Req1} = cowboy_req:body(Req),
case json_decode(Body) of
  {Params} ->
    Title =
      proplists:get_value(<<"title">>, Params, <<"News">>),
    Content =
      proplists:get_value(<<"content">>, Params, <<" ">>),
    NewsFlash = canillita_news:new(Title, Content),
    notify(NewsFlash),
    {true, Req1, State};
  {bad_json, Reason} ->
    {ok, Req2} =
      cowboy_req:reply(400, [], jiffy:encode(Reason), Req1),
    {halt, Req2, State}
end.
```

Obtiene el *body* del request

inaka

REST HANDLER

LA FUNCIÓN HANDLE_POST

```
handle_post(Req, State) ->
{ok, Body, Req1} = cowboy_req:body(Req),
case json_decode(Body) of
{Params} ->
    Title =
        proplists:get_value(<<"title">>, Params, <<"News">>),
    Content =
        proplists:get_value(<<"content">>, Params, <<" ">>),
    NewsFlash = canillita_news:new(Title, Content),
    notify(NewsFlash),
    {true, Req1, State};
{bad_json, Reason} ->
    {ok, Req2} =
        cowboy_req:reply(400, [], jiffy:encode(Reason), Req1),
    {halt, Req2, State}
end.
```

Lo parsea como JSON

inaka

REST HANDLER

LA FUNCIÓN HANDLE_POST

```
handle_post(Req, State) ->
  {ok, Body, Req1} = cowboy_req:body(Req),
  case json_decode(Body) of
    {Params} ->
      Title =
        proplists:get_value(<<"title">>, Params, <<"News">>),
      Content =
        proplists:get_value(<<"content">>, Params, <<" ">>),
      NewsFlash = canillita_news:new(Title, Content),
      notify(NewsFlash),
      {true, Req1, State};
    {bad_json, Reason} ->
      {ok, Req2} =
        cowboy_req:reply(400, [], jiffy:encode(Reason), Req1),
      {halt, Req2, State}
  end.
```

Extrae los campos title y content

inaka

REST HANDLER

LA FUNCIÓN HANDLE_POST

```
handle_post(Req, State) ->
{ok, Body, Req1} = cowboy_req:body(Req),
case json_decode(Body) of
  {Params} ->
    Title =
      proplists:get_value(<<"title">>, Params, <<"News">>),
    Content =
      proplists:get_value(<<"content">>, Params, <<" ">>),
    NewsFlash = canillita_news:new(Title, Content),
    notify(NewsFlash),
    {true, Req1, State};
  {bad_json, Reason} ->
    {ok, Req2} =
      cowboy_req:reply(400, [], jiffy:encode(Reason), Req1),
    {halt, Req2, State}
end.
```

Crea y almacena la noticia

inaka

REST HANDLER

LA FUNCIÓN HANDLE_POST

```
handle_post(Req, State) ->
{ok, Body, Req1} = cowboy_req:body(Req),
case json_decode(Body) of
  {Params} ->
    Title =
      proplists:get_value(<<"title">>, Params, <<"News">>),
    Content =
      proplists:get_value(<<"content">>, Params, <<" ">>),
    NewsFlash = canillita_news:new(Title, Content),
    notify(NewsFlash),
    {true, Req1, State};
  {bad_json, Reason} ->
    {ok, Req2} =
      cowboy_req:reply(400, [], jiffy:encode(Reason), Req1),
    {halt, Req2, State}
end.
```

La envía a quienes estén escuchando

inaka

REST HANDLER

LA FUNCIÓN HANDLE_POST

```
handle_post(Req, State) ->
{ok, Body, Req1} = cowboy_req:body(Req),
case json_decode(Body) of
  {Params} ->
    Title =
      proplists:get_value(<<"title">>, Params, <<"News">>),
    Content =
      proplists:get_value(<<"content">>, Params, <<" ">>),
    NewsFlash = canillita_news:new(Title, Content),
    notify(NewsFlash),
    {true, Req1, State};
  {bad_json, Reason} ->
    {ok, Req2} =
      cowboy_req:reply(400, [], jiffy:encode(Reason), Req1),
    {halt, Req2, State}
end.
```

Devuelve 204 No Content

inaka

LOOP HANDLER

- en nuestro handler, desde `init` estamos llamando a `handle_get`
- más allá de `init`, `cowboy_req` define una única función para implementar un *loop handler*: `info`
- es llamada cada vez que el proceso recibe un mensaje

LOOP HANDLER

LA FUNCIÓN HANDLE_GET

```
handle_get(Req) ->
  {ok, Req1} =
    cowboy_req:chunked_reply(
      200, [{"content-type", <<"text/event-stream">>}], Req),

  LatestNews = canillita_news:latest_news(),

  lists:foreach(
    fun(NewsFlash) ->
      send_flash(<<"old_news_flash">>, NewsFlash, Req1)
    end, LatestNews),

  pg2:join(canillita_listeners, self()),

  {loop, Req1, {}}.
```

LOOP HANDLER

LA FUNCIÓN HANDLE_GET

```
handle_get(Req) ->
{ok, Req1} =
  cowboy_req:chunked_reply(
    200, [{"content-type", <<"text/event-stream">>}], Req),

LatestNews = canillita_news:latest_news(),

lists:foreach(
  fun(NewsFlash) ->
    send_flash(<<"old_news_flash">>, NewsFlash, Req1)
  end, LatestNews),

pg2:join(canillita_listeners, self()),

{loop, Req1, {}}.
```

Setea el encoding y comienza a responder

inaka

LOOP HANDLER

LA FUNCIÓN HANDLE_GET

```
handle_get(Req) ->
{ok, Req1} =
  cowboy_req:chunked_reply(
    200, [{"content-type", <<"text/event-stream">>}], Req),

LatestNews = canillita_news:latest_news(),

lists:foreach(
  fun(NewsFlash) ->
    send_flash(<<"old_news_flash">>, NewsFlash, Req1)
  end, LatestNews),

pg2:join(canillita_listeners, self()),

{loop, Req1, {}}.
```

Obtiene las noticias de la base de datos

inaka

LOOP HANDLER

LA FUNCIÓN HANDLE_GET

```
handle_get(Req) ->
  {ok, Req1} =
    cowboy_req:chunked_reply(
      200, [{"content-type", <<"text/event-stream">>}], Req),

  LatestNews = canillita_news:latest_news(),

  lists:foreach(
    fun(NewsFlash) ->
      send_flash(<<"old_news_flash">>, NewsFlash, Req1)
    end, LatestNews),

  pg2:join(canillita_listeners, self()),

  {loop, Req1, {}}.
```

Envía cada una de ellas usando la función `send_flash`

inaka

LOOP HANDLER

LA FUNCIÓN HANDLE_GET

```
handle_get(Req) ->
  {ok, Req1} =
    cowboy_req:chunked_reply(
      200, [{"content-type", <<"text/event-stream">>}], Req),

  LatestNews = canillita_news:latest_news(),

  lists:foreach(
    fun(NewsFlash) ->
      send_flash(<<"old_news_flash">>, NewsFlash, Req1)
    end, LatestNews),

  pg2:join(canillita_listeners, self()),

  {loop, Req1, {}}.
```

Se subscribe para recibir futuras noticias

inaka

LOOP HANDLER

LA FUNCIÓN HANDLE_GET

```
handle_get(Req) ->
  {ok, Req1} =
    cowboy_req:chunked_reply(
      200, [{"content-type", <<"text/event-stream">>}], Req),

  LatestNews = canillita_news:latest_news(),

  lists:foreach(
    fun(NewsFlash) ->
      send_flash(<<"old_news_flash">>, NewsFlash, Req1)
    end, LatestNews),

  pg2:join(canillita_listeners, self()),

  {loop, Req1, {}}.
```

y se queda esperando mensajes, que llegarán a info

inaka

LOOP HANDLER

OTRAS FUNCIONES

```
info({news_flash, NewsFlash}, Req, State) ->
    send_flash(<<"news_flash">>, NewsFlash, Req),
    {loop, Req, State}.

send_flash(Event, NewsFlash, Req) ->
    Title = canillita_news:get_title(NewsFlash),
    Content = canillita_news:get_content(NewsFlash),
    Chunk = <<"event: ", Event/binary, "\n",
            "data: ", Title/binary, "\n",
            "data: ", Content/binary, "\n\n">>,
    cowboy_req:chunk(CHUNK, Req).

notify(NewsFlash) ->
    lists:foreach(fun(Listener) ->
        Listener ! {news_flash, NewsFlash}
    end, pg2:get_members(canillita_listeners)).
```


LOOP HANDLER

OTRAS FUNCIONES

```
info({news_flash, NewsFlash}, Req, State) ->
  send_flash(<<"news_flash">>, NewsFlash, Req),
  {loop, Req, State}.

send_flash(Event, NewsFlash, Req) ->
  Title = canillita_news:get_title(NewsFlash),
  Content = canillita_news:get_content(NewsFlash),
  Chunk = <<"event: ", Event/binary, "\n",
          "data: ", Title/binary, "\n",
          "data: ", Content/binary, "\n\n">>,
  cowboy_req:chunk(CHUNK, Req).

notify(NewsFlash) ->
  lists:foreach(fun(Listener) ->
    Listener ! {news_flash, NewsFlash}
  end, pg2:get_members(canillita_listeners)).
```

La función `info`, envía la noticia

LOOP HANDLER

OTRAS FUNCIONES

```
info({news_flash, NewsFlash}, Req, State) ->
    send_flash(<<"news_flash">>, NewsFlash, Req),
    {loop, Req, State}.

send_flash(Event, NewsFlash, Req) ->
    Title = canillita_news:get_title(NewsFlash),
    Content = canillita_news:get_content(NewsFlash),
    Chunk = <<"event: ", Event/binary, "\n",
            "data: ", Title/binary, "\n",
            "data: ", Content/binary, "\n\n">>,
    cowboy_req:chunk(CHUNK, Req).

notify(NewsFlash) ->
    lists:foreach(fun(Listener) ->
        Listener ! {news_flash, NewsFlash}
    end, pg2:get_members(canillita_listeners)).
```

y vuelve a esperar mensajes

inaka

LOOP HANDLER

OTRAS FUNCIONES

```
info({news_flash, NewsFlash}, Req, State) ->
    send_flash(<<"news_flash">>, NewsFlash, Req),
    {loop, Req, State}.

send_flash(Event, NewsFlash, Req) ->
    Title = canillita_news:get_title(NewsFlash),
    Content = canillita_news:get_content(NewsFlash),
    Chunk = <<"event: ", Event/binary, "\n",
            "data: ", Title/binary, "\n",
            "data: ", Content/binary, "\n\n">>,
    cowboy_req:chunk(CHUNK, Req).

notify(NewsFlash) ->
    lists:foreach(fun(Listener) ->
        Listener ! {news_flash, NewsFlash}
    end, pg2:get_members(canillita_listeners)).
```

La función `send_flash` extrae los campos de la noticia

LOOP HANDLER

OTRAS FUNCIONES

```
info({news_flash, NewsFlash}, Req, State) ->
    send_flash(<<"news_flash">>, NewsFlash, Req),
    {loop, Req, State}.

send_flash(Event, NewsFlash, Req) ->
    Title = canillita_news:get_title(NewsFlash),
    Content = canillita_news:get_content(NewsFlash),
    Chunk = <<"event: ", Event/binary, "\n",
            "data: ", Title/binary, "\n",
            "data: ", Content/binary, "\n\n">>,
    cowboy_req:chunk(Chunk, Req).

notify(NewsFlash) ->
    lists:foreach(fun(Listener) ->
        Listener ! {news_flash, NewsFlash}
    end, pg2:get_members(canillita_listeners)).
```

compone un bloque de texto a enviar

LOOP HANDLER

OTRAS FUNCIONES

```
info({news_flash, NewsFlash}, Req, State) ->
    send_flash(<<"news_flash">>, NewsFlash, Req),
    {loop, Req, State}.

send_flash(Event, NewsFlash, Req) ->
    Title = canillita_news:get_title(NewsFlash),
    Content = canillita_news:get_content(NewsFlash),
    Chunk = <<"event: ", Event/binary, "\n",
            "data: ", Title/binary, "\n",
            "data: ", Content/binary, "\n\n">>,
    cowboy_req:chunk(CHUNK, Req).

notify(NewsFlash) ->
    lists:foreach(fun(Listener) ->
        Listener ! {news_flash, NewsFlash}
    end, pg2:get_members(canillita_listeners)).
```

y lo envía al cliente

LOOP HANDLER

OTRAS FUNCIONES

```
info({news_flash, NewsFlash}, Req, State) ->
    send_flash(<<"news_flash">>, NewsFlash, Req),
    {loop, Req, State}.

send_flash(Event, NewsFlash, Req) ->
    Title = canillita_news:get_title(NewsFlash),
    Content = canillita_news:get_content(NewsFlash),
    Chunk = <<"event: ", Event/binary, "\n",
            "data: ", Title/binary, "\n",
            "data: ", Content/binary, "\n\n">>,
    cowboy_req:chunk(CHUNK, Req).

notify(NewsFlash) ->
    lists:foreach(fun(Listener) ->
        Listener ! {news_flash, NewsFlash}
    end, pg2:get_members(canillita_listeners)).
```

La función `notify` recorre la lista de subscriptos

LOOP HANDLER

OTRAS FUNCIONES

```
info({news_flash, NewsFlash}, Req, State) ->
    send_flash(<<"news_flash">>, NewsFlash, Req),
    {loop, Req, State}.

send_flash(Event, NewsFlash, Req) ->
    Title = canillita_news:get_title(NewsFlash),
    Content = canillita_news:get_content(NewsFlash),
    Chunk = <<"event: ", Event/binary, "\n",
            "data: ", Title/binary, "\n",
            "data: ", Content/binary, "\n\n">>,
    cowboy_req:chunk(CHUNK, Req).

notify(NewsFlash) ->
    lists:foreach(fun(Listener) ->
        Listener ! {news_flash, NewsFlash}
    end, pg2:get_members(canillita_listeners)).
```

y le envía un mensaje a cada uno

SUMODB DOCUMENT

canillita_news

- Implementa el behaviour `sumo_doc`
- Encapsula estado y comportamiento del modelo **News**
- Administra su persistencia a través de *SumoDB*

El behaviour `sumo_doc` define tres funciones a implementar:

- `sumo_schema`: definición del modelo
- `sumo_sleep`: traducción al formato de *SumoDB*
- `sumo_wakeup`: traducción a nuestro formato

SUMODB DOCUMENT

canillita_news

- Implementa el behaviour `sumo_doc`
- Encapsula estado y comportamiento del modelo **News**
- Administra su persistencia a través de *SumoDB*

El behaviour `sumo_doc` define tres funciones a implementar:

- `sumo_schema`: definición del modelo
- `sumo_sleep`: traducción al formato de *SumoDB*
- `sumo_wakeup`: traducción a nuestro formato

CANILLITA NEWS

BEHAVIOUR CALLBACKS

```
sumo_schema() ->
  sumo:new_schema(?MODULE,
    [ sumo:new_field(id, integer,
                        [id, not_null, auto_increment])
      , sumo:new_field(title, text, [not_null])
      , sumo:new_field(content, text, [not_null])
      , sumo:new_field(created_at, datetime, [not_null])
      , sumo:new_field(updated_at, datetime, [not_null])
    ]).

sumo_sleep(NewsFlash) -> NewsFlash.

sumo_wakeup(NewsFlash) -> NewsFlash.
```

CANILLITA NEWS

OTRAS FUNCIONES

```
new(Title, Content) ->
  Now = {datetime, calendar:universal_time()},
  NewsFlash = [ {title,      Title}
                 , {content,   Content}
                 , {created_at, Now}
                 , {updated_at, Now}],
  sumo:persist(canillita_news, NewsFlash).

get_title(NewsFlash) ->
  proplists:get_value(title, NewsFlash).
get_content(NewsFlash) ->
  proplists:get_value(content, NewsFlash).

latest_news() -> sumo:find_all(canillita_news).
```

CANILLITA NEWS

OTRAS FUNCIONES

```
new(Title, Content) ->  
  Now = {datetime, calendar:universal_time()},  
  NewsFlash = [ {title, Title},  
                 , {content, Content},  
                 , {created_at, Now},  
                 , {updated_at, Now}],  
  sumo:persist(canillita_news, NewsFlash).  
  
get_title(NewsFlash) ->  
  proplists:get_value(title, NewsFlash).  
get_content(NewsFlash) ->  
  proplists:get_value(content, NewsFlash).  
  
latest_news() -> sumo:find_all(canillita_news).
```

La función `new`, crea una entidad y la persiste

CANILLITA NEWS

OTRAS FUNCIONES

```
new(Title, Content) ->
  Now = {datetime, calendar:universal_time()},
  NewsFlash = [ {title,      Title}
                 , {content,   Content}
                 , {created_at, Now}
                 , {updated_at, Now}],
  sumo:persist(canillita_news, NewsFlash).

get_title(NewsFlash) ->
  proplists:get_value(title, NewsFlash).
get_content(NewsFlash) ->
  proplists:get_value(content, NewsFlash).

latest_news() -> sumo:find_all(canillita_news).
```

Las funciones `get_*` son simples proyectores

CANILLITA NEWS

OTRAS FUNCIONES

```
new(Title, Content) ->
  Now = {datetime, calendar:universal_time()},
  NewsFlash = [ {title,      Title}
                 , {content,   Content}
                 , {created_at, Now}
                 , {updated_at, Now}],
  sumo:persist(canillita_news, NewsFlash).

get_title(NewsFlash) ->
  proplists:get_value(title, NewsFlash).
get_content(NewsFlash) ->
  proplists:get_value(content, NewsFlash).

latest_news() -> sumo:find_all(canillita_news).
```

La función `latest_news` retorna todas las entidades

¿Y qué tal funciona?

TSUNG



- <http://tsung.erlang-projects.org/>
- Herramienta de medición de carga multi-protocolo distribuida
- Hecha en Erlang
- Puede utilizarse también para testear

TSUNG



- <http://tsung.erlang-projects.org/>
- Herramienta de medición de carga multi-protocolo distribuida
- Hecha en Erlang
- Puede utilizarse también para testear

TSUNG



- <http://tsung.erlang-projects.org/>
- Herramienta de medición de carga multi-protocolo distribuida
- Hecha en Erlang
- Puede utilizarse también para testear

ESCENARIO

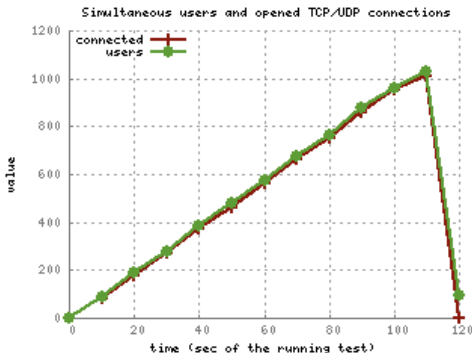
TEST

- Duración: 500 segundos
- Requests a POST /news: 1 por segundo
- Requests a GET /news: 50 por segundo

HARDWARE

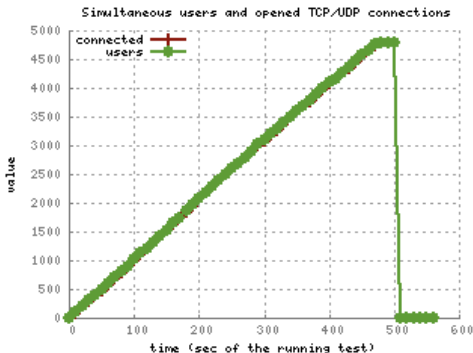
- MacBook PRO con OSX 10.9
- Procesador: 2.4 GHz Intel Core i5
- Memoria: 8 GB 1600 MHz DDR3

USUARIOS CONECTADOS



Un servidor *básico* en *Ruby* soportó cerca de **1000** conexiones simultáneas y luego la virtual murió

USUARIOS CONECTADOS



Un servidor *básico* en *Erlang* soportó más de **4500** conexiones simultáneas y luego comenzó a no responder a nuevas conexiones

Entonces...

¿**Erlang** es mejor que **Ruby** en este caso?

Yo qué sé
(Warren Sanchez)

... vale la pena intentar comprobarlo!!

Entonces...

¿**Erlang** es mejor que **Ruby** en este caso?

Yo qué sé
(Warren Sanchez)

... vale la pena intentar comprobarlo!!

Entonces...

¿**Erlang** es mejor que **Ruby** en este caso?

Yo qué sé
(Warren Sanchez)

... vale la pena intentar comprobarlo!!

MORALEJA

Construir **tu primer sistema** en *Erlang* puede ser algo **costoso** y hasta abrumador.

Sin embargo, para ciertos **escenarios habituales** no resulta difícil construir **las aplicaciones necesarias** e **integrarlas** con otros sistemas, aún sin dominar su arquitectura a fondo.

Y esas aplicaciones traen *gratis* todas las ventajas de *Erlang*, como **escalabilidad horizontal** y **supervisión de procesos**

Además, ése puede ser el **punto de partida**, para continuar aprendiendo el lenguaje paso a paso.

MORALEJA

Construir **tu primer sistema** en *Erlang* puede ser algo **costoso** y hasta abrumador.

Sin embargo, para ciertos **escenarios habituales** no resulta difícil construir **las aplicaciones necesarias** e **integrarlas** con otros sistemas, aún sin dominar su arquitectura a fondo.

Y esas aplicaciones traen *gratis* todas las ventajas de *Erlang*, como **escalabilidad horizontal** y **supervisión de procesos**. Además, ése puede ser el **punto de partida**, para continuar aprendiendo el lenguaje paso a paso.

MORALEJA

Construir **tu primer sistema** en *Erlang* puede ser algo **costoso** y hasta abrumador.

Sin embargo, para ciertos **escenarios habituales** no resulta difícil construir **las aplicaciones necesarias** e **integrarlas** con otros sistemas, aún sin dominar su arquitectura a fondo.

Y esas aplicaciones traen *gratis* todas las ventajas de *Erlang*, como **escalabilidad horizontal** y **supervisión de procesos**

Además, ése puede ser el **punto de partida**, para continuar aprendiendo el lenguaje paso a paso.

MORALEJA

Construir **tu primer sistema** en *Erlang* puede ser algo **costoso** y hasta abrumador.

Sin embargo, para ciertos **escenarios habituales** no resulta difícil construir **las aplicaciones necesarias** e **integrarlas** con otros sistemas, aún sin dominar su arquitectura a fondo.

Y esas aplicaciones traen *gratis* todas las ventajas de *Erlang*, como **escalabilidad horizontal** y **supervisión de procesos**

Además, ése puede ser el **punto de partida**, para continuar aprendiendo el lenguaje paso a paso.

MORALEJA

Construir **tu primer sistema** en *Erlang* puede ser algo **costoso** y hasta abrumador.

Sin embargo, para ciertos **escenarios habituales** no resulta difícil construir **las aplicaciones necesarias** e **integrarlas** con otros sistemas, aún sin dominar su arquitectura a fondo.

Y esas aplicaciones traen *gratis* todas las ventajas de *Erlang*, como **escalabilidad horizontal** y **supervisión de procesos**. Además, ése puede ser el **punto de partida**, para continuar aprendiendo el lenguaje paso a paso.

MATERIALES

SOBRE MÍ

- Soy **@elbrujoalcon** en Twitter
- Soy **elbrujoalcon** en GitHub

SOBRE INAKA

- Pueden ver nuestro sitio web: <http://inaka.net>
- Y nuestro Blog: <http://inaka.net/blog>

SOBRE CANILLITA

- El código está en GitHub: **inaka/canillita**
- Las slides también: **inaka/talks**

Muchas Gracias!