THE BASIC SERVER IN ERLANG

The first common pattern I'll describe is one we've already used. When writing the event server, we had what could be called a *client-server model*. The event server would receive calls from the client, act on them and then reply to it if the protocol said to do so.

For this chapter, we'll use a very simple server, allowing us to focus on the essential properties of it. Here's thekitty_server:

```
%%%%% Naive version
-module(kitty_server).

-
export([start_link/0, order_cat/4, return_cat/2, close_shop
/1]).

-record(cat, {name, color=green, description}).

%%% Client API
start_link() -> spawn_link(fun init/0).

%% Synchronous call
order_cat(Pid, Name, Color, Description) ->
Ref = erlang:monitor(process, Pid),
Pid ! {self(), Ref, {order, Name, Color, Description}},
receive
{Ref, Cat} ->
erlang:demonitor(Ref, [flush]),
Cat;
{'DOWN', Ref, process, Pid, Reason} ->
erlang:error(Reason)
after 5000 ->
erlang:error(timeout)
end.

%% This call is asynchronous
return_cat(Pid, Cat = #cat{}) ->
Pid ! {return, Cat},
ok.

%% Synchronous call
close_shop(Pid) ->
Ref = erlang:monitor(process, Pid),
Pid ! {self(), Ref, terminate},
receive
{Ref, ok} ->
erlang:demonitor(Ref, [flush]),
ok;
{'DOWN', Ref, process, Pid, Reason} ->
erlang:error(Reason)
after 5000 ->
erlang:error(timeout)
end.

%%% Server functions
init() -> loop([]).

loop(Cats) ->
receive
{Pid, Ref, {order, Name, Color, Description}} ->
if Cats =:= [] ->
Pid ! {Ref, make_cat(Name, Color, Description)},
loop(Cats);
Cats =/= [] -> % got to empty the stock
Pid ! {Ref, hd(Cats)},
loop(tl(Cats))
end;
{return, Cat = #cat{}} ->
loop([Cat|Cats]);
{Pid, Ref, terminate} ->
Pid ! {Ref, ok},
terminate(Cats);
Unknown ->
%% do some logging here too
io:format("Unknown message: ~p~n", [Unknown]),
loop(Cats)
end.

%%% Private functions
make_cat(Name, Col, Desc) ->
#cat{name=Name, color=Col, description=Desc}.

terminate(Cats) ->
[io:format("~p was set free.~n",[C#cat.name]) || C <- Cats],
ok.
```

So this is a kitty server/store. The behavior is extremely simple: you describe a cat and you get that cat. If someone returns a cat, it's added to a list and is then automatically sent as the next order instead of what the client actually asked for (we're in this kitty store for the money, not smiles):

```
1> c(kitty_server).
{ok,kitty_server}
2> rr(kitty_server).
[cat]
3> Pid = kitty_server:start_link().
<0.57.0>
4> Cat1 = kitty_server:order_cat(Pid, carl, brown, "loves to
burn bridges").
#cat{name = carl,color = brown,
description = "loves to burn bridges"}
5> kitty_server:return_cat(Pid, Cat1).
ok
6> kitty_server:order_cat(Pid, jimmy, orange, "cuddly").
#cat{name = carl,color = brown,
description = "loves to burn bridges"}
7> kitty_server:order_cat(Pid, jimmy, orange, "cuddly").
#cat{name = jimmy,color = orange,description = "cuddly"}
8> kitty_server:return_cat(Pid, Cat1).
ok
9> kitty_server:close_shop(Pid).
carl was set free.
ok
10> kitty_server:close_shop(Pid).
** exception error: no such process or port
in function  kitty_server:close_shop/1
```

Looking back at the source code for the module, we can see patterns we've previously applied. The sections where we set monitors up and down, apply timers, receive data, use a main loop, handle the init function, etc. should all be familiar. It should be possible to abstract away these things we end up repeating all the time.

Let's first take a look at the client API. The first thing we can notice is that both synchronous calls are extremely similar. These are the calls that would likely go in abstraction libraries as mentioned in the previous section. For now, we'll just abstract these away as a single function in a new module which will hold all the generic parts of the kitty server:

```
-module(my_server).
-compile(export_all).

call(Pid, Msg) ->
Ref = erlang:monitor(process, Pid),
Pid ! {self(), Ref, Msg},
receive
{Ref, Reply} ->
erlang:demonitor(Ref, [flush]),
Reply;
{'DOWN', Ref, process, Pid, Reason} ->
erlang:error(Reason)
after 5000 ->
erlang:error(timeout)
end.
```

This takes a message and a PID, sticks them into in the function, then forwards the message for you in a safe manner. From now on, we can just substitute the message sending we do with a call to this function. So if we were to rewrite a new kitty server to be paired with the abstracted `my_server`, it could begin like this:

```
-module(kitty_server2).
-
export([start_link/0, order_cat/4, return_cat/2, close_shop
/1]).

-record(cat, {name, color=green, description}).

%%% Client API
start_link() -> spawn_link(fun init/0).

%% Synchronous call
order_cat(Pid, Name, Color, Description) ->
my_server:call(Pid, {order, Name, Color, Description}).

%% This call is asynchronous
return_cat(Pid, Cat = #cat{}) ->
Pid ! {return, Cat},
ok.

%% Synchronous call
close_shop(Pid) ->
my_server:call(Pid, terminate).
```

The next big generic chunk of code we have is not as obvious as the `call/2` function. Note that every process we've written so far has a loop where all the messages are pattern matched. This is a bit of a touchy part, but here we have to separate the pattern matching from the loop itself. One quick way to do it would be to add:

```
loop(Module, State) ->
receive
Message -> Module:handle(Message, State)
end.
```

And then the specific module can look like this:

```
handle(Message1, State) -> NewState1;
handle(Message2, State) -> NewState2;
...
handle(MessageN, State) -> NewStateN.
```

This is better. There are still ways to make it even cleaner. If you paid attention when reading the `kitty_server` module (and I hope you did!), you will have noticed we have a specific way to call synchronously and another one to call asynchronously. It would be pretty helpful if our generic server implementation could provide a clear way to know which kind of call is which.

In order to do this, we will need to match different kinds of messages in `my_server:loop/2`. This means we'll need to change the `call/2` function a little bit so synchronous calls are made obvious by adding the atom `sync` to the message on the function's second line:

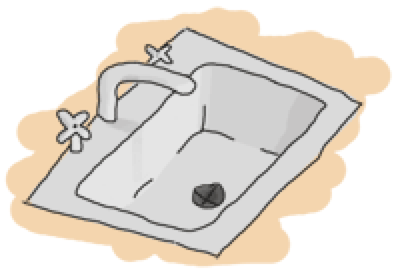
```
call(Pid, Msg) ->
Ref = erlang:monitor(process, Pid),
Pid ! {sync, self(), Ref, Msg},
receive
{Ref, Reply} ->
erlang:demonitor(Ref, [flush]),
Reply;
{'DOWN', Ref, process, Pid, Reason} ->
erlang:error(Reason)
after 5000 ->
erlang:error(timeout)
end.
```

We can now provide a new function for asynchronous calls. The function `cast/2` will handle this:

```
cast(Pid, Msg) ->
Pid ! {async, Msg},
ok.
```

With this done, the loop can now look like this:

```
loop(Module, State) ->
receive
{async, Msg} ->
loop(Module, Module:handle_cast(Msg, State));
{sync, Pid, Ref, Msg} ->
loop(Module, Module:handle_call(Msg, Pid, Ref, State))
end.
```



And then you could also add specific slots to handle messages that don't fit the sync/async concept (maybe they were sent by accident) or to have your debug functions and other stuff like hot code reloading in there.

One disappointing thing with the loop above is that the abstraction is leaking. The programmers who will use `my_server` will still need to know about references when sending synchronous messages and replying to them. That makes the abstraction useless. To use it, you still need to understand all the boring details. Here's a quick fix for it:

```
loop(Module, State) ->
receive
{async, Msg} ->
loop(Module, Module:handle_cast(Msg, State));
{sync, Pid, Ref, Msg} ->
loop(Module, Module:handle_call(Msg, {Pid, Ref}, State))
end.
```

By putting both variables *Pid* and *Ref* in a tuple, they can be passed as a single argument to the other function as a variable with a name like *From*. Then the user doesn't have to know anything about the variable's innards. Instead, we'll provide a function to send replies that should understand what *From* contains:

```
reply({Pid, Ref}, Reply) ->
Pid ! {Ref, Reply}.
```

What is left to do is specify the starter functions (`start`, `start_link` and `init`) that pass around the module names and whatnot. Once they're added, the module should look like this:

```
-module(my_server).
-export([start/2, start_link/2, call/2, cast/2, reply/2]).

%%% Public API
start(Module, InitialState) ->
spawn(fun() -> init(Module, InitialState) end).

start_link(Module, InitialState) ->
spawn_link(fun() -> init(Module, InitialState) end).

call(Pid, Msg) ->
Ref = erlang:monitor(process, Pid),
Pid ! {sync, self(), Ref, Msg},
receive
```

```
{Ref, Reply} ->
erlang:demonitor(Ref, [flush]),
Reply;
{'DOWN', Ref, process, Pid, Reason} ->
erlang:error(Reason)
after 5000 ->
erlang:error(timeout)
end.

cast(Pid, Msg) ->
Pid ! {async, Msg},
ok.

reply({Pid, Ref}, Reply) ->
Pid ! {Ref, Reply}.

%%% Private stuff
init(Module, InitialState) ->
loop(Module, Module:init(InitialState)).

loop(Module, State) ->
receive
{async, Msg} ->
loop(Module, Module:handle_cast(Msg, State));
{sync, Pid, Ref, Msg} ->
loop(Module, Module:handle_call(Msg, {Pid, Ref}, State))
end.
```

The next thing to do is reimplement the kitty server, now `kitty_server2` as a callback module that will respect the interface we defined for `my_server`. We'll keep the same interface as the previous implementation, except all the calls are now redirected to go through `my_server`:

```
-module(kitty_server2).

-
export([start_link/0, order_cat/4, return_cat/2, close_shop
/1]).
-export([init/1, handle_call/3, handle_cast/2]).

-record(cat, {name, color=green, description}).

%%% Client API
start_link() -> my_server:start_link(?MODULE, []).

%% Synchronous call
```

```
order_cat(Pid, Name, Color, Description) ->
my_server:call(Pid, {order, Name, Color, Description}).

%% This call is asynchronous
return_cat(Pid, Cat = #cat{}) ->
my_server:cast(Pid, {return, Cat}).

%% Synchronous call
close_shop(Pid) ->
my_server:call(Pid, terminate).
```

Note that I added a second `-export()` at the top of the module. Those are the functions `my_server` will need to call to make everything work:

```
%%% Server functions
init([]) -> []. %% no treatment of info here!

handle_call({order, Name, Color, Description}, From,
Cats) ->
if Cats =:= [] ->
my_server:reply(From, make_cat(Name, Color, Description)),
Cats;
Cats =/= [] ->
my_server:reply(From, hd(Cats)),
tl(Cats)
end;

handle_call(terminate, From, Cats) ->
my_server:reply(From, ok),
terminate(Cats).

handle_cast({return, Cat = #cat{}}, Cats) ->
[Cat|Cats].
```

And then what needs to be done is to re-add the private functions:

```
%%% Private functions
make_cat(Name, Col, Desc) ->
#cat{name=Name, color=Col, description=Desc}.

terminate(Cats) ->
[io:format("~p was set free.~n",[C#cat.name]) || C <- Cats],
exit(normal).
```

Just make sure to replace the `ok` we had before by `exit(normal)` in `terminate/1`, otherwise the server will keep going on.

The code should be compilable and testable, and run in exactly the same manner as it was before. The code is quite similar, but let's see what changed.

Source : http://learnyousomeerlang.com/what-is-otp