

# Alternatives in Error Handling

Dmitry Groshev

Erlang User Conference 2012



28.05.2012

# What's it all about

- ▶ error handling is easy in Erlang
- ▶ case/try/catch/happy path coding/let it crash
- ▶ error handling is hard in Erlang
- ▶ dark side of Erlang

# What's it all about

```
1 handle(Data) ->
2     case test1(Data) of
3         {ok, Data2} ->
4             case test2(Data2) of
5                 {ok, Data3} ->
6                     case test3(Data3) of
7                         {ok, Data4} ->
8                             do_something(Data4);
9                             {error, Err} ->
10                                 {error, {test3, Err}}
11                         end;
12                     {error, Err} ->
13                         {error, {test2, Err}}
14                 end;
15             {error, Err} ->
16                 {error, {test1, Err}}
17     end.
```

# Not a real solution

```
1 handle(Data) ->
2     case test1(Data) of
3         {ok, Data2} -> handle2(Data2);
4         {error, Err} -> {error, {test1, Err}}
5     end.
6
7 handle2(Data) ->
8     case test2(Data) of
9         {ok, Data2} -> handle3(Data2);
10        {error, Err} -> {error, {test2, Err}}
11    end.
```

- ▶ Still messy
- ▶ Nonsensical function names
- ▶ Lot of noise

- ▶ XML Schema
- ▶ JSON Schema
- ▶ Sheriff

Sheriff: <https://github.com/extend/sheriff>

```
1 -type colors() :: blue | red | green | yellow.  
2 -type paintable_object() :: #paintable_object{}.  
3  
4 paint(Color, Object) ->  
5     true = sheriff:check(Color, colors),  
6     true = sheriff:check(Object, paintable_object),  
7     do_paint(Color, Object).
```

## Still not good enough

```
1 -type colors() :: blue | red | green | yellow.  
2 paint(Color, Object) ->  
3     case sheriff:check(Color, colors) of  
4         true ->  
5             do_paint(Color, Object);  
6         false ->  
7             {error, badarg}  
8     end.
```



# Expressiveness problem

- ▶ IP: 183.234.123.93
- ▶ 4 numbers and dots?

# Expressiveness problem

- ▶ IP: 183.234.123.93
- ▶ 4 numbers and dots?
- ▶ IPv6: E3D7:0000:0000:0000:51F4:9BC8:C0A8:6420
- ▶ shortcut: E3D7::51F4:9BC8:C0A8:6420
- ▶ mixed: E3D7::51F4:9BC8:192.168.100.32
- ▶ regexpable after all?

# Expressiveness problem

- ▶ IP: 183.234.123.93
- ▶ 4 numbers and dots?
- ▶ IPv6: E3D7:0000:0000:0000:51F4:9BC8:C0A8:6420
- ▶ shortcut: E3D7::51F4:9BC8:C0A8:6420
- ▶ mixed: E3D7::51F4:9BC8:192.168.100.32
- ▶ regexable after all?
- ▶ if IP=127.0.0.1, port != 1234 (reserved for internal services)
- ▶ not so regexable

spec language  $\rightarrow$  programming language

# What's exactly a problem here?

- ▶ no return
- ▶ no implicit branching
- ▶ explicit branching is verbose

# Not really

Exceptions are an implicit branch

```
1 i_love_exceptions() ->  
2     {ok, Data} = get_data(),  
3     {ok, Params} = get_params(Data).
```

# Exceptions

```
1> list_to_integer(<<"abc">>).  
** exception error: bad argument  
   in function list_to_integer/1  
   called as list_to_integer(<<"abc">>)
```

# Try/catch

```
1 1> try list_to_integer(<<"abc">>)
2 1> catch error:badarg -> not_an_integer
3 1> end.
4 not_an_integer
```



## Not a real solution again

```
1 test() ->
2   try
3     A = list_to_integer(StringA),
4     B = list_to_integer(StringB),
5     {ok, {A, B}}
6   catch error:badarg -> {error, smth_is_wrong}
7   end.
```

Monads

# Comma

```
1 comma_is_not_so_simple() ->  
2     Foo = make_foo(),  
3     make_bar(Foo).
```

# Conditional comma

```
1 conditional_comma() ->
2   comma(make_foo(),
3         fun (Foo) -> comma(make_bar(Foo),
4                             fun (Bar) -> Bar end)
5         end).
```

Monad = comma + comma's expected datatype + return (value to comma's datatype of value)

Erlando: <https://github.com/rabbitmq/erlando>

# Erlando's magic

```
1 magic() ->
2     do([Monad | |
3         A <- make_foo(),
4         Bar <- make_bar(A),
5         Bar]).
```

# File example

```
1 write_file(Path, Data, Modes) ->
2   Modes1 = [binary, write | (Modes -- [binary, write])],
3   case make_binary(Data) of
4     Bin when is_binary(Bin) ->
5       case file:open(Path, Modes1) of
6         {ok, Hdl} ->
7           case file:write(Hdl, Bin) of
8             ok ->
9               case file:sync(Hdl) of
10                 ok ->
11                   file:close(Hdl);
12                   {error, _} = E ->
13                     file:close(Hdl),
14                     E
15                 end;
16                 {error, _} = E ->
17                   file:close(Hdl),
18                   E
19               end;
20               {error, _} = E -> E
21           end;
22           {error, _} = E -> E
23   end.
```

# File example with magic

```
1 write_file(Path, Data, Modes) ->
2   Modes1 = [binary, write |
3             (Modes -- [binary, write])],
4   do([error_m ||
5       Bin <- make_binary(Data),
6       Hdl <- file:open(Path, Modes1),
7       Result <- return(do([error_m ||
8                           file:write(Hdl, Bin),
9                           file:sync(Hdl)])),
10      file:close(Hdl),
11      Result])
```



# On the other hand

- ▶ performance overhead
- ▶ magic
- ▶ lack of supporting libraries (Erlang is not Haskell)

z\_validate: [https://github.com/si14/z\\_validate](https://github.com/si14/z_validate)

# About this library

- ▶ started at EUC 2011 hackathon
- ▶ intended to solve exactly this problem without excessive abstraction
- ▶ provides some shortcuts like `binary_to_integer` and `wrapper` to `lists:keyfind`

## First idea: tag values with error labels

$z\_value = \text{value (probably incorrect)} + \text{error label}$

# First idea: tag values with error labels

```
1 validate_some_input(Input) ->
2   try
3     WrappedInput = z_wrap(Input, error_in_foo),
4     Foo = z_bin_to_int(
5       z_proplist_get(MaybeInput, {foo})),
6     SmallFoo = z_int_in_range(Foo, {1, 10}),
7     z_return(z_unwrap(SmallFoo))
8   catch
9     ?Z_OK(Result)    -> {ok, Result};
10    ?Z_ERROR(Error)  -> {error, Error}
11  end.
12
```

# Composable!

```
1  z_extract_small_int(List, Key) ->  
2      z_int_in_range(  
3          z_bin_to_int(  
4              z_proplist_get(List, {Key}),  
5              {1, 10})).
```

## Second idea

```
1 -define(Z_CATCH(EXPR, ERROR),  
2     try  
3         EXPR  
4     catch  
5         _:_ -> throw({z_throw, {error, ERROR}})  
6     end).
```

Turned out to be practical

# Handler example

```
1  try
2      {Method, TaskName, VarSpecs} =
3          ?Z_CATCH({_, _, _} = lists:keyfind(Method, 1, TaskSpecs),
4                  bad_method),
5      TaskVarsRoute =
6          ?Z_CATCH([fetch_var(RouteVar, RouteVarType, Bindings)
7                  || {RouteVar, RouteVarType} <- RouteVars],
8                  bad_route),
9      TaskVars = [?Z_CATCH(fetch_var(Var, VarType, QSVals),
10                          {bad_var, Var})
11                  || {Var, VarType} <- VarSpecs],
12      z_return(rnbwdash_task:create(...))
13  catch
14      ?Z_OK(Task) -> form_reply(run_task(Task), Errors, Req@);
15      ?Z_ERROR(Err) -> form_error(Err, Req@)
16  end
```



# Pattern matching works well

```
1 {Method, TaskName, VarSpecs} =  
2   ?Z_CATCH({_, _, _} = lists:keyfind(Method, 1,  
3                                           TaskSpecs),  
4           bad_method)
```

# Plays well with lists

```
1 TaskVarsRoute =  
2   ?Z_CATCH([fetch_var(RouteVar, RouteVarType, Bindings)  
3             || {RouteVar, RouteVarType} <- RouteVars],  
4             bad_route)
```

# Push Z\_CATCH inside list comprehension

```
1 TaskVars = [?Z_CATCH(fetch_var(Var, VarType, QSVals),  
2                     {bad_var, Var})  
3             || {Var, VarType} <- VarSpecs]
```

# Error dispatch

```
1 error(bad_route) ->
2   {404, <<"Check path variables">>};
3 error(bad_method) ->
4   {405, <<"No such method in API">>};
5 error({bad_var, Var}) ->
6   {400, [<<"Check variable ">>, Var]}.
```

# Problems

- ▶ looks non-idiomatic
- ▶ Dialyzer isn't good at exceptions

# Dialyzer fail

```
1 good() ->
2   A = 1,
3   B = "string",
4   A + B.
5
6 bad() ->
7   A = 1,
8   B = "string",
9   C = try throw(B)
10      catch _:BThrown -> BThrown
11      end,
12   A + C.
```

# Performance tests: good data

```
1 -define(GOOD_DATA,  
2     [{login, <<"test_login">>},  
3      {password, <<"test_password">>},  
4      {session_id, <<"123">>},  
5      {good_user, <<"true">>},  
6      {some_other_id, <<"345">>},  
7      {yet_another_id, <<"56">>},  
8      {extra_data,  
9         term_to_binary({foo, bar, baz})}]]).
```

# Performance tests: bad data

```
1 -define(BAD_DATA1,  
2     [{login, <<"test_login">>},  
3      {session_id, <<"123">>}, %% no password  
4      {good_user, <<"true">>},  
5      {some_other_id, <<"345">>},  
6      {yet_another_id, <<"56">>},  
7      {extra_data,  
8         term_to_binary({foo, bar, baz})}]]).
```



# Performance tests: bad data

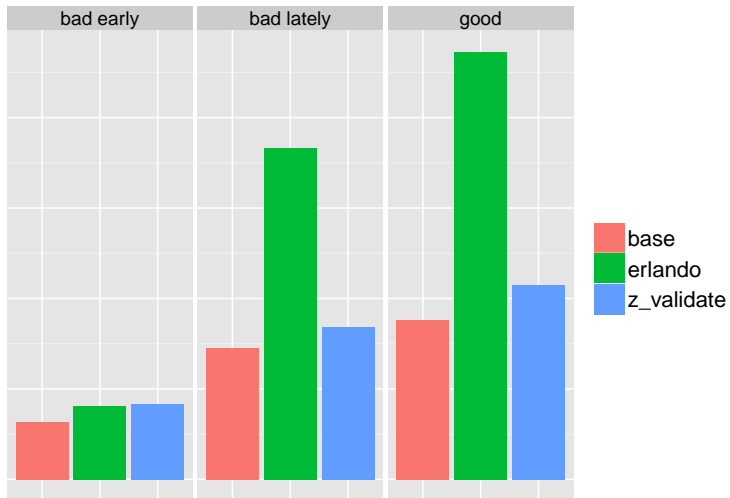
```
1 -define(BAD_DATA2,  
2     [{login, <<"test_login">>},  
3      {password, <<"test_password">>},  
4      {session_id, <<"123">>},  
5      {good_user, <<"true">>},  
6      {some_other_id, <<"345">>},  
7      {yet_another_id, <<"56abc">>}, %% bad ID  
8      {extra_data,  
9       term_to_binary({foo, bar, baz})}]]).
```

# Performance tests: baseline handler

```
1 test_handler_base(Data) ->
2   try
3     Login           = proplist_get(Data, login),
4     Password        = proplist_get(Data, password),
5     SessionBin      = proplist_get(Data, session_id),
6     Session         = bin_to_int(SessionBin),
7     GoodUserBin     = proplist_get(Data, good_user),
8     GoodUser        = bin_to_bool(GoodUserBin),
9     SomeOtherIdBin  = proplist_get(Data, some_other_id),
10    SomeOtherId      = bin_to_int(SomeOtherIdBin),
11    YetAnotherIdBin  = proplist_get(Data, yet_another_id),
12    YetAnotherId     = bin_to_int(YetAnotherIdBin),
13    ExtraDataBin     = proplist_get(Data, extra_data),
14    ExtraData        = bin_to_term(ExtraDataBin),
15
16    #request{login=Login, password=Password, ...}
17  catch A:B -> {A, B}
```

12 statements

# Performance comparison



## Questions?

Libraries:

<https://github.com/extend/sheriff>

<https://github.com/rabbitmq/erlando>

[https://github.com/si14/z\\_validate](https://github.com/si14/z_validate)

Slides:

<https://github.com/si14/euc-2012-slides>