
ExActor

downloads 32M downloads 32M

I don't maintain this project anymore. In hindsight, I don't think it was a good idea in the first place. I haven't been using ExActor myself for years, and I recommend sticking with regular GenServer instead :-)

Simplifies implementation of [GenServer](#) based processes in Elixir.

[ExActor](#) helps removing the boilerplate that typically occurs when using [GenServer](#) behaviour. In particular, [ExActor](#) can be useful in following situations:

- `start` function just packs all arguments into a tuple which it forwards to `init/1` via `GenServer.start`.
- Calls and casts interface functions just forward all arguments to the server process via `GenServer.call` and `GenServer.cast`.
- Process is registered and all interface functions rely on this property.
- Some `handle_*` functions don't need the state.
- All handlers need to specify timeout or hibernate.
- More liberal grouping of handler functions (you don't need to group calls and casts separately)

For other cases, you may need to use plain [GenServer](#) functions (which can be used together with [ExActor](#) macros). [ExActor](#) is not meant to fully replace [GenServer](#). It just tries to reduce boilerplate in most common cases.

If you're new to Elixir, Erlang, and OTP, and are not familiar on how [GenServer](#) works, I strongly suggest you learn about it first. It's really not that hard, and you can use Elixir docs as the starting point. It's also worth going through Mix/OTP getting started guide. Once you're familiar with [GenServer](#), you can consider using [ExActor](#) to reduce the boilerplate.

Online documentation is available [here](#).

The stable package is available on [hex](#).

Basic usage

Be sure to include a dependency in your `mix.exs`:

```
1 deps: [{:exactor, "~> 2.2.4", warn_missing: false}, ...]
```

[ExActor](#) is a compile-time dependency only. No need to add it into the list of dependent applications. All code transformations are performed at compile time. If you're using `exrm` to build OTP

releases, you may need to supply the `warn_missing: false` option to prevent warnings about a missing application dependency.

```
1 defmodule Calculator do
2   use ExActor.GenServer
3
4   defstart start_link, do: initial_state(0)
5
6   defcast inc(x), state: state, do: new_state(state + x)
7   defcast dec(x), state: state, do: new_state(state - x)
8
9   defcall get, state: state, do: reply(state)
10
11  defcast stop, do: stop_server(:normal)
12 end
```

This module be used in a typical fashion:

```
1 {:ok, calculator} = Calculator.start_link
2 Calculator.inc(calculator, 10)
3 Calculator.dec(calculator, 3)
4 Calculator.get(calculator)
5
6 Calculator.stop(calculator)
```

The module definition above is translated at compile-time into something like:

```
1 defmodule Calculator do
2   use GenServer
3
4   def start_link, do: GenServer.start_link(__MODULE__, nil)
5   def stop(pid), do: GenServer.cast(pid, :stop)
6
7   def inc(pid, x), do: GenServer.cast(pid, {:inc, x})
8   def dec(pid, x), do: GenServer.cast(pid, {:dec, x})
9   def get(pid), do: GenServer.call(pid, :get)
10
11  def init(_), do: {:ok, 0}
12
13  def handle_cast({:inc, x}, state), do: {:noreply, state + x}
14  def handle_cast({:dec, x}, state), do: {:noreply, state - x}
15  def handle_cast(:stop, state), do: {:stop, :normal, state}
16
17  def handle_call(:get, _, state), do: {:reply, state, state}
18 end
```

A bit more complex and feature rich example is presented here.

Predefines

To use `ExActor` macros, you must choose a predefine module and `use` it into your own module. A predefine is an `ExActor` module that provides some default implementations for `GenServer` callbacks.

Following predefines are currently provided:

- `ExActor.GenServer` - All `GenServer` callbacks are provided by `GenServer` from Elixir standard library.
- `ExActor.Strict` - All `GenServer` callbacks are provided. The default implementations for all except `code_change` and `terminate` will cause the server to be stopped.
- `ExActor.Tolerant` - All `GenServer` callbacks are provided. The default implementations ignore all messages without stopping the server.
- `ExActor.Empty` - No default implementation for `GenServer` callbacks are provided.

It is up to you to decide which predefine you want to use. See online docs for detailed description. You can also build your own predefine. Refer to the source code of the existing ones as a template.

Process registration

```
1 defmodule Calculator do
2   use ExActor.GenServer, export: :calculator
3
4   # you can also use via, and global
5   # use ExActor.GenServer, export: {:global, :calculator}
6   # use ExActor.GenServer, export: {:via, :gproc, :calculator}
7
8   ...
9 end
10
11 # all functions defined via defcall and defcast will take
12 # advantage of the export option
13 Calculator.start
14 Calculator.inc(5)
15 Calculator.get
```

Handling of return values

```
1 defstart start_link, do: initial_state(arg)
2
3 defcall foo, do: set_and_reply(new_state, response)
4 defcast bar, do: new_state(new_state)
```

```
5
6 defhandleinfo :stop, do: stop(normal)
7 defhandleinfo _, do: noreply
```

See here for detailed list.

Simplified initialization

```
1 defstart start_link(x, y, z) do
2   # Generates start_link function and `init/1` clause. The code runs in
    init/1 function.
3   initial_state(x + y + z)
4 end
```

By default, corresponding `GenServer` function is deduced from the function name, so you can use either `start_link` or `start`. If you want a custom function name, you need to provide explicit `:link` option:

```
1 defstart my_start(...), link: true do
2   ...
3 end
```

Dynamic start parameters

```
1 defmodule Calculator do
2   use ExActor.GenServer
3
4   # gen_server_opts: :runtime will add additional argument to the start
5   # function. This argument will be passed as options to the `GenServer
    `start
6   # function.
7   defstart start_link(x), gen_server_opts: :runtime, do: ...
8 end
9
10 # You can pass `name: :foo` due to `gen_server_opts: :runtime` option
    in the starter
11 Calculator.start_link(x, name: :foo)
12
13 # Or in the supervisor specification:
14 Supervisor.start_link(
15   [
16     worker(Calculator, [x, [name: :foo]]),
17     # ...
18   ]
19 )
```

Cluster support

```
1 defmodule Database do
2   use ExActor.GenServer, export: :database
3
4   defabcast store(key, value), do: ...
5   defmulticall get(key), do: ...
6 end
7
8 # called on all nodes
9 Database.store(key, value)
10 Database.get(key)
11
12 # called on specified nodes
13 Database.store(some_nodes, key, value)
14 Database.get(some_nodes, key)
```

Private interface functions

There are private versions available in form of `defstartp`, `defcallp`, `defcastp`, `defmulticallp`, and `defabcastp`. The only difference here is that interface functions are defined with `defp`. This can help you when you need to include some custom logic before or after the operation. See [here](#) for an example.

Pattern matching

```
1 defstart start_link(1), do:
2 defstart start_link(2), do:
3 defstart start_link(x), when: x < 5, do:
4
5 defcall a(1), do: ...
6 defcall a(2), do: ...
7 defcall a(x), state: 1, do: ...
8 defcall a(x), when: x > 1, do: ...
9 defcall a(_), do: ...
10
11 defhandleinfo :msg, state: {...}, when: ..., do: ...
```

All matches take place on both interface and handler functions.

Default arguments are also supported:

```
1 defcall inc(x \\ 1), ...
```

In this case, we'll end up with two `inc` interface functions, and a single `handle_call` function that matches on `{:inc, x}`.

Implementing just handlers

Can be useful to handle messages:

```
1 defhandleinfo :some_message, do:
2 defhandleinfo :another_message, state: ..., do:
```

Or to pattern match on the state:

```
1 # Body-less clause defines only the interface function
2 defcast inc
3
4 # Handle clauses pattern match on the state
5 defhandlecast inc, state: state, when: is_number(state),
6   do: new_state(state + 1)
7
8 defhandlecast inc, do: new_state(0)
```

Using from

```
1 defcall my_request(...), from: from do
2   ...
3   spawn_link(fn ->
4     ...
5     GenServer.reply(from, ...)
6   end)
7
8   noreply
9 end
```

Server-wide timeouts and hibernate

Timeout:

```
1 defstart ... do
2   # Instructs `ExActor` to include timeout in all responses made via
3   # responder
4   # macros, such as `new_state` or `noreply`. As the result, a `:
5   # timeout` message
6   # will be sent to the server after specified inactivity time.
7   timeout_after(:timer.seconds(10))
8 end
```

Hibernation:

```
1 defstart ... do
2   # Instructs `ExActor` to include `:hibernate` in all responses made
    via responder
3   # macros, such as `new_state` or `noreply`.
4   hibernate
5 end
```

Dynamic code generation friendliness

May be useful if you need to dynamically generate your requests. For example, if calls/casts simply delegate to some module, we could do something like:

```
1 defmodule DynActor do
2   use ExActor.GenServer
3
4   for op <- [:op1, :op2] do
5     defcall unquote(op), state: state do
6       SomeModule.unquote(op)(state)
7     end
8   end
9 end
```

A more involved example

In the following code, `ExActor` is used to implement a simple ETS based cache with basic cluster replication:

```
1 defmodule Cache do
2   use ExActor.GenServer
3
4   # Starter allows clients to specify cache name. Notice how this is
    used
5   # in `gen_server_opts` as a registered name of the server.
6   defstart start(cache_name, timeout_after \\ :infinity),
7     gen_server_opts: [name: cache_name]
8   do
9     # Specifies timeout which will be used in all handler responses
10    timeout_after(timeout_after)
11    :ets.new(cache_name, [:named_table, :set, :protected])
12    initial_state(cache_name)
13  end
14
15  # Looks up the cache in the client process
16  def get(cache_name, key) do
```

```

17     case :ets.lookup(cache_name, key) do
18         [{^key, value}] -> value
19         [] -> nil
20     end
21 end
22
23 # An example of a more complex interface function. A get attempt is
24 # made
25 # in the client process, and then we optionally issue a private call
26 # request.
27 def get_or_create(cache_name, key, fun) do
28     case get(cache_name, key) do
29         nil -> server_get_or_create(cache_name, key, fun)
30         existing -> existing
31     end
32 end
33
34 # Private call request used from `get_or_create`
35 defcallp server_get_or_create(key, fun), state: cache_name do
36     case get(cache_name, key) do
37         nil ->
38             new = fun.()
39             store(cache_name, key, new)
40             # Makes a distributed call to all other nodes
41             set(Node.list, cache_name, key, new)
42             new
43         existing -> existing
44     end
45     |> reply
46 end
47
48 # Distributed setter - stores to all nodes in the cluster
49 defmulticall set(key, value), state: cache_name do
50     store(cache_name, key, value)
51     reply(:ok)
52 end
53
54 defp store(cache_name, key, value) do
55     :ets.insert(cache_name, {key, value})
56 end
57
58 # Stops the server on timeout message
59 defhandleinfo :timeout, do: stop_server(:normal)
60 defhandleinfo _, do: noreply
61 end

```