# ADDING SUPERVISION AND NAMESPACES

## Adding Supervision

In order to be a more stable application, we should write another 'restarter' as we did in the last chapter. Open up a file named sup.erl where our supervisor will be:

```
-module(sup).
-export([start/2, start_link/2, init/1, loop/1]).

start(Mod,Args) ->
spawn(?MODULE, init, [{Mod, Args}]).

start_link(Mod,Args) ->
spawn_link(?MODULE, init, [{Mod, Args}]).

init({Mod,Args}) ->
process_flag(trap_exit, true),
loop({Mod,start_link,Args}).

loop({M,F,A}) ->
Pid = apply(M,F,A),
receive
{'EXIT', _From, shutdown} ->
exit(shutdown); % will kill the child too
{'EXIT', Pid, Reason} ->
io:format("Process ~p exited for reason
~p~n",[Pid,Reason]),
loop({M,F,A})
end.
```

This is somewhat similar to the 'restarter', although this one is a tad more generic. It can take any module, as long as it has a `start_link` function. It will restart the process it watches indefinitely, unless the supervisor itself is terminated with a shutdown exit signal. Here it is in use:

```
1> c(evserv), c(sup).
{ok,sup}
2> SupPid = sup:start(evserv, []).
<0.43.0>
3> whereis(evserv).
<0.44.0>
4> exit(whereis(evserv), die).
true
Process <0.44.0> exited for reason die
5> exit(whereis(evserv), die).
```

```
Process <0.48.0> exited for reason die
true
6> exit(SupPid, shutdown).
true
7> whereis(evserv).
undefined
```

As you can see, killing the supervisor will also kill its child.

**Note:** We'll see much more advanced and flexible supervisors in the chapter about OTP supervisors. Those are the ones people are thinking of when they mention *supervision trees*. The supervisor demonstrated here is only the most basic form that exists and is not exactly fit for production environments compared to the real thing.

## Namespaces (or lack thereof)

Because Erlang has a flat module structure (there is no hierarchy), It is frequent for some applications to enter in conflict. One example of this is the frequently used `user` module that almost every project attempts to define at least once. This clashes with the `user` module shipped with Erlang. You can test for any clashes with the function `code:clash/0`.

Because of this, the common pattern is to prefix every module name with the name of your project. In this case, our reminder application's modules should be renamed to `reminder_evserv`, `reminder_sup` and `reminder_event`.

Some programmers then decide to add a module, named after the application itself, which wraps common calls that programmers could use when using their own application. Example calls could be functions such as starting the application with a supervisor, subscribing to the server, adding and cancelling events, etc.

It's important to be aware of other namespaces, too, such as registered names that must not clash, database tables, etc.

That's pretty much it for a very basic concurrent Erlang application. This one showed we could have a bunch of concurrent processes without thinking too hard about it: supervisors, clients, servers, processes used as timers (and we could have thousands of them), etc. No need to synchronize them, no locks, no real main loop. Message passing has made it simple to compartmentalize our application into a few modules with separated concerns and tasks.

The basic calls inside evserv.erl could now be used to construct clients that would allow to interact with the event server from somewhere outside of the Erlang VM and make the program truly useful.

Before doing that, though, I suggest you read up on the OTP framework. The next few chapters will cover some power comes from using it. It's a carefully crafted and well-engineered tool that any self-respecting Erlang programmer has to know.

Source : http://learnyousomeerlang.com/designing-a-concurrent-application