14

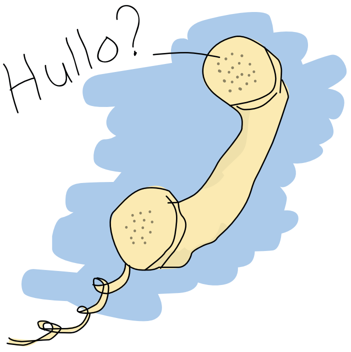
AN INTRODUCTION TO OTP

In this chapter, we'll get started with Erlang's OTP framework. OTP stands for Open Telecom Platform, though these days it's more about software that has the properties of telecom applications than telecom itself. If half of Erlang's greatness comes from its concurrency and distribution, and the other half comes from its error handling capabilities, then the OTP framework provides the third half.

During the previous chapters we've seen a few examples of common practices of how to write concurrent applications with the language's built-in facilities: links, monitors, servers, timeouts, trapping exits, and so on. There were a few "gotchas" involved in concurrent programming: Things must be done in a certain order, race conditions need to be avoided, and a process could die at any time. We also covered hot code loading, naming processes, adding supervisors, and other techniques.

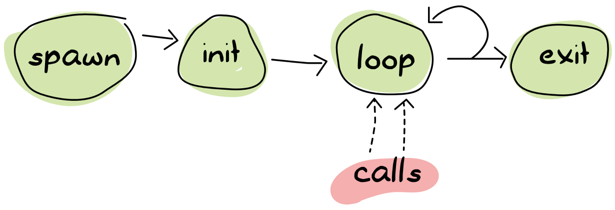
Doing all of this manually is time consuming and error prone. There are corner cases to be forgotten about and pits to fall in to. The OTP framework takes care of this by grouping these essential practices into a set of libraries that have been carefully engineered and battle hardened over the years. Every Erlang programmer should use them.

The OTP framework is also a set of modules and standards designed to help you build applications. Given that most Erlang programmers end up using OTP, most Erlang applications you'll encounter in the wild will tend to follow these standards.



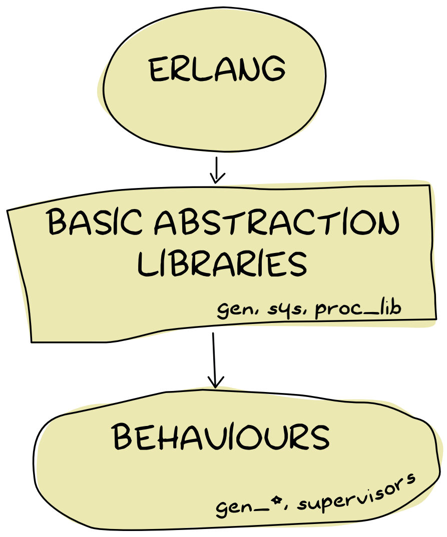
The Common Process, Abstracted

One of the things we've done many times in the previous process examples is divide everything in accordance to very specific tasks. In most processes, we had a function in charge of spawning the new process, a function in charge of giving the process its initial values, a main loop, and so on. These parts, as it turns out, are usually present in all concurrent programs you'll write, no matter what the process might be used for.



The engineers and computer scientists behind the OTP framework spotted these patterns and included them in a bunch of common libraries.

The OTP libraries are built with code that is equivalent to most of the abstractions we used (like using references to tag messages), with the advantages of being used for years in the field and built with far more caution than we used in our implementations. They contain functions to safely spawn and initialize processes, send messages to them in a fault-tolerant manner, and perform many other tasks. But you should rarely need to use these libraries yourself. The abstractions they contain are so basic and universal that a lot more interesting things, called *behaviors*, were built on top of them.



In this and the following chapters, we'll look at a few of the common uses of processes, and how they can be abstracted and then made generic. Then for each of these, we'll explore the corresponding implementation with the OTP framework's behaviors.

The Basic Server

The common pattern we'll explore in this chapter is one we've already used. For the event server we wrote in Chapter 13, we used a *client/server* model. The event server receives calls from the client, acts on them, and then replies to the client if the protocol says to do so.

Introducing the Kitty Server

For this chapter, we'll use a very simple server, allowing us to focus on its essential properties. Here's the kitty_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 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, and so on should be familiar. It should be possible to abstract away these things we end up repeating all the time. Let's start with the client API.

Generalizing Calls

The first thing to notice in the source code is that both synchronous calls are extremely similar. These are the calls that would likely go in abstraction libraries, as mentioned earlier. For now, we'll just abstract these away as a single function in a new module that 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 the function, and 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).
```

Generalizing the Server Loop

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 part is a bit touchy, but here we need to separate the pattern matching from the loop itself. One quick way to do it would be to add this:

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

And then the specific module would look like this:

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

This is better, but there are ways to make it even cleaner.

If you paid attention when reading or entering the `kitty_server` module (and I hope you did!), you will have noticed we have a specific way to call synchronously and another way 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.

To accomplish 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 bit so synchronous calls are made obvious. We'll do this by adding the atom `sync` to the message on the function's second line, as follows:

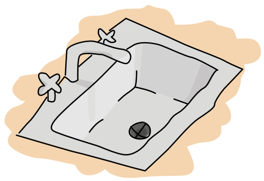
```
call(Pid, Msg) ->
    Ref = erlang:monitor(process, Pid),
    Pid ! {sync, self(), Ref, Msg},
    receive
        {Ref, Reply} ->
            erlang:demonitor(Ref, [flush]),
            Reply;
        {'DOWN', Ref, process, Pid, Reason} ->
            erlang:error(Reason)
    after 5000 ->
     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

Now the loop looks 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 about our loop 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:

```
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.
```

With both the variables `Pid` and `Ref` placed 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 need 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}.
```

Starter Functions

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.
```

Generalizing Kitty Server

Next, we need to re-implement 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 we added a second `-export()` at the top of the module. These 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 we need to reinsert 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 with `exit(normal)` in `terminate/1`; otherwise, the server will keep running.

You should be able to compile and test the code, and run it in exactly the same manner as the previous version. The code is quite similar, but let's see what has changed.

Specific vs. Generic

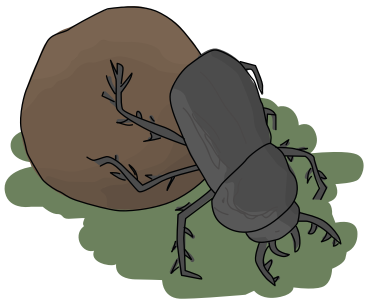
Our kitty server example demonstrates the core of OTP (conceptually speaking). This is what OTP really is all about: taking all the generic components, extracting them in libraries, making sure they work well, and then reusing that code when possible. Then all that's left to do is focus on the specific stuff—things that will always change from application to application.

Obviously, you don't benefit much by doing things that way with only the kitty server. It looks a bit like abstraction for abstraction's sake. If the application you needed to ship to a customer were nothing but the kitty server, then the first version might be fine. However, for larger applications, it might be worth the effort to separate generic parts of your code from the specific sections.

Imagine that we have some Erlang software running on a server. Our software has a few kitty servers running, a veterinary process (you send your broken kitties, and it returns them fixed), a kitty beauty salon, a server for pet food, and so on. Most of these can be implemented with a client/server pattern. As time passes, your complex system becomes full of different servers running around.

Adding servers adds complexity in terms of code, and also in terms of testing, maintenance, and understanding. Each implementation might be different, programmed in different styles by various people, and so on. However, if all these servers share the same common `my_server` abstraction, you substantially reduce that complexity. You understand the basic concept of the module instantly ("Oh, it's a server!"), and there's a single generic implementation of it to test and document. The rest of the effort can be put into each specific implementation of the server.

This means you reduce a lot of time tracking and solving bugs (just do it in one place for all servers). It also means that you reduce the number of bugs you introduce. If you were to rewrite the `my_server:call/3` or the process's main loop all the time, not only would it be more time-consuming, but chances of forgetting one step or another would skyrocket, and so would bugs. Fewer bugs mean fewer calls during the night to go fix something, which is definitely good for all of us (I bet you don't appreciate going to the office on days off to fix bugs either).



Another interesting outcome of separating the generic from the specific is that we instantly made it much easier to test our individual modules. If you wanted to unit test the old kitty server implementation, you would need to spawn one process per test, give it the right state, send your messages, and hope for the reply you expected. On the other hand, our second kitty server requires us to run the function calls over only the `handle_call/3` and `handle_cast/2` functions, and see what they output as a new state. There is no need to set up servers. Just pass the state in as a function parameter. Note that this also means the generic aspect of the server is much easier to test, given you can just implement very simple functions that do nothing other than let you focus on the behavior you want to observe.

A less obvious advantage of using common abstractions in this way is that if everyone uses the exact same backend for their processes, when someone optimizes that single backend to make it a little faster, every process using it out there will run a little faster, too. For this principle to work in practice, it's usually necessary to have a whole lot of people using the same abstractions and putting effort in them. Luckily for the Erlang community, that's what happens with the OTP framework.

In our kitty server modules, there are a bunch of things we haven't yet addressed: named processes, configuring the timeouts, adding debug information, what to do with unexpected messages, how to tie in hot code loading, handling specific errors, abstracting away the need to write most replies, handling most ways to shut down a server, making sure the server plays nice with supervisors, and more. Going over all of this is superfluous for this text, but it would be necessary in real products that need to be shipped. Again, you might see why doing all of this by yourself is a bit of a risky task. Luckily for you (and the people who will support your applications), the Erlang/OTP team managed to handle all of that with the `gen_server` behavior. `gen_server` is a bit like `my_server` on steroids, except it has years and years of testing and production use behind it.

Callback to the Future

Similar to the interface we started designing in this chapter, the OTP `gen_server` asks us to provide functions to deal with initialization and termination of processes, the handling of synchronous and asynchronous requests done through message passing, and a few other tasks.

The init Function

The first callback is an `init/1` function. It is similar to the one we used with `my_server` in that it is used to initialize the server's state and do all of these one-time tasks that the server will depend on. The function can return `{ok, State}`, `{ok, State, TimeOut}`, `{ok, State, hibernate}`, `{stop, Reason}`, or `ignore`.

The normal `{ok, State}` return value doesn't really need explaining other than that `State` will be passed directly to the main loop of the process as the state to keep later on. The `TimeOut` variable is meant to be added to the tuple whenever you need a deadline before which you expect the server to receive a message. If no message is received before the deadline, a special one (the atom `timeout`) is sent to the server, which should be handled with `handle_info/2` (described later in this chapter). This option is seldom used in production code, because you can't always know which messages you will receive, and any of them will be enough to reset the timer. It is usually better to use a function such as `erlang:start_timer/3` and handle things manually for better control.

On the other hand, if you do expect the process to take a long time before getting a reply and are worried about memory, you can add the `hibernate` atom to the tuple. Hibernation basically reduces the size of the process's state until it gets a message, at the cost of some processing power. If you are in doubt about using hibernation, you probably don't need it.

Returning `{stop, Reason}` should be done when something went wrong during the initialization.

A CLOSER LOOK AT HIBERNATION

There's a more technical definition of process hibernation, if you're interested. When the BIF `erlang:hibernate(M,F,A)` is called, the call stack for the currently running process is discarded (the function never returns). The garbage collection then kicks in, and what's left is one continuous heap that is shrunken to the size of the data in the process. This basically compacts all the data so the process takes less space.

Once the process receives a message, the function `M:F` with `A` as arguments is called, and the execution resumes.

NOTE *While `init/1` is running, execution is blocked in the process that spawned the server. This is because it is waiting for a "ready" message sent automatically by the `gen_server` module to make sure everything went fine.*

The handle_call Function

The function `handle_call/3` is used to work with synchronous messages. It takes three arguments: `Request`, `From`, and `State`. It's pretty similar to how we programmed our own `handle_call/3` in `my_server`. The biggest difference is how you reply to messages. In our own abstraction of a server, it was necessary to use `my_server:reply/2` to talk back to the process. In the case of `gen_server`, eight different return values are possible, taking the form of tuples:

```
{reply,Reply,NewState}
{reply,Reply,NewState,TimeOut}
{reply,Reply,NewState,hibernate}
{noreply,NewState}
{noreply,NewState,TimeOut}
{noreply,NewState,hibernate}
{stop,Reason,Reply,NewState}
{stop,Reason,NewState}
```

For all of these values, TimeOut and hibernate work the same way as for init/1. Whatever is in Reply will be sent back to whoever called the server in the first place.

Notice that there are three possible noreply options. When you use noreply, the generic part of the server will assume you're taking care of sending the reply back yourself. This can be done with gen_server:reply/2, which can be used in the same way as my_server:reply/2.

Most of the time, you'll need only the reply tuples. However, there are a few valid reasons to use noreply, such as when you want another process to send the reply for you, or when you want to send an acknowledgment ("Hey! I received the message!") but still process it afterward (without replying this time). If this is what you choose to do, it is absolutely necessary to use gen_server:reply/2; otherwise, the call will time out and cause a crash.

The handle_cast Function

The handle_cast/2 callback works a lot like the one in my_server. It takes the parameters Message and State and is used to handle asynchronous calls. You do whatever you want in there, in a manner quite similar to what's doable with handle_call/3. On the other hand, only tuples without replies are valid return values:

```
{noreply,NewState}
{noreply,NewState,TimeOut}
{noreply,NewState,hibernate}
{stop,Reason,NewState}
```

The handle_info Function

Earlier, I mentioned that our own server didn't really deal with messages that do not fit our interface. Well, handle_info/2 is the solution. It's very similar to handle_cast/2, and in fact, returns the same tuples. The difference is that this callback is there only for messages that were sent directly with the ! operator and special ones like init/1's timeout, monitors' notifications, and EXIT signals.

The terminate Function

The callback terminate/2 is called whenever one of the three handle_*something* functions returns a tuple of the form {stop, Reason, NewState} or {stop, Reason, Reply, NewState}. It takes two parameters, Reason and State, corresponding to the same values from the stop tuples.

The terminate/2 function will also be called when its parent (the process that spawned it) dies, if and only if the gen_server is trapping exits.

NOTE *If any reason other than `normal`, `shutdown`, or `{shutdown, Term}` is used when `terminate/2` is called, the OTP framework will see this as a failure and start logging the process's state, reason for failures, last messages received, and so on. This makes debugging easier, which might save your life quite a few times.*

This function is pretty much the direct opposite of `init/1`, so whatever was done in there should have its opposite in `terminate/2`. It's your server's janitor—the function in charge of locking the door after making sure everyone is gone. Of course, the function is helped by the VM itself, which should usually delete all ETS tables (see Chapter 25), close all sockets (see Chapter 23), and handle other tasks for you. Note that the return value of this function doesn't really matter, because the code stops executing after it has been called.

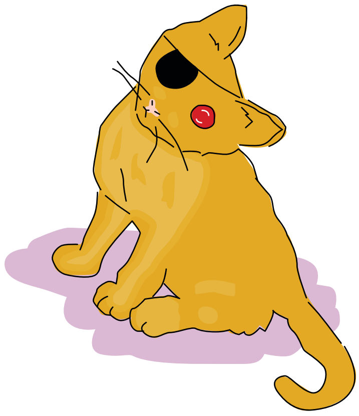
The code_change Function

The function `code_change/3` lets you upgrade code. It takes the form `code_change(PreviousVersion, State, Extra)`. Here, the variable `PreviousVersion` is either the version term itself (see Chapter 2 if you forgot what this is) in the case of an upgrade or `{down, Version}` in the case of a downgrade (just reloading older code). The `State` variable holds all of the current server state so you can convert it.

Imagine for a moment that we used an orddict to store all of our data. However, as time passes, the orddict becomes too slow, and we decide to replace it with a regular dict. In order to avoid the process crashing on the next function call, the conversion from one data structure to the other can be done in there, safely. All we need to do is return the new state with `{ok, NewState}`. We'll make use of this feature in Chapter 22, when we see relups as well as the `Extra` variable. We won't worry about these things for now.

So now we have all the callbacks defined. Don't worry if you're a little lost. The OTP framework is a bit circular sometimes; to understand part A of the framework, you need to understand part B, but then part B requires that you understand part A. The best way to get over that confusion is to actually implement a `gen_server`.

.BEAM Me Up, Scotty!



Now we'll build `kitty_gen_server`. It will be similar to `kitty_server2`, with only minimal API changes. First start a new module with the following lines in it:

```
-module(kitty_gen_server).
-behavior(gen_server).
```

NOTE *Both* `behavior` *and* `behaviour` *are accepted by the Erlang compiler.*

And try to compile it. You should get something like this:

```
1> c(kitty_gen_server).
./kitty_gen_server.erl:2: Warning: undefined callback function code_change/3
(behavior 'gen_server')
./kitty_gen_server.erl:2: Warning: undefined callback function handle_call/3
(behavior 'gen_server')
./kitty_gen_server.erl:2: Warning: undefined callback function handle_cast/2
(behavior 'gen_server')
./kitty_gen_server.erl:2: Warning: undefined callback function handle_info/2
(behavior 'gen_server')
./kitty_gen_server.erl:2: Warning: undefined callback function init/1
(behavior 'gen_server')
./kitty_gen_server.erl:2: Warning: undefined callback function terminate/2
(behavior 'gen_server')
{ok,kitty_gen_server}
```

The compilation worked, but there are warnings about missing callbacks. This is because of the gen_server behavior. A *behavior* is basically a way for a module to specify functions it expects another module to have. The behavior is the contract sealing the deal between the well-behaved generic part of the code and the specific, error-prone part of the code (yours).

DEFINING BEHAVIORS

Defining your own behaviors is really simple. You just need to export a function called `behavior_info/1`, implemented as follows:

```
-module(my_behavior).
-export([behavior_info/1]).

%% init/1, some_fun/0 and other/3 are now expected callbacks.
behavior_info(callbacks) -> [{init,1}, {some_fun, 0}, {other, 3}];
behavior_info(_) -> undefined.
```

And that's about it for behaviors. You can just use `-behavior(my_behavior).` in a module, implementing behaviors to get compiler warnings if you forgot a function.

The first function we had for our kitty server was `start_link/0`. This one can be changed to the following:

```
start_link() -> gen_server:start_link(?MODULE, [], []).
```

The first parameter is the callback module, the second one is a term to pass to `init/1`, and the third one is about debugging options for running servers. You could add a fourth parameter in the first position: `{local, *Name*}`, which is the name to register the server with. Note that while the previous version of the function simply returned a pid, this one instead returns {ok, Pid}.

The next functions are now as follows:

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

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

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

All of these calls are equivalent to those we had in my_server. Note that a third parameter can be passed to gen_server:call to give a timeout, in milliseconds. If you don't give a timeout to the function (or the atom infinity), the default is set to 5 seconds. If no reply is received before the time is up, the call crashes. This is an entirely arbitrary value, and many Erlang regulars will tell you that it should be changed to default to infinity. In my own experience, I often wanted replies to come in faster than 5 seconds, and having this timer force crashes has generally helped me diagnose more important problems.

Now we'll be able to add the gen_server callbacks. Table 14-1 shows the relationship we have between calls and callbacks.

Table 14-1: Relationship Between Calls and Callbacks

gen_server	YourModule
start/3-4	init/1
start_link/3-4	init/1
call/2-3	handle_call/3
cast/2	handle_cast/2

And then we have the other callbacks, which are more about special cases: handle_info/2, terminate/2, and code_change/3.

Let's begin by changing those we already have to fit the model: init/1, handle_call/3, and handle_cast/2.

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

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

```
handle_call(terminate, _From, Cats) ->
    {stop, normal, ok, Cats}.

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

Again, very little has changed here. In fact, the code is now shorter, thanks to smarter abstractions.

Now we get to the new callbacks. The first one is `handle_info/2`. Given this is a toy module and we have no logging system predefined, just outputting the unexpected messages will be enough.

```
handle_info(Msg, Cats) ->
    io:format("Unexpected message: ~p~n",[Msg]),
    {noreply, Cats}.
```

As a general rule of thumb, always log unexpected messages in `handle_cast/2` and `handle_info/2`. You might also want to log them in `handle_call/3`, but generally speaking, not replying to calls (coupled with the default 5 seconds timeout) is enough to achieve the same result.

The next one is the `terminate/2` callback. It will be very similar to the `terminate/1` private function we used earlier.

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

And here's the last callback, `code_change/3`:

```
code_change(_OldVsn, State, _Extra) ->
    %% No change planned. The function is there for the behavior,
    %% but will not be used.
    {ok, State}.
```

Just remember to keep in the `make_cat/3` private function:

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

And we can now try the brand-new code:

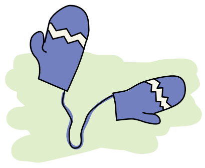
```
1> c(kitty_gen_server).
{ok,kitty_gen_server}
2> rr(kitty_gen_server).
[cat]
3> {ok, Pid} = kitty_gen_server:start_link().
{ok,<0.253.0>}
```

```
4> Pid ! <<"Test handle_info">>.
Unexpected message: <<"Test handle_info">>
<<"Test handle_info">>
5> Cat = kitty_gen_server:order_cat(Pid, "Cat Stevens",
5>       white, "not actually a cat").
#cat{name = "Cat Stevens",color = white,
     description = "not actually a cat"}
6> kitty_gen_server:return_cat(Pid, Cat).
ok
7> kitty_gen_server:order_cat(Pid, "Kitten Mittens",
7>   black, "look at them little paws!").
#cat{name = "Cat Stevens",color = white,
     description = "not actually a cat"}
```

Because we returned the Cat cat to the server, it's given back to us before we can order anything new. If we try again, we should get what we want:

```
8> kitty_gen_server:order_cat(Pid, "Kitten Mittens",
8>   black, "look at them little paws!").
#cat{name = "Kitten Mittens",color = black,
     description = "look at them little paws!"}
9> kitty_gen_server:return_cat(Pid, Cat).
ok
10> kitty_gen_server:close_shop(Pid).
"Cat Stevens" was set free.
ok
```

Hot damn, it works!



So what can we say about this generic adventure? Probably the same generic stuff as before: Separating the generic from the specific is a great idea on every point. Maintenance is simpler. Complexity is reduced. The code is safer, easier to test, and less prone to bugs. And if there are bugs, they are easier to fix.

Generic servers are only one of the many available abstractions, but they're certainly one of the most used ones. We'll explore more of these abstractions and behaviors in the next chapters.

15
RAGE AGAINST THE FINITE-STATE MACHINES

Finite-state machines are a central part of numerous implementations of important protocols in the industrial world. They allow programmers to represent complex procedures and sequences of events in a way that can be understood with ease.

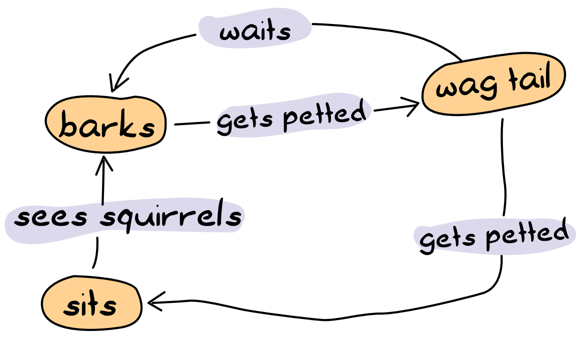
Although the most mathematically inclined readers might know finite-state machines under stricter mathematical definitions, the finite-state machines used in Erlang are more inspired by them than a direct implementation. A typical Erlang finite-state machine can be implemented as a process running a given set of functions (their states) and receiving messages (events) that force a state transition.

They were used so frequently in the telecom world that the OTP engineers ended up writing a behavior for them: `gen_fsm`.

This chapter introduces the concept of finite-state machines as used in the Erlang world and its OTP counterpart. We'll experiment with them by designing a fully asynchronous, message-based protocol for a client-to-client trading system that could be added to a video game.

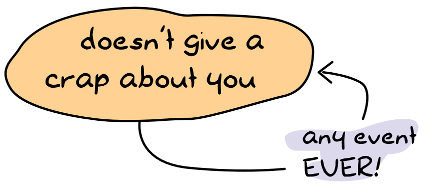
What Is a Finite-State Machine?

A finite-state machine (FSM) is not really a machine, but it does have a finite number of states. I've always found FSMs easier to understand with graphs and diagrams. For example, the following is a simplistic diagram for a (very dumb) dog as a state machine:



Here, the dog has three states: sitting, barking, or wagging his tail. Different events or inputs may force the dog to change his state. If a dog is calmly sitting and sees a squirrel, he will start barking and won't stop until you pet him again. However, if the dog is sitting and you pet him, we have no idea what might happen. In the Erlang world, the dog could crash (and eventually be restarted by his supervisor). In the real world, restarting your dog would be pretty unusual (and a little freaky), though that would mean the dog could come back after being run over by a car, so it's not all bad.

Here's a cat's state diagram for comparison:



This cat has a single state, and no event can ever change it.

Implementing the cat state machine in Erlang is a fun and simple task:

```
-module(cat_fsm).
-export([start/0, event/2]).

start() ->
    spawn(fun() -> dont_give_crap() end).

event(Pid, Event) ->
    Ref = make_ref(), % won't care for monitors here
    Pid ! {self(), Ref, Event},
    receive
        {Ref, Msg} -> {ok, Msg}
    after 5000 ->
        {error, timeout}
    end.

dont_give_crap() ->
    receive
        {Pid, Ref, _Msg} -> Pid ! {Ref, meh};
        _ -> ok
    end,
    io:format("Switching to 'dont_give_crap' state~n"),
    dont_give_crap().
```

We can try the module to see that the cat really never gives a crap:

```
1> c(cat_fsm).
{ok,cat_fsm}
2> Cat = cat_fsm:start().
<0.67.0>
3> cat_fsm:event(Cat, pet).
Switching to 'dont_give_crap' state
{ok,meh}
4> cat_fsm:event(Cat, love).
Switching to 'dont_give_crap' state
{ok,meh}
5> cat_fsm:event(Cat, cherish).
Switching to 'dont_give_crap' state
{ok,meh}
```

The same can be done for the dog FSM, except more states are available:

```
-module(dog_fsm).
-export([start/0, squirrel/1, pet/1]).

start() -> spawn(fun() -> bark() end).

squirrel(Pid) -> Pid ! squirrel.

pet(Pid) -> Pid ! pet.

bark() ->
    io:format("Dog says: BARK! BARK!~n"),
    receive
        pet ->
            wag_tail();
        _ ->
            io:format("Dog is confused~n"),
            bark()
    after 2000 ->
        bark()
    end.

wag_tail() ->
    io:format("Dog wags its tail~n"),
    receive
        pet ->
            sit();
        _ ->
            io:format("Dog is confused~n"),
            wag_tail()
    after 30000 ->
        bark()
    end.

sit() ->
    io:format("Dog is sitting. Gooooood boy!~n"),
    receive
        squirrel ->
            bark();
        _ ->
            io:format("Dog is confused~n"),
            sit()
    end.
```

It should be relatively simple to match each of the states and transitions to those shown in the dog's state diagram. Here's the FSM in use:

```
6> c(dog_fsm).
{ok,dog_fsm}
7> Pid = dog_fsm:start().
Dog says: BARK! BARK!
<0.46.0>
Dog says: BARK! BARK!
Dog says: BARK! BARK!
Dog says: BARK! BARK!
8> dog_fsm:pet(Pid).
pet
Dog wags its tail
9> dog_fsm:pet(Pid).
Dog is sitting. Gooooood boy!
pet
10> dog_fsm:pet(Pid).
Dog is confused
pet
Dog is sitting. Gooooood boy!
11> dog_fsm:squirrel(Pid).
Dog says: BARK! BARK!
squirrel
Dog says: BARK! BARK!
12> dog_fsm:pet(Pid).
Dog wags its tail
pet
❶ 13> %% wait 30 seconds
Dog says: BARK! BARK!
Dog says: BARK! BARK!
Dog says: BARK! BARK!
```

```
13> dog_fsm:pet(Pid).
Dog wags its tail
pet
14> dog_fsm:pet(Pid).
Dog is sitting. Gooooood boy!
pet
```

You can follow along with the schema if you want (I usually do, since it helps me to make sure that nothing is wrong). Note that at ❶, the command entered is strictly a comment intended for the reader, although the Erlang shell deals with it fine.

That's really the core of FSMs implemented as Erlang processes. There are things that could have been done differently. We could have passed state in the arguments of the state functions in a way similar to what we do with a server's main loop. We could also have added `init` and `terminate` functions, handled code updates, and so on.

A difference between the dog and cat FSMs is that the cat's events are *synchronous* and the dog's events are *asynchronous*. In a real FSM, both could be used in a mixed manner, but I went for the simplest representation out of pure, untapped laziness.

There is also another event form the examples do not show: global events that can happen in any state. One example of such an event could be when the dog gets a sniff of food. Once the "smell food" event is triggered, no matter which state the dog is in, he will go looking for the source of food.

We won't spend too much time implementing all of this in our "written-on-a-napkin" FSM. Instead, we'll move directly to the `gen_fsm` behavior.

Generic Finite-State Machines

The `gen_fsm` behavior is somewhat similar to `gen_server` in that it is a specialized version of that behavior. The biggest difference is that rather than handling *calls* and *casts*, we're handling *synchronous* and *asynchronous* events. Similar to our dog and cat examples, each state is represented by a function. Here, we'll go through the callbacks our modules need to implement in order to work.

The init Function

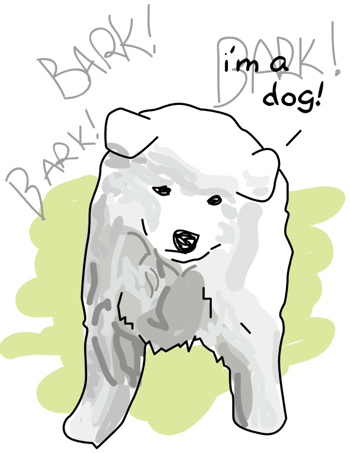
The `init` function for FSMs is the same `init/1` as used for generic servers, except the return values accepted are `{ok, StateName, Data}`, `{ok, StateName, Data, Timeout}`, `{ok, StateName, Data, hibernate}`, and `{stop, Reason}`. The `stop` tuple works in the same manner as for `gen_server`, and both `hibernate` and `Timeout` keep the same semantics.

What's new here is the `StateName` variable. `StateName` is an atom and represents the next callback function to be called. For our dog, this would be the `bark` state.

The StateName Function

The functions StateName/2 and StateName/3 are placeholder names, and you decide what they will be. Let's suppose the init/1 function returns the tuple {ok, sitting, dog}. This means the FSM will be in the sitting state. This is not the same kind of state as we have seen with gen_server, but more like the sit, bark, and wag_tail states of our dog FSM. These states dictate a context in which you handle a given event.

As an example, consider someone calling you on your phone. If you're in the state "sleeping on a Saturday morning," your reaction might be to yell at the phone. If your state is "waiting for a job interview," chances are you'll pick up the phone and answer politely. On the other hand, if you're in the state "dead," then I am surprised you can even read this text at all.



In our FSM, the init/1 function said we should be in the sitting state. Whenever the gen_fsm process receives an event, either the function sitting/2 or sitting/3 will be called. The sitting/2 function is called for asynchronous events, and sitting/3 is called for synchronous events.

The arguments for sitting/2 (or generally StateName/2) are Event, the actual message sent as an event, and StateData, the data that was carried over the calls. The sitting/2 function can then return the tuples {next_state, NextStateName, NewStateData}, {next_state, NextStateName, NewStateData, Timeout}, {next_state, NextStateName, hibernate}, and {stop, Reason, NewStateData}.

The arguments for sitting/3 are similar, except there is a From variable in between Event and StateData. The From variable is used in exactly the same way as it is for gen_server, including gen_fsm:reply/2. The StateName/3 functions can return the following tuples:

```
{reply, Reply, NextStateName, NewStateData}
{reply, Reply, NextStateName, NewStateData, Timeout}
{reply, Reply, NextStateName, NewStateData, hibernate}

{next_state, NextStateName, NewStateData}
{next_state, NextStateName, NewStateData, Timeout}
{next_state, NextStateName, NewStateData, hibernate}

{stop, Reason, Reply, NewStateData}
{stop, Reason, NewStateData}
```

Note that there's no limit on how many of these functions you can have, as long as they are exported. The atoms returned as NextStateName in the tuples will determine whether or not the function will be called.

The handle_event Function

Earlier, I mentioned global events, which trigger a specific reaction no matter what state we're in (the dog smelling food will drop whatever he is doing and look for food). For these events that should be treated the same way in every state, the `handle_event/3` callback is what you want. The function takes arguments similar to `StateName/2`, with the exception that it accepts a `StateName` variable in between them (`handle_event(Event, StateName, Data)`), telling you what the state was when the event was received. It returns the same values as `StateName/2`.

The handle_sync_event Function

The `handle_sync_event/4` callback is to `StateName/3` what `handle_event/2` is to `StateName/2`. It handles synchronous global events, takes the same parameters, and returns the same kind of tuples as `StateName/3`.

Now might be a good time to explain how we know whether an event is global or if it's meant to be sent to a specific state. To determine this, we can look at the function used to send an event to the FSM. Asynchronous events aimed at any `StateName/2` function are sent with `gen_fsm:send_event/2`, and synchronous events to be picked up by `StateName/3` are sent with `gen_fsm:sync_send_event/2-3` (the optional third argument is the timeout).

The two equivalent functions for global events are `gen_fsm:send_all_state_event/2` and `gen_fsm:sync_send_all_state_event/2-3` (quite a long name).

The code_change and terminate Functions

The `code_change` function works exactly the same as it does for `gen_server`, except that it takes an extra state parameter when called, such as `code_change(OldVersion, StateName, Data, Extra)`, and returns a tuple of the form `{ok, NextStateName, NewStateData}`.

Similarly, `terminate` acts a bit like what we have for generic servers. `terminate(Reason, StateName, Data)` should do the opposite of `init/1`.

A Trading System Specification

It's time to put all of this information about FSMs into practice. Many Erlang tutorials about FSMs use examples containing telephone switches and the like. It's my guess that most programmers will rarely need to deal with telephone switches for state machines. Here, we'll look at an example that is more fitting for many developers. We'll design and implement an item trading system for a fictional video game.

The design I have picked is somewhat challenging. Rather than using a central broker through which players route items and confirmations (which, frankly, would be easier), we're going to implement a server where both players speak to each other directly (which has the advantage of being easily distributable).

Show Me Your Moves

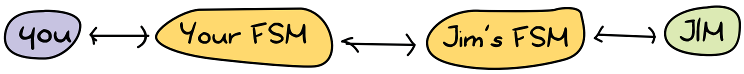
To begin, we should define the actions that can be taken by our players when trading. The first is asking for a trade to be set up. The other user should also be able to accept that trade. We won't give the players the right to deny a trade, though, because we want to keep things simple. It would be easy to add that feature later.

Once the trade is set up, our users should be able to negotiate with each other. This means they should be able to make offers and also to retract those offers. When both players are satisfied with the offer, they can declare themselves as ready to finalize the trade. The data should then be saved somewhere on both sides. At any point in time, the players should be able to cancel the whole trade. Some pleb could be offering only items deemed unworthy to the other party (who might be very busy), and so it should be possible to backhand that player with a well-deserved cancellation.

In short, the following actions should be possible:

- Ask for a trade.
- Accept a trade.
- Offer items.
- Retract an offer.
- Declare self as ready.
- Brutally cancel the trade.

When each of these actions is taken, the other player's FSM should be made aware of it. This makes sense, because when you're playing the game and Jim tells his FSM to send an item to you, your FSM must be made aware of it. This means both players can talk to their own FSM, which will talk to the other's FSM. This gives us something a bit like this:

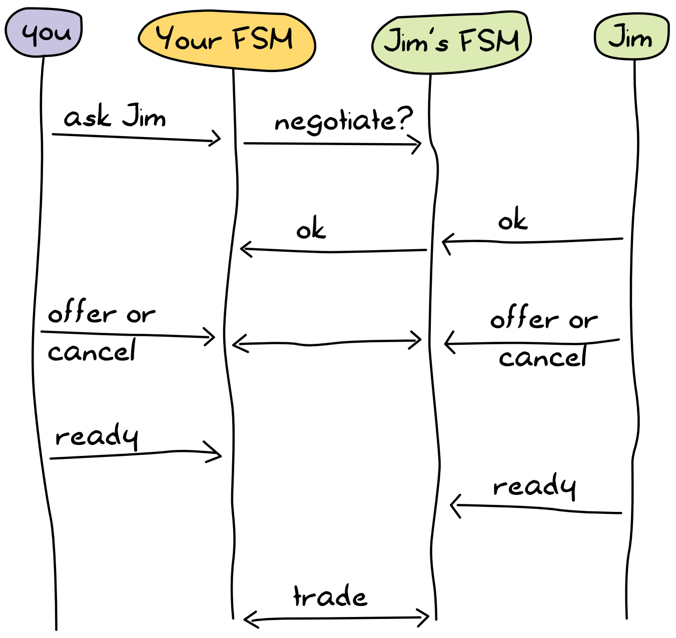


The first thing to notice when we have two identical processes communicating with each other is that we need to avoid synchronous calls as much as possible. If Jim's FSM sends a message to your FSM and then waits for its reply, while at the same time, your FSM sends a message over to Jim's FSM and waits for its own specific reply, both end up waiting for the other without ever replying. This effectively freezes both FSMs. We have a *deadlock*.

One solution is to wait for a timeout and then move on, but then there will be leftover messages in both processes' mailboxes, and the protocol will be messed up. This certainly is a can of worms, and so we want to avoid it.

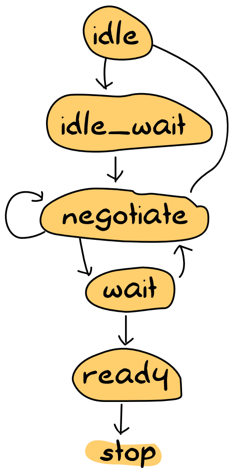
The simplest way to handle this is to go fully asynchronous. Note that Jim might still make a synchronous call to his own FSM; there's no risk here because the FSM won't need to call Jim, and so no deadlock can occur between them.

When two of these FSMs communicate together, the whole exchange might look a bit like this:



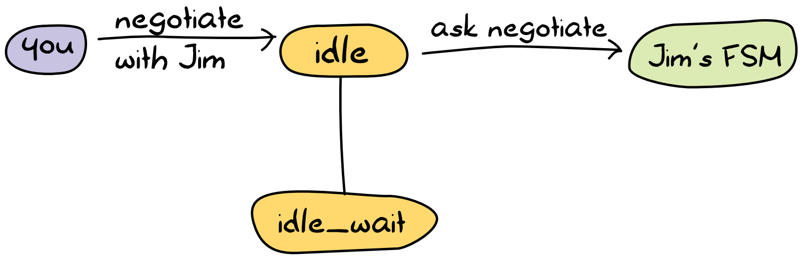
Both FSMs are in an idle state. When you ask Jim to trade, Jim needs to accept before things move on. Then both you and Jim can offer items or withdraw them. When both players declare themselves ready, the trade can take place. This is a simplified version of all that can happen. We'll consider all possible cases as we implement the trading system.

Defining the State Diagrams and Transitions

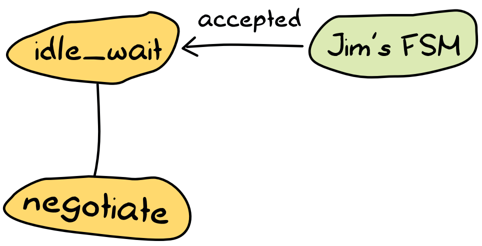


Here comes the tough part: defining the state diagrams and how state transitions happen. Usually, a good bit of thinking goes into this, because you need to consider all the small things that could go wrong (and some things might go wrong even after you've reviewed the definitions many times). Here's the one I decided to implement:

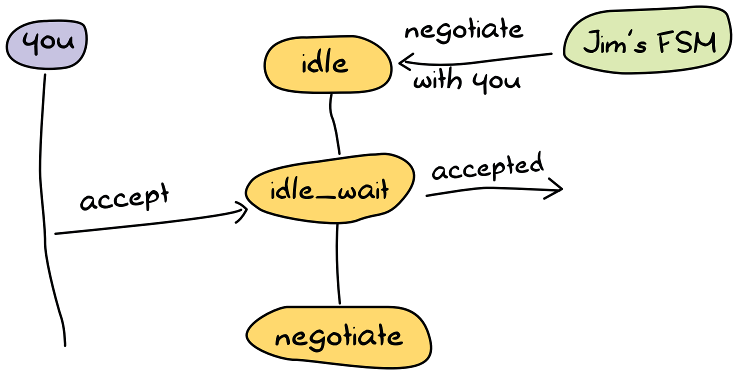
At first, both FSMs start in the `idle` state. At this point, one thing we can do is ask some other player to negotiate with us:



We go into `idle_wait` mode in order to wait for an eventual reply after our FSM forwarded the request. Once the other FSM sends the reply, ours can switch to `negotiate`:

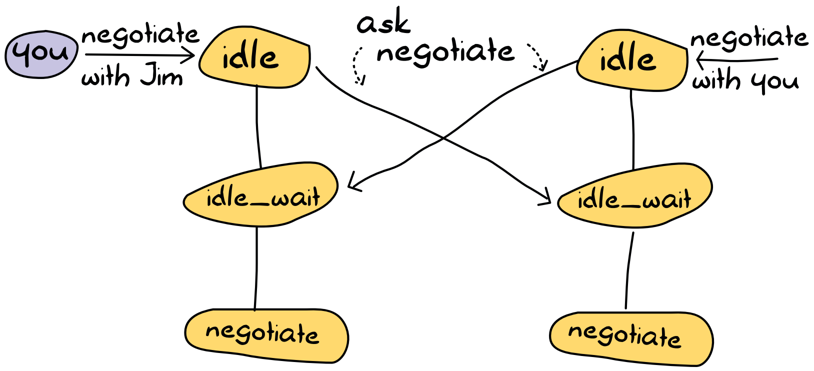


The other player should also be in the `negotiate` state after this. Obviously, if we can invite the other player, the other player can invite us. If all goes well, the diagram should end up looking like this:



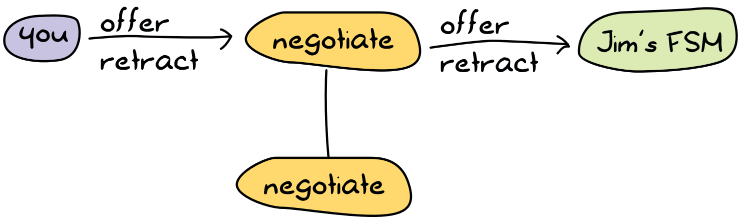
So this is pretty much the opposite of the two previous state diagrams bundled into one. Note that we expect the player to accept the offer in this case.

What happens if, by pure chance, we ask the other player to trade with us at the same time he asks us to trade?

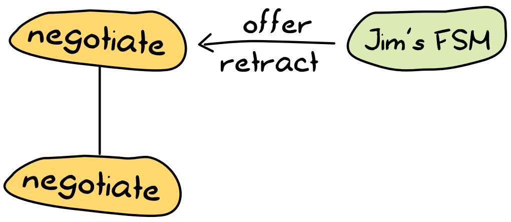


In this case, both clients ask their own FSM to negotiate with the other one. As soon as the ask negotiate messages are sent, both FSMs switch to `idle_wait` state. Then they will be able to process the negotiation question. Reviewing the previous state diagrams, we see that this combination of events is the only time we'll receive ask negotiate messages while in the `idle_wait` state. Consequently, we know that getting these messages in `idle_wait` means that we hit the race condition and can assume both users want to talk to each other. We can move both of them to the `negotiate` state.

So now we're negotiating. Good for us! According to the actions listed earlier, we must support users offering items and then retracting the offer:

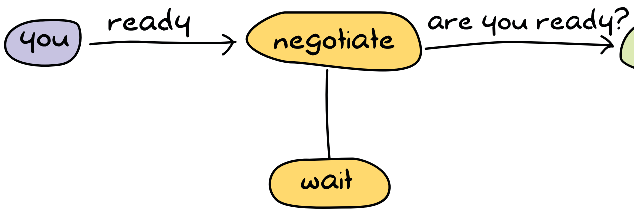


All this does is forward our client's message to the other FSM. Both FSMs will need to hold a list of items offered by either player, so they can update that list when receiving such messages. We stay in the `negotiate` state after this; maybe the other player wants to offer items:

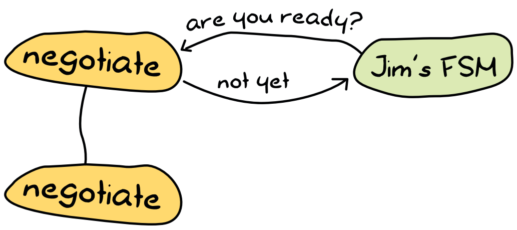


Here, our FSM basically acts in a similar manner by remaining in the `negotiate` state. This is normal.

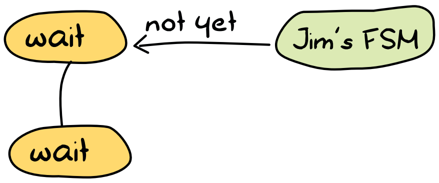
Once we get tired of offering things and think we're generous enough, we need to say we're ready to officialize the trade. Because we must synchronize both players, we'll need to use an intermediary state, as we did for `idle` and `idle_wait`:



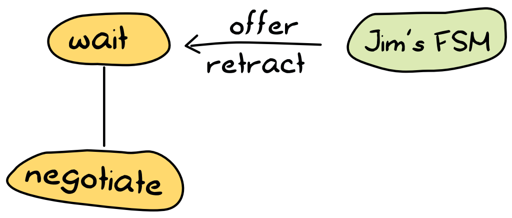
Here, as soon as our player is ready, our FSM asks Jim's FSM if he is ready. Pending its reply, our own FSM falls into its `wait` state. The reply we'll get will depend on Jim's FSM state. If it's in `wait` state, it will tell us that it's ready. Otherwise, it will tell us that it's not ready yet. That's precisely what our FSM automatically replies to Jim if he asks us if we are ready when in `negotiate` state:



Our FSM will remain in `negotiate` mode until our player says he is ready. Let's assume he did and we're now in the `wait` state. However, Jim's not there yet. This means that when we declared ourselves as ready, we'll have asked Jim if he was also ready and his FSM will have replied "not yet":

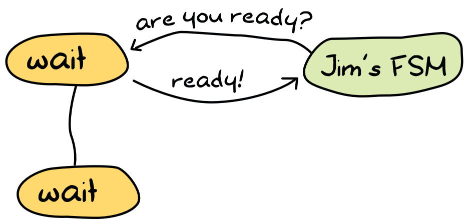


He is not ready, but we are. We can't do much but keep waiting. While waiting for Jim (who is still negotiating, by the way), it is possible that he will try to send us more items or maybe cancel his previous offers:

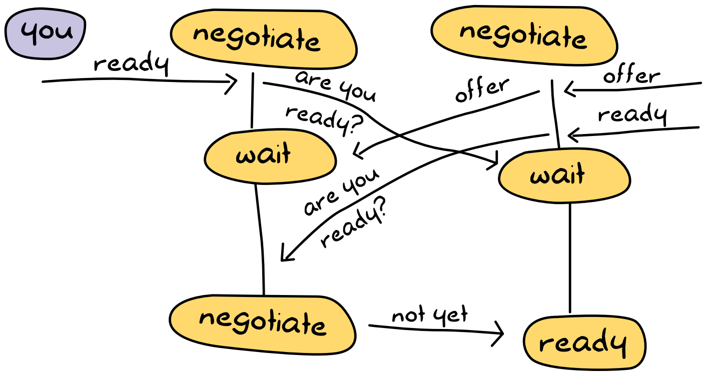


Of course, we want to avoid Jim removing all of his items and then clicking "I'm ready," screwing us over in the process. As soon as he changes the items offered, we go back into the `negotiate` state so we can either modify our own offer or examine the current one and decide we're ready. Rinse and repeat.

At some point, Jim will be ready to finalize the trade, too. When this happens, his FSM will ask ours if we are ready:



Then our FSM replies that we indeed are ready. We stay in the `wait` state and refuse to move to the `ready` state though. Why is this? Because there's a potential race condition! Imagine that the following sequence of events takes place, without doing this necessary step:

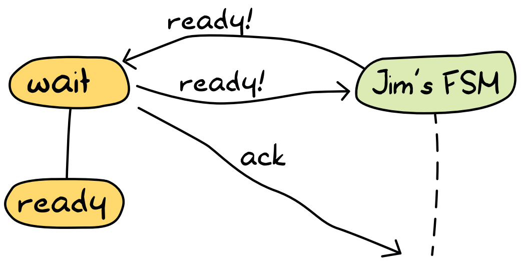


Because of the way messages are received, we could possibly process the item offer only *after* we declared ourselves ready and also *after* Jim declared himself as ready. This means that as soon as we read the offer message, we switch back to the `negotiate` state. During that time, Jim will have told us he is ready. If he were to change states right there and move on to `ready` (as in the preceding illustration), he would be caught waiting indefinitely, while we wouldn't know what the hell to do. This could also happen the other way around!

One way to solve this is by adding a layer of indirection (thanks to David Wheeler). This is why we stay in `wait` mode and send "ready!" (as shown in our previous state diagram).

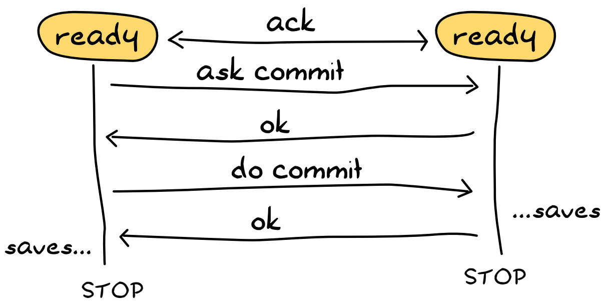
NOTE *David Wheeler, a computer scientist* (http://en.wikipedia.org/wiki/David_Wheeler_(computer_scientist))*, is often quoted as saying, "All problems in computer science can be solved by another level of indirection . . . except for the problem of too many layers of indirection."*

Here's how we deal with that "ready!" message, assuming we were already in the `ready` state because we told our FSM we were ready beforehand:



When we receive "ready!" from the other FSM, we send "ready!" back again. This is to make sure that we won't have the double race condition mentioned earlier. This will create a superfluous "ready!" message in one of the two FSMs, but we'll just have to ignore it in this case. We then send an "ack" message (and Jim's FSM will do the same) before moving to the `ready` state. The "ack" message exists due to some implementation details about synchronizing clients, which we'll look at later in the chapter. Whew—we finally managed to synchronize both players.

So now there's the `ready` state. This one is a bit special. Both players are ready and have basically given the FSMs all the control they need. This lets us implement a bastardized version of a *two-phase commit* to make sure things go right when making the trade official:



Our version (as described above) will be rather simplistic. Writing a truly correct two-phase commit would require a lot more code than what is necessary for us to understand FSMs. (For more information about two-phase commits, see *http://en.wikipedia.org/wiki/Two_phase_commit*.)

Finally, we need to allow the trade to be canceled at any time. This means that somehow—no matter what state we're in—we'll need to listen to the "cancel" message from either side and quit the transaction. It should also be common courtesy to let the other side know we're going before leaving.

At this point, we've covered a whole lot of information. Don't worry if it takes a while to fully grasp the concepts. A bunch of people looked over my protocol to see if it was right, and even then, we all missed a few race conditions, which I caught a few days later when reviewing the code. It's normal to need to read the code more than once, especially if you are not used to asynchronous protocols. If this is the case, I fully encourage you to try to design your own protocol. Then ask yourself these questions:

- What happens if two people do the same actions very fast?
- What if they chain two other events quickly?
- What do I do with messages I don't handle when changing states?

You'll see that the complexity grows quickly. You might find a solution similar to mine, or possibly a better one (let me know if this is the case!). No matter the outcome, it's a very interesting problem to work on, and our FSMs are still relatively simple.

Once you've digested all of this (or before, if you're a rebel reader), you can move on to the next section, where we implement the gaming system. For now, you can take a nice coffee break if you feel like doing so.

Game Trading Between Two Players

Now we'll implement our trading system protocol with OTP's `gen_fsm`. The first step is to create the interface.

The Public Interface

There will be three callers for our module: the player, the `gen_fsm` behavior, and the other player's FSM. We will need to export only the player function and `gen_fsm` functions, though. This is because the other FSM will also run within the `trade_fsm` module and can access them from the inside.

```
-module(trade_fsm).
-behavior(gen_fsm).

%% public API
-export([start/1, start_link/1, trade/2, accept_trade/1,
         make_offer/2, retract_offer/2, ready/1, cancel/1]).
%% gen_fsm callbacks
-export([init/1, handle_event/3, handle_sync_event/4, handle_info/3,
         terminate/3, code_change/4,
         % custom state names
         idle/2, idle/3, idle_wait/2, idle_wait/3, negotiate/2,
         negotiate/3, wait/2, ready/2, ready/3]).
```

So that's our API. You can see we'll have some functions that will be both synchronous and asynchronous (`idle/2` and `idle/3`, for example). This is mostly because we want our client to call us synchronously in some cases, but the other FSM can do it asynchronously. Having the client synchronous simplifies our logic a whole lot by limiting the number of contradicting messages that can be sent one after the other. We'll get to that part when we add the `gen_fsm` callbacks later. Let's first implement the actual public API according to the preceding protocol definition.

```
%%% PUBLIC API
start(Name) ->
    gen_fsm:start(?MODULE, [Name], []).

start_link(Name) ->
    gen_fsm:start_link(?MODULE, [Name], []).

%% Ask for a begin session. Returns when/if the other accepts.
trade(OwnPid, OtherPid) ->
    gen_fsm:sync_send_event(OwnPid, {negotiate, OtherPid}, 30000).

%% Accept someone's trade offer.
accept_trade(OwnPid) ->
    gen_fsm:sync_send_event(OwnPid, accept_negotiate).

%% Send an item on the table to be traded.
make_offer(OwnPid, Item) ->
    gen_fsm:send_event(OwnPid, {make_offer, Item}).

%% Cancel trade offer.
retract_offer(OwnPid, Item) ->
    gen_fsm:send_event(OwnPid, {retract_offer, Item}).

%% Mention that you're ready for a trade. When the other
%% player also declares they're ready, the trade is done.
ready(OwnPid) ->
    gen_fsm:sync_send_event(OwnPid, ready, infinity).
```

```
%% Cancel the transaction.
cancel(OwnPid) ->
    gen_fsm:sync_send_all_state_event(OwnPid, cancel).
```

This is rather standard, and we've already covered these `gen_fsm` functions (except `start/3-4` and `start_link/3-4`, which I believe you can figure out) in this chapter.

FSM-to-FSM Functions

Next, we'll implement the FSM-to-FSM functions. The first ones have to do with trade setups, when we want to invite the other user to join us in a trade.

```
%% Ask the other FSM's Pid for a trade session.
ask_negotiate(OtherPid, OwnPid) ->
    gen_fsm:send_event(OtherPid, {ask_negotiate, OwnPid}).

%% Forward the client message accepting the transaction.
accept_negotiate(OtherPid, OwnPid) ->
    gen_fsm:send_event(OtherPid, {accept_negotiate, OwnPid}).
```

The first function asks the other pid if it wants to trade, and the second one is used to reply (asynchronously, of course).

We can then write the functions to offer and cancel offers. According to our protocol, this is how these functions should look:

```
%% Forward a client's offer.
do_offer(OtherPid, Item) ->
    gen_fsm:send_event(OtherPid, {do_offer, Item}).

%% Forward a client's offer cancellation.
undo_offer(OtherPid, Item) ->
    gen_fsm:send_event(OtherPid, {undo_offer, Item}).
```

The next calls relate to being ready for trade or not. Again, given our protocol, we have three messages in total. The first is `are_you_ready`, which can have the two messages `not_yet` or `'ready!'` as replies.

```
%% Ask the other side if he's ready to trade.
are_you_ready(OtherPid) ->
    gen_fsm:send_event(OtherPid, are_you_ready).

%% Reply that the side is not ready to trade,
%% i.e. is not in 'wait' state.
not_yet(OtherPid) ->
    gen_fsm:send_event(OtherPid, not_yet).

%% Tells the other fsm that the user is currently waiting
%% for the ready state. State should transition to 'ready'.
am_ready(OtherPid) ->
    gen_fsm:send_event(OtherPid, 'ready!').
```

The other functions are those that are to be used by both FSMs when doing the commit in the ready state. Their precise usage will be described in more detail later in the chapter, but their names and the sequence/state diagram shown earlier should give you an idea of their purpose, and you can still transcribe them to your own version of trade_fsm.

```
%% Acknowledge that the fsm is in a ready state.
ack_trans(OtherPid) ->
    gen_fsm:send_event(OtherPid, ack).

%% Ask if ready to commit.
ask_commit(OtherPid) ->
    gen_fsm:sync_send_event(OtherPid, ask_commit).

%% Begin the synchronous commit.
do_commit(OtherPid) ->
    gen_fsm:sync_send_event(OtherPid, do_commit).
```

Oh, and there's also the courtesy function allowing us to warn the other FSM we canceled the trade:

```
notify_cancel(OtherPid) ->
    gen_fsm:send_all_state_event(OtherPid, cancel).
```

The gen_fsm Callbacks

We can now move to the really interesting part: the gen_fsm callbacks. The first callback is init/1. In our case, we'll want each FSM to hold a name for the user it represents (that way, our output will be nicer) in the data it keeps passing on to itself as the last argument of each callback. What else do we want to hold in memory? In our case, we want the other player's (Jim's) FSM pid, the items we offer, and the items the other player's FSM offers. We'll also add the reference of a monitor (so we know to abort if the other dies) and a from field, used to do delayed replies.

```
-record(state, {name="",
                other,
                ownitems=[],
                otheritems=[],
                monitor,
                from}).
```

In the case of init/1, we'll only care about our name for now. Note that we'll begin in the idle state.

```
init(Name) ->
    {ok, idle, #state{name=Name}}.
```

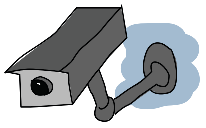
The next callbacks to consider are the states themselves. So far, we have covered the state transitions and calls that can be made, but we'll need a way to make sure everything goes all right. We'll write a few utility functions first.

```
%% Send players a notice. This could be messages to their clients
%% but for our purposes, outputting to the shell is enough.
notice(#state{name=N}, Str, Args) ->
    io:format("~s: "++Str++"~n", [N|Args]).

%% Allows to log unexpected messages.
unexpected(Msg, State) ->
    io:format("~p received unknown event ~p while in state ~p~n",
              [self(), Msg, State]).
```

And we can start with the `idle` state. For the sake of convention, we'll cover the asynchronous version first. This part of the `idle` state callbacks shouldn't need to care about anything but the *other player* asking for a trade. This is because our own player, if you look at the API functions, will use a synchronous call and will therefore need a different callback, with three arguments.

```
idle({ask_negotiate, OtherPid}, S=#state{}) ->
    Ref = monitor(process, OtherPid),
    notice(S, "~p asked for a trade negotiation", [OtherPid]),
    {next_state, idle_wait, S#state{other=OtherPid, monitor=Ref}};
idle(Event, Data) ->
    unexpected(Event, idle),
    {next_state, idle, Data}.
```



A monitor is set up to allow us to handle the other dying, and its reference is stored in the FSM's data along with the other's pid, before moving to the `idle_wait` state. Note that we will report all unexpected events and ignore them by staying in the state we were already in. We can have a few out-of-band messages here and there that could be the result of race conditions. It's usually safe to ignore them, but we can't easily get rid of them. It's just better not to crash the whole FSM on receipt of these unknown but somewhat expected messages.

When our own client asks the FSM to contact another player for a trade, it will send a synchronous event. The `idle/3` callback will be needed.

```
idle({negotiate, OtherPid}, From, S=#state{}) ->
    ask_negotiate(OtherPid, self()),
    notice(S, "asking user ~p for a trade", [OtherPid]),
    Ref = monitor(process, OtherPid),
    {next_state, idle_wait, S#state{other=OtherPid, monitor=Ref, from=From}};
idle(Event, _From, Data) ->
    unexpected(Event, idle),
    {next_state, idle, Data}.
```

We proceed in a way similar to the asynchronous version, except we need to actually ask the other side whether it wants to negotiate with us. You'll notice that we do not reply to the client yet. This is because we have nothing interesting to say, and we want the client locked and waiting for the trade to be accepted before doing anything. The reply will be sent only if the other side accepts once we're in idle_wait.

When we're there, we need to deal with the other player agreeing to negotiate following our invitation or asking to negotiate at the same time we did (a race condition, as described in the protocol).

```
idle_wait({ask_negotiate, OtherPid}, S=#state{other=OtherPid}) ->
    gen_fsm:reply(S#state.from, ok),
    notice(S, "starting negotiation", []),
    {next_state, negotiate, S};
%% The other side has accepted our offer. Move to negotiate state.
idle_wait({accept_negotiate, OtherPid}, S=#state{other=OtherPid}) ->
    gen_fsm:reply(S#state.from, ok),
    notice(S, "starting negotiation", []),
    {next_state, negotiate, S};
idle_wait(Event, Data) ->
    unexpected(Event, idle_wait),
    {next_state, idle_wait, Data}.
```

This gives us two transitions to the negotiate state, but remember that we must use gen_fsm:reply/2 to reply to our client to say it's okay to start offering items. There's also the case of our FSM's client accepting the trade suggested by the other party.

```
idle_wait(accept_negotiate, _From, S=#state{other=OtherPid}) ->
    accept_negotiate(OtherPid, self()),
    notice(S, "accepting negotiation", []),
    {reply, ok, negotiate, S};
idle_wait(Event, _From, Data) ->
    unexpected(Event, idle_wait),
    {next_state, idle_wait, Data}.
```

Again, this one moves on to the negotiate state. Here, we must handle asynchronous queries to add and remove items coming both from the client and the other FSM. However, we have not yet decided how to store items. Let's say we're somewhat lazy and assume users won't trade that many items, so simple lists will do it for now. However, we might need to change that later, so it would be a good idea to wrap item operations in their own functions. Add the following functions at the bottom of the file with notice/3 and unexpected/2:

```
%% Adds an item to an item list.
add(Item, Items) ->
    [Item | Items].
```

```
%% Removes an item from an item list.
remove(Item, Items) ->
    Items -- [Item].
```

These functions are simple, but they have the role of isolating the actions (adding and removing items) from their implementation (using lists). We could easily move to proplists, dicts, or any other data structure without disrupting the rest of the code.

Using both of these functions, we can implement offering and removing items:

```
negotiate({make_offer, Item}, S=#state{ownitems=OwnItems}) ->
    do_offer(S#state.other, Item),
    notice(S, "offering ~p", [Item]),
    {next_state, negotiate, S#state{ownitems=add(Item, OwnItems)}};
%% Own side retracting an item offer.
negotiate({retract_offer, Item}, S=#state{ownitems=OwnItems}) ->
    undo_offer(S#state.other, Item),
    notice(S, "cancelling offer on ~p", [Item]),
    {next_state, negotiate, S#state{ownitems=remove(Item, OwnItems)}};
%% Other side offering an item.
negotiate({do_offer, Item}, S=#state{otheritems=OtherItems}) ->
    notice(S, "other player offering ~p", [Item]),
    {next_state, negotiate, S#state{otheritems=add(Item, OtherItems)}};
%% Other side retracting an item offer.
negotiate({undo_offer, Item}, S=#state{otheritems=OtherItems}) ->
    notice(S, "Other player cancelling offer on ~p", [Item]),
    {next_state, negotiate, S#state{otheritems=remove(Item, OtherItems)}};
```

This is an ugly aspect of using asynchronous messages on both sides. One set of messages has the form "make" and "retract," while the other has "do" and "undo." This is entirely arbitrary and only used to differentiate between player-to-FSM communications and FSM-to-FSM communications. Note that in those messages coming from our own player, we need to tell the other side about the changes we're making.

Another responsibility is to handle the are_you_ready message in the protocol. This is the last asynchronous event to handle in the negotiate state.

```
negotiate(are_you_ready, S=#state{other=OtherPid}) ->
    io:format("Other user ready to trade.~n"),
    notice(S,
           "Other user ready to transfer goods:~n"
           "You get ~p, The other side gets ~p",
           [S#state.otheritems, S#state.ownitems]),
    not_yet(OtherPid),
    {next_state, negotiate, S};
negotiate(Event, Data) ->
    unexpected(Event, negotiate),
    {next_state, negotiate, Data}.
```

As described in the protocol, whenever we're not in the wait state and receive this message, we must reply with not_yet. We're also outputting trade details to the user so a decision can be made.

When such a decision is made and the user is ready, the ready event will be sent. This one should be synchronous because we don't want the user to keep modifying his offer by adding items while claiming he is ready.

```
negotiate(ready, From, S = #state{other=OtherPid}) ->
    are_you_ready(OtherPid),
    notice(S, "asking if ready, waiting", []),
    {next_state, wait, S#state{from=From}};
negotiate(Event, _From, S) ->
    unexpected(Event, negotiate),
    {next_state, negotiate, S}.
```

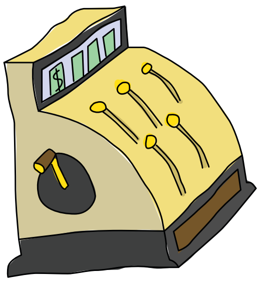
At this point, a transition to the wait state should be made. Note that just waiting for the other player is not interesting. We save the From variable so we can use it with gen_fsm:reply/2 when we have something to tell the client.

The wait state is a funny beast. New items might be offered and retracted because the other player might not be ready. So it makes sense to automatically roll back to the negotiate state. It would suck to have great items offered to us, only for the other player to remove them and declare himself ready, stealing our loot. Going back to negotiation is a good decision.

```
wait({do_offer, Item}, S=#state{otheritems=OtherItems}) ->
    gen_fsm:reply(S#state.from, offer_changed),
    notice(S, "other side offering ~p", [Item]),
    {next_state, negotiate, S#state{otheritems=add(Item, OtherItems)}};
wait({undo_offer, Item}, S=#state{otheritems=OtherItems}) ->
    gen_fsm:reply(S#state.from, offer_changed),
    notice(S, "Other side cancelling offer of ~p", [Item]),
    {next_state, negotiate, S#state{otheritems=remove(Item, OtherItems)}};
```

Now that's something meaningful, and we reply to the player with the coordinates we stored in S#state.from.

The next messages we need to worry about are those related to synchronizing both FSMs so they can move to the ready state and confirm the trade. For this set, we should really focus on the protocol defined earlier.



The three messages we could have are are_you_ready (because the other player just declared himself ready), not_yet (because we asked the other player if he was ready and he was not), and 'ready!' (because we asked the other player if he was ready and he was).

We'll start with are_you_ready. Remember that in the protocol we said that a race condition could be hidden there. The only thing we can do is send the 'ready!' message with am_ready/1 and deal with the rest later.

```
wait(are_you_ready, S=#state{}) ->
    am_ready(S#state.other),
    notice(S, "asked if ready, and I am. Waiting for same reply", []),
    {next_state, wait, S};
```

We'll be stuck waiting again, so it's not worth replying to our client yet. Similarly, we won't reply to the client when the other side sends a `not_yet` reply to our invitation.

```
wait(not_yet, S = #state{}) ->
    notice(S, "Other not ready yet", []),
    {next_state, wait, S};
```

On the other hand, if the other player is ready, we send an extra `'ready!'` message to the other FSM, reply to our own player, and then move to the `ready` state.

```
wait('ready!', S=#state{}) ->
    am_ready(S#state.other),
    ack_trans(S#state.other),
    gen_fsm:reply(S#state.from, ok),
    notice(S, "other side is ready. Moving to ready state", []),
    {next_state, ready, S};
%% Don't care about these!
wait(Event, Data) ->
    unexpected(Event, wait),
    {next_state, wait, Data}.
```

You might have noticed that we've used `ack_trans/1`. In fact, both FSMs should use it. Why is this? To understand, we need to start looking at what goes on in the `ready` state.

When in the `ready` state, both players' actions become useless (except canceling). We won't care about new item offers. This gives us some liberty. Basically, both FSMs can talk to each other freely without worrying about the rest of the world. This lets us implement our bastardization of a two-phase commit. To begin this commit without either player acting, we'll need an event to trigger an action from the FSMs. The `ack` event from `ack_trans/1` is used for that. As soon as we're in the `ready` state, the message is treated and acted upon, and the transaction can begin.

Two-phase commits require synchronous communications, though. This means we can't have both FSMs starting the transaction at once, because they will end up deadlocked. The secret is to find a way to decide that one FSM should initiate the commit, while the other will sit and wait for orders from the first one.

It turns out that the engineers and computer scientists who designed Erlang were pretty smart (well, we knew that already). The pids of any processes can be compared to each other and sorted. This can be done no matter when the process was spawned, whether it's still alive or not, or if it comes from another VM (we'll see more about this when we get into distributed Erlang in Chapter 26).

Knowing that two pids can be compared and one will be greater than the other, we can write a function `priority/2` that will take two pids and tell a process whether it has been elected.

```
priority(OwnPid, OtherPid) when OwnPid > OtherPid -> true;
priority(OwnPid, OtherPid) when OwnPid < OtherPid -> false.
```

And by calling this function, we can have one process starting the commit and the other following orders.

Here's what this gives us when included in the `ready` state, after receiving the `ack` message:

```
ready(ack, S=#state{}) ->
    case priority(self(), S#state.other) of
        true ->
            try
                notice(S, "asking for commit", []),
                ready_commit = ask_commit(S#state.other),
                notice(S, "ordering commit", []),
                ok = do_commit(S#state.other),
                notice(S, "committing...", []),
                commit(S),
                {stop, normal, S}
            catch Class:Reason ->
                %% Abort! Either ready_commit or do_commit failed.
                notice(S, "commit failed", []),
                {stop, {Class, Reason}, S}
            end;
        false ->
            {next_state, ready, S}
    end;
ready(Event, Data) ->
    unexpected(Event, ready),
    {next_state, ready, Data}.
```

This big `try ... catch` expression is the leading FSM deciding how the commit works. Both `ask_commit/1` and `do_commit/1` are synchronous. This lets the leading FSM call them freely. You can see that the other FSM just waits. It will then receive the orders from the leading process. The first message should be `ask_commit`. This is just to make sure both FSMs are still there—nothing bad happened, and they're both dedicated to completing the task.

```
ready(ask_commit, _From, S) ->
    notice(S, "replying to ask_commit", []),
    {reply, ready_commit, ready, S};
```

Once this is received, the leading process will ask to confirm the transaction with do_commit. That's when we must commit our data.

```
ready(do_commit, _From, S) ->
    notice(S, "committing...", []),
    commit(S),
    {stop, normal, ok, S};
ready(Event, _From, Data) ->
    unexpected(Event, ready),
    {next_state, ready, Data}.
```

And once it's done, we leave. The leading FSM will receive ok as a reply and will know to commit on its own end afterward. This explains why we need the big try ... catch: If the replying FSM dies or its player cancels the transaction, the synchronous calls will crash after a timeout. The commit should be aborted in this case.

Just so you know, the commit function is defined as follows:

```
commit(S = #state{}) ->
    io:format("Transaction completed for ~s. "
              "Items sent are:~n~p,~n received are:~n~p.~n"
              "This operation should have some atomic save "
              "in a database.~n",
              [S#state.name, S#state.ownitems, S#state.otheritems]).
```

Pretty underwhelming, eh? It's generally not possible to do a true safe commit with only two participants; a third party is usually required to judge if both players did everything right. A true commit function should contact that third party on behalf of both players, and then do the safe write to a database for them or roll back the whole exchange. We won't go into such details here, and the current commit/1 function will be enough for this example.

We're not finished yet. We have not yet covered two types of events: a player canceling the trade and the other player's FSM crashing. The former can be dealt with by using the callbacks handle_event/3 and handle_sync_event/4. Whenever the other user cancels, we'll receive an asynchronous notification.

```
%% The other player has sent this cancel event.
%% Stop whatever we're doing and shut down!
handle_event(cancel, _StateName, S=#state{}) ->
    notice(S, "received cancel event", []),
    {stop, other_cancelled, S};
handle_event(Event, StateName, Data) ->
    unexpected(Event, StateName),
    {next_state, StateName, Data}.
```

And we must not forget to tell the other player before we quit, like this:

```
%% This cancel event comes from the client. We must warn the other
%% player that we have a quitter!
handle_sync_event(cancel, _From, _StateName, S = #state{}) ->
    notify_cancel(S#state.other),
    notice(S, "cancelling trade, sending cancel event", []),
    {stop, cancelled, ok, S};
%% Note: DO NOT reply to unexpected calls. Let the call-maker crash!
handle_sync_event(Event, _From, StateName, Data) ->
    unexpected(Event, StateName),
    {next_state, StateName, Data}.
```

The last event to take care of is when the other FSM goes down. Fortunately, we set a monitor back in the `idle` state. We can match on this and react accordingly:

```
handle_info({'DOWN', Ref, process, Pid, Reason}, _, S=#state{other=Pid, monitor=Ref}) ->
    notice(S, "Other side dead", []),
    {stop, {other_down, Reason}, S};
handle_info(Info, StateName, Data) ->
    unexpected(Info, StateName),
    {next_state, StateName, Data}.
```

Note that even if the `cancel` or `'DOWN'` events happen while we're in the commit, everything should be safe, and the players will still have their own items. No exploit allowing people to steal others' items hides in there.

NOTE *We used `io:format/2` for most of our messages to let the FSMs communicate with their own clients. In a real-world application, you might want something more flexible. One approach is to let the client send in a pid, which will receive the notices sent to it. That process could be linked to a GUI or any other system to make the player aware of the events. The `io:format/2` solution was chosen for its simplicity, allowing us to focus on the FSM and the asynchronous protocols.*

There are only two callbacks left to cover: `code_change/4` and `terminate/3`. For now, we don't need to do anything with `code_change/4`. We just export it so the next version of the FSM can call it when it will be reloaded. Our `terminate` function is also really short because we didn't handle real resources in this example.

```
code_change(_OldVsn, StateName, Data, _Extra) ->
    {ok, StateName, Data}.

%% Transaction completed.
terminate(normal, ready, S=#state{}) ->
    notice(S, "FSM leaving.", []);
    terminate(_Reason, _StateName, _StateData) ->
    ok.
```

Whew, we're finally finished.

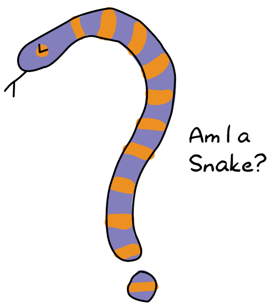
We can now try our trading system. Well, trying it is a bit annoying because we need two processes to communicate with each other. To solve this, I've written the tests in the file *trade_calls.erl* (available from *http://learnyousomeerlang.com/static/erlang/trade_calls.erl*), which can run three different scenarios:

- `main_ab/0` will run a standard trade and output everything.
- `main_cd/0` will cancel the transaction halfway through.
- `main_ef/0` is very similar to `main_ab/0`, except it contains a different race condition.

If you try these, the first and third tests should succeed, while the second one should fail (with a load of error messages, but that's how it goes).

That Was Really Something

If you've found this chapter a bit harder than the others, I must admit that I've just gone crazy and decided to make something difficult out of the generic FSM behavior. If you feel confused, consider these questions:



- Can you understand how different events are handled depending on the state your process is in?
- Do you understand how you can transition from one state to the other?
- Do you know when to use `send_event/2` and `sync_send_event/2-3` as opposed to `send_all_state_event/2` and `sync_send_all_state_event/3`?

If you answered yes to these questions, you understand what `gen_fsm` is about.

The rest of it—the asynchronous protocols, delaying replies and carrying the `From` variable, giving a priority to processes for synchronous calls, bastardized two-phase commits, and so on—*are not essential to understand.* They're mostly there to show what can be done and to highlight the difficulty of writing truly concurrent software, even in a language like Erlang. Erlang doesn't excuse you from planning or thinking, and Erlang won't solve your problems for you. It will only give you tools.

That being said, if you understood everything about these points, you can be proud of yourself (especially if you had never written concurrent software before). You are now starting to really think concurrently.

Fit for the Real World?

A real game would have a lot of stuff going on that could make trading even more complex. Items could be worn by the characters and damaged by enemies while they're being traded. Maybe items could be moved in and out of the inventory while being exchanged. Are the players on the same server? If not, how do you synchronize commits to different databases?

Our trade system is sane when detached from the reality of any game. Before trying to fit it in a game (if you dare), make sure everything goes right. Test it, test it, and test it again. You'll likely find that testing concurrent and parallel code is a complete pain. You'll lose hair, friends, and a piece of your sanity. Even after this, you'll need to keep in mind that your system is always as strong as its weakest link, and thus potentially very fragile nonetheless.

WARNING *While the model for this trade system seems sound, subtle concurrency bugs and race conditions can often rear their ugly heads a long time after they were written, and even if they have been running for years. While my code is generally bulletproof (yeah, right), you sometimes must face swords and knives. Beware the dormant bugs.*

Fortunately, we can put all of this madness behind us. We'll next see how OTP allows you to handle various events, such as alarms and logs, with the help of the `gen_event` behavior.

16

EVENT HANDLERS

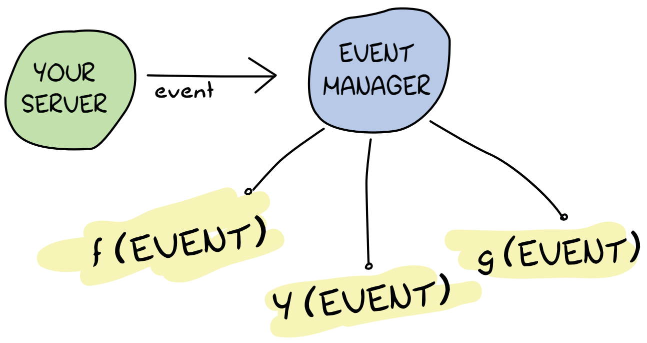
Back in Chapter 13, when we built the reminder application, I mentioned that we could notify clients, whether by instant messaging, email, or some other method. In Chapter 15, our trading system used `io:format/2` to notify people of what was going on. You can probably see the common link between these cases: They're all about letting people (or some process or application) know about an event that happened at some point in time. In one case, we output only the results; in the other, we took the pid of subscribers before sending them a message.

This chapter covers the OTP event handlers, one of the many strategies to handle notifications. After reviewing the handlers, we will put this knowledge in practice by implementing a notification system for sports events.

Handle This! *pumps shotgun*

The output approach we have used for notifications is minimalist and cannot be extended with ease. The one with subscribers is certainly valid. In fact, it's pretty useful when each of the subscribers has a long-running operation to do after receiving an event. In simpler cases, where you do not necessarily want a standby process waiting for events for each of the callbacks, a third approach can be taken.

This third approach simply takes a process that accepts functions and lets them run on any incoming event. This process is usually called an *event manager*, and it might end up looking a bit like this:



Taking this approach has a few advantages:

- If your server has many subscribers, it can keep going because it needs to forward events only once—to the manager.
- If there is a lot of data to be transferred, the data transfer happens only once, and all callbacks operate on that same instance of the data.
- You don't need to spawn processes for short-lived tasks.

And, of course, there are some downsides, too:

- If all functions need to run for a long time, they're going to block each other. This can be prevented by actually having the function forward the event to a process, basically using the event manager as an event forwarder (similar to what we did for the reminder app in Chapter 13).
- A function that loops indefinitely can prevent any new event from being handled until something crashes.

The way to resolve these issues is actually a bit underwhelming. Basically, you need to turn the event manager approach into the subscriber one. Luckily, the event manager approach is flexible enough to make this change relatively easy, and you'll see how in this chapter.

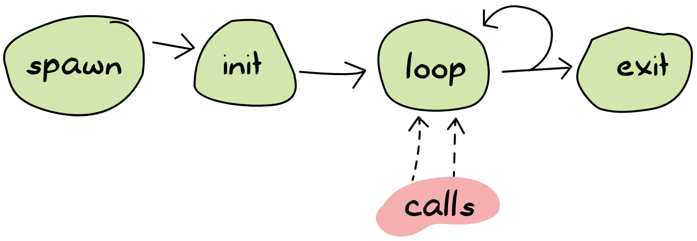
I usually start by writing a basic version of the OTP behavior in pure Erlang beforehand, but in this case, we'll just go straight to the point. Here comes `gen_event`.

Generic Event Handlers

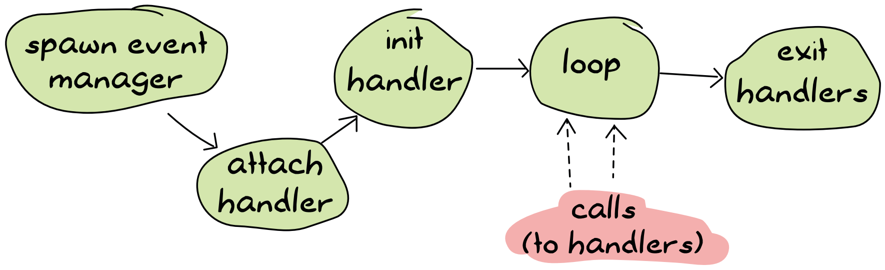
The `gen_event` behavior differs quite a bit from the `gen_server` and `gen_fsm` behaviors in that you are never really starting a process. The part about "accepting a callback" is the reason for this.

The `gen_event` behavior basically runs the process that accepts and calls functions, and you only need to provide a module with these functions. This means that you have nothing to do with event manipulation except to place your callback functions in a format that pleases the event manager. All managing is done for free; you provide only what's specific to your application. This is not really surprising, given that OTP is all about separating the generic from the specific.

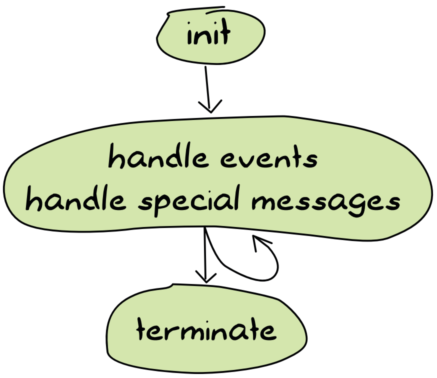
This separation, however, means that the standard spawn/initialize/loop/terminate pattern will be applied only to event handlers. Recall that event handlers are a bunch of functions running in the manager. This means the currently presented model:



switches to something more like this for the programmer:



Each event handler can hold its own state, which is carried around by the manager. Each event handler can then take this form:



Now let's look at the event handlers' callbacks.

The init and terminate Functions

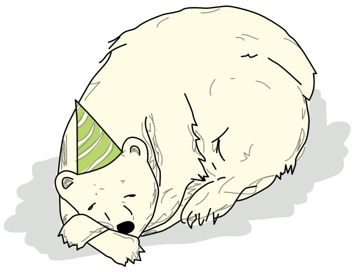
The `init` and `terminate` functions are similar to what we've seen in the previous behaviors with servers and FSMs. The `init/1` function takes a list of arguments and returns `{ok, State}`. Whatever happens in `init/1` should have its counterpart in `terminate/2`.

The handle_event Function

The `handle_event(Event, State)` function is more or less the core of `gen_event`'s callback modules. Like `gen_server`'s `handle_cast/2`, `handle_event/2` works asynchronously. However, it differs in what it can return:

- `{ok, NewState}`
- `{ok, NewState, hibernate}`, which puts the event manager itself into hibernation until the next event
- `remove_handler`
- `{swap_handler, Args1, NewState, NewHandler, Args2}`

The tuple `{ok, NewState}` works in a way similar to what we've seen with `gen_server:handle_cast/2`. It simply updates its own state and doesn't reply to anyone. In the case of `{ok, NewState, hibernate}`, the whole event manager will be put in hibernation. Remember that event handlers run in the same process as their manager.



Then `remove_handler` drops the handler from the manager. This can be useful whenever your event handler knows it's finished and it has nothing else to do.

Finally, there's {swap_handler, Args1, NewState, NewHandler, Args2}. This one is not used too frequently. It removes the current event handler and replaces that handler with a new one. Returning such a tuple will result in the manager first calling CurrentHandler:terminate(Args1, NewState) and removing the current handler, and then adding a new one by calling NewHandler:init(Args2, ResultFromTerminate). This can be useful in the cases where you know some specific event happened and you're better off giving control to a new handler. Generally, this is one of those things that you'll simply know when you need it and apply it then.

All incoming events can come from gen_event:notify/2, which is asynchronous, like gen_server:cast/2. There is also gen_event:sync_notify/2, which is synchronous. This is a bit funny to say, because handle_event/2 remains asynchronous. The idea here is that the function call returns only after all the event handlers have seen and treated the new message. Until then, the event manager will keep blocking the calling process by not replying.

The handle_call Function

The handle_call function is similar to a gen_server's handle_call callback, except that it can return {ok, Reply, NewState}, {ok, Reply, NewState, hibernate}, {remove_handler, Reply}, or {swap_handler, Reply, Args1, NewState, Handler2, Args2}. The gen_event:call/3-4 function is used to make the call.

This raises a question: How does this work when we have something like 15 different event handlers? Do we expect 15 replies, or just 1 that contains them all? Well, in fact, we'll be forced to choose only one handler to reply to us. We'll get into the details of how this is done when we attach handlers to our event manager in "Game Events" on page 253, but if you're impatient, you can refer to the gen_event:add_handler/3 function's documentation to try to figure it out.

The handle_info Function

The handle_info/2 callback is pretty much the same as handle_event (it has the same return values and such), with the exception that it treats only out-of-band messages, such as exit signals and messages sent directly to the event manager with the ! operator. It has use cases similar to those of handle_info in gen_server and gen_fsm.

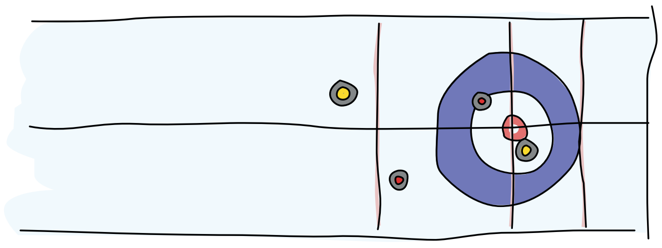
The code_change Function

The code_change function works in the same manner as it does for gen_server, except it's for each individual event handler. It takes the arguments OldVsn, State, and Extra, which are, in order, the version number, the current handler's state, and data we can ignore for now. All it needs to do is return {ok, NewState}.

It's Curling Time!

Now it's time to see what we can do with gen_event. For this example, we'll make a set of event handlers to track game updates of one of the most entertaining sports in the world: curling.

For those who have never seen or played curling (which is a shame!), the rules are relatively simple. Two teams try to send a curling stone (a thick stone disc weighing between 38 and 44 pounds (17 and 20 kilograms) with a handle attached to the top) sliding on the ice to the middle of the red circle:



The teams do this with 16 stones, and the team with the stone closest to the center wins a point at the end of the round (called an *end*). If the team has the two closest stones, it earns two points; if it has the three closest stones, it's worth three points, and so on. There are 10 ends, and the team with the most points at the completion of the 10 ends wins the game.

There are more rules, making the game more fascinating, but this is a book on Erlang, not extremely fascinating winter sports. If you want to learn more about curling, I suggest you head over to the Wikipedia article on the topic.

For this entirely real-world-relevant scenario, we'll be working for the next winter Olympic Games. The city where everything happens has just finished building the arena where the matches will take place, and they're working on getting the scoreboard ready. It turns out that we need to program a system that will let some official enter game events—such as when a stone has been thrown, when a round ends, and when a game is over—and then route these events to the scoreboard, to a stats system, to news reporters' feeds, and so on.

Being as clever as we are, we know this is a chapter on gen_event and deduce we will likely use it to accomplish our task. We won't implement all the rules in this example, but feel free to do so after you've built the sample app—I promise not to be mad.

The Scoreboard

We'll start with the scoreboard. Because they're installing it right now, we'll make use of a fake module that would usually let us interact with it, but for this example, it will use only standard output to show what's going on. This is called a *mock*, and it's there to help us develop code about parts of the system that do not exist yet. This is where the following *curling_scoreboard_hw.erl* file comes in.

```
-module(curling_scoreboard_hw).
-export([add_point/1, next_round/0, set_teams/2, reset_board/0]).

%% This is a 'dumb' module that's only there to replace what a real hardware
%% controller would likely do. The real hardware controller would likely hold
%% some state and make sure everything works right, but this one doesn't mind.

%% Shows the teams on the scoreboard.
set_teams(TeamA, TeamB) ->
    io:format("Scoreboard: Team ~s vs. Team ~s~n", [TeamA, TeamB]).

next_round() ->
    io:format("Scoreboard: round over~n").

add_point(Team) ->
    io:format("Scoreboard: increased score of team ~s by 1~n", [Team]).

reset_board() ->
    io:format("Scoreboard: All teams are undefined and all scores are 0~n").
```

So this is all the functionality the scoreboard has. Scoreboards usually have a timer and other awesome features, but it seems like the Olympics Committee didn't feel like having us implementing trivialities for a tutorial.

Game Events

This hardware interface lets us have a bit of design time to ourselves. We know that there are a few events to handle for now: adding teams, going to the next round, and setting the number of points. We will use only the `reset_board` functionality when starting a new game and won't need it as part of our protocol. The events we need might take the following form in our protocol:

- `{set_teams, TeamA, TeamB}`, where this is translated to a single call to `curling_scoreboard_hw:set_teams(TeamA, TeamB)`
- `{add_points, Team, N}`, where this is translated to `N` calls to `curling_scoreboard_hw:add_point(Team)`
- `next_round`, which gets translated to a single call to the function with the same name

We can start our implementation with this basic event handler skeleton:

```
-module(gen_event_callback).
-behavior(gen_event).
```

```
-export([init/1, handle_event/2, handle_call/2, handle_info/2, code_change/3,
terminate/2]).

init([]) -> {ok, []}.

handle_event(_, State) -> {ok, State}.

handle_call(_, State) -> {ok, ok, State}.

handle_info(_, State) -> {ok, State}.

code_change(_OldVsn, State, _Extra) -> {ok, State}.

terminate(_Reason, _State) -> ok.
```

This is a skeleton that we can use for every `gen_event` callback module out there. For now, the scoreboard event handler itself won't need to do anything special except forward the calls to the hardware module. We expect the events to come from `gen_event:notify/2`, so the handling of the protocol should be done in `handle_event/2`. The file *curling_scoreboard.erl* contains the changes to the skeleton, as follows:

```
-module(curling_scoreboard).
-behavior(gen_event).

-export([init/1, handle_event/2, handle_call/2, handle_info/2, code_change/3,
terminate/2]).

init([]) ->
    {ok, []}.

handle_event({set_teams, TeamA, TeamB}, State) ->
    curling_scoreboard_hw:set_teams(TeamA, TeamB),
    {ok, State};
handle_event({add_points, Team, N}, State) ->
    [curling_scoreboard_hw:add_point(Team) || _ <- lists:seq(1,N)],
    {ok, State};
handle_event(next_round, State) ->
    curling_scoreboard_hw:next_round(),
    {ok, State};
handle_event(_, State) ->
    {ok, State}.

handle_call(_, State) ->
    {ok, ok, State}.

handle_info(_, State) ->
    {ok, State}.
```

You can see the updates done to the handle_event/2 function. Now let's try it:

```
1> c(curling_scoreboard_hw).
{ok,curling_scoreboard_hw}
2> c(curling_scoreboard).
{ok,curling_scoreboard}
3> {ok, Pid} = gen_event:start_link().
{ok,<0.43.0>}
4> gen_event:add_handler(Pid, curling_scoreboard, []).
ok
5> gen_event:notify(Pid, {set_teams, "Pirates", "Scotsmen"}).
Scoreboard: Team Pirates vs. Team Scotsmen
ok
6> gen_event:notify(Pid, {add_points, "Pirates", 3}).
ok
Scoreboard: increased score of team Pirates by 1
Scoreboard: increased score of team Pirates by 1
Scoreboard: increased score of team Pirates by 1
7> gen_event:notify(Pid, next_round).
Scoreboard: round over
ok
8> gen_event:delete_handler(Pid, curling_scoreboard, turn_off).
ok
9> gen_event:notify(Pid, next_round).
ok
```

A few things are going on here. The first is that we're starting gen_event as a stand-alone process. We then attach our event handler to it dynamically with gen_event:add_handler/3. This can be done as many times as you want. However, as mentioned in the handle_call discussion earlier, this might cause problems when you want to work with a particular event handler.

If you want to call, add, or delete a specific handler when there's more than one instance of it, you'll need to find a way to uniquely identify it. My favorite way of doing this (which works great if you don't have anything more specific in mind) is to just use make_ref() as a unique value. To give this value to the handler, you add it by calling add_handler/3 as gen_event:add_handler(Pid, {Module, Ref}, Args). From this point on, you can use {Module, Ref} to talk to that specific handler, and the problem is solved.

Next, we send messages to the event handler, which successfully calls the hardware module. We then remove the handler. Here, turn_off is an argument to the terminate/2 function, which our implementation currently doesn't care about. The handler is gone, but we can still send events to the event manager. Hooray.

One awkward aspect of the preceding code snippet is that we're forced to call the gen_event module directly and show everyone what our protocol looks like. A better option would be to provide an *abstraction module* on top of it that just wraps up all the calls we need. This will look a lot nicer to everyone using our code and will, again, let us change the implementation if (when) we need to do so. It will also let us specify which handlers are necessary to include for a standard curling game.

```
-module(curling).
-export([start_link/2, set_teams/3, add_points/3, next_round/1]).

start_link(TeamA, TeamB) ->
    {ok, Pid} = gen_event:start_link(),
    %% The scoreboard will always be there.
    gen_event:add_handler(Pid, curling_scoreboard, []),
    set_teams(Pid, TeamA, TeamB),
    {ok, Pid}.

set_teams(Pid, TeamA, TeamB) ->
    gen_event:notify(Pid, {set_teams, TeamA, TeamB}).

add_points(Pid, Team, N) ->
    gen_event:notify(Pid, {add_points, Team, N}).

next_round(Pid) ->
    gen_event:notify(Pid, next_round).
```

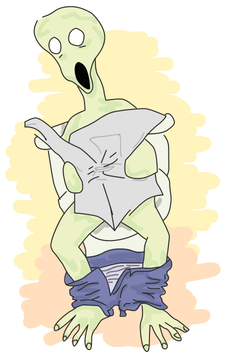
And now we can run it.

```
1> c(curling).
{ok,curling}
2> {ok, Pid} = curling:start_link("Pirates", "Scotsmen").
Scoreboard: Team Pirates vs. Team Scotsmen
{ok,<0.78.0>}
3> curling:add_points(Pid, "Scotsmen", 2).
Scoreboard: increased score of team Scotsmen by 1
Scoreboard: increased score of team Scotsmen by 1
ok
4> curling:next_round(Pid).
Scoreboard: round over
ok
```

This doesn't look like much of an advantage, but it's really about making the code nicer to use (and it reduces the possibilities of writing the messages incorrectly).

This being done, the code should now be usable by officials. Olympics do require us to do a little bit more, say, satisfying the press.

Alert the Press!



We want international reporters to be able to get live data from the official in charge of updating our system. Because this is an example program, we won't go through the steps of setting up a socket and writing a protocol for the updates, but we'll set up the system to do it by putting an intermediary process in charge of it.

Basically, whenever a news organization feels like getting into the game feed, the organization will register its own handler that just forwards the data the organization needs. We'll effectively turn our gen_event server into a kind of message hub, just routing messages to whoever needs them.

First, we'll update the *curling.erl* module with the new interface. Because we want things to be easy to use, we'll add only two functions: join_feed/2 and leave_feed/2. Joining the feed should be doable just by inputting the correct pid for the event manager and the pid to forward all the events to. This should return a unique value that can then be used to unsubscribe from the feed with leave_feed/2.

```
%% Subscribes the pid ToPid to the event feed.
%% The specific event handler for the newsfeed is
%% returned in case someone wants to leave.
join_feed(Pid, ToPid) ->
    HandlerId = {curling_feed, make_ref()},
    gen_event:add_handler(Pid, HandlerId, [ToPid]),
    HandlerId.

leave_feed(Pid, HandlerId) ->
    gen_event:delete_handler(Pid, HandlerId, leave_feed).
```

Note that we're using the technique described earlier for multiple handlers ({curling_feed, make_ref()}). You can see that this function expects a gen_event callback module named curling_feed. If we used only the name of the module as a HandlerId, things would have still worked fine, except that we would have no control over which handler to delete when we're finished with one instance of it. The event manager would just pick one of the handlers in an undefined manner. Using a reference makes sure that some guy from the *Head-Smashed-In Buffalo Jump* (Alberta, Canada) press leaving the place won't disconnect a journalist from *The Economist* (no idea why that magazine would do a report on curling, but you never know). Anyway, here is the implementation for the curling_feed module:

```
-module(curling_feed).
-behavior(gen_event).
```

```
-export([init/1, handle_event/2, handle_call/2, handle_info/2, code_change/3,
terminate/2]).

init([Pid]) -> {ok, Pid}.

handle_event(Event, Pid) ->
    Pid ! {curling_feed, Event},
    {ok, Pid}.

handle_call(_, State) -> {ok, ok, State}.

handle_info(_, State) -> {ok, State}.

code_change(_OldVsn, State, _Extra) -> {ok, State}.

terminate(_Reason, _State) -> ok.
```

The only interesting thing here is still the handle_event/2 function, which blindly forwards all events to the subscribing pid.

Now let's use the new modules.

```
1> c(curling), c(curling_feed).
{ok,curling_feed}
2> {ok, Pid} = curling:start_link("Saskatchewan Roughriders",
2>                                "Ottawa Roughriders").
Scoreboard: Team Saskatchewan Roughriders vs. Team Ottawa Roughriders
{ok,<0.165.0>}
3> HandlerId = curling:join_feed(Pid, self()).
{curling_feed,#Ref<0.0.0.909>}
4> curling:add_points(Pid, "Saskatchewan Roughriders", 2).
Scoreboard: increased score of team Saskatchewan Roughriders by 1
ok
Scoreboard: increased score of team Saskatchewan Roughriders by 1
5> flush().
Shell got {curling_feed,{add_points,"Saskatchewan Roughriders",2}}
ok
6> curling:leave_feed(Pid, HandlerId).
ok
7> curling:next_round(Pid).
Scoreboard: round over
ok
8> flush().
ok
```

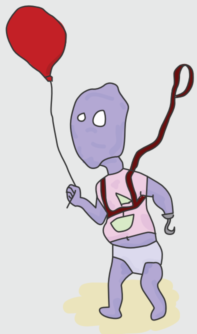
Here, we added ourselves to the feed, got the updates, and then left and stopped receiving them. You can actually try to add many subscribers many times, and it will work fine.

This introduces a problem though. What if one of the curling feed subscribers crashes? Do we just keep the handler going on there? Ideally, we wouldn't need to do that, and in fact, we don't have to. All that needs to be done is to change the call from `gen_event:add_handler/3` to `gen_event:add_sup_handler/3`. If we crash, the handler is gone. Then on the opposite end, if the `gen_event` manager crashes, the message `{gen_event_EXIT, Handler, Reason}` is sent back to us so we can handle it. Easy enough, right? Think again.

DON'T DRINK TOO MUCH KOOL-AID

At some time in your childhood, you probably went to your aunt or grandmother's place for a party or some other event. While there, you would have several adults watching over you in addition to your parents. If you misbehaved, you would get scolded by your mom, dad, aunt, grandmother, and so on, and then everyone would keep harassing you long after you clearly knew you had done something wrong. Well, `gen_event:add_sup_handler/3` is a bit like that—seriously.

Whenever you use `gen_event:add_sup_handler/3`, a link is set up between your process and the event manager so both of them are supervised and the handler knows if its parent process fails. In the section on monitors in Chapter 12, I mentioned that monitors are great for writing libraries that need to know what's going on with other processes because, unlike links, monitors can be stacked. Well, `gen_event` predates the appearance of monitors in Erlang, and a strong commitment to backward-compatibility introduced this pretty bad wart. Basically, because you could have the same process acting as the parent of many event handlers, the library doesn't ever unlink the processes (except when it terminates for good) just in case. Monitors would actually solve the problem, but they aren't being used there.



This mean that everything goes okay when your own process crashes—the supervised handler is terminated (with the call to `YourModule:terminate({stop, Reason}, State)`). Everything goes okay when your handler itself crashes (but not the event manager)—you will receive `{gen_event_EXIT, HandlerId, Reason}`. When the event manager is shut down though, either of the following will happen:

- You will receive the `{gen_event_EXIT, HandlerId, Reason}` message, and then crash because you're not trapping exits.
- You will receive the `{gen_event_EXIT, HandlerId, Reason}` message, and then a standard `'EXIT'` message that is either superfluous or confusing.

That's quite a wart, but at least you know about it. You can try to switch your event handler to a supervised one if you feel like it. It will be safer, even if it risks being more annoying in some cases. Safety first.

We're not finished yet! What happens if some members of the media are not there on time? We need to be able to tell them from the feed what the current state of the game is. For this, we'll write an additional event handler named curling_accumulator. Again, before writing it entirely, we might want to add it to the curling module with the few calls we want, as follows:

```
-module(curling).
-export([start_link/2, set_teams/3, add_points/3, next_round/1]).
-export([join_feed/2, leave_feed/2]).
-export([game_info/1]).

start_link(TeamA, TeamB) ->
    {ok, Pid} = gen_event:start_link(),
    %% The scoreboard will always be there.
    gen_event:add_handler(Pid, curling_scoreboard, []),
    %% Start the stats accumulator.
    gen_event:add_handler(Pid, curling_accumulator, []),
    set_teams(Pid, TeamA, TeamB),
    {ok, Pid}.

...

%% Returns the current game state.
game_info(Pid) ->
    gen_event:call(Pid, curling_accumulator, game_data).
```

Notice that the game_info/1 function here uses only curling_accumulator as a handler ID. In the cases where you have many versions of the same handler, the hint about using make_ref() (or any other means) to ensure you write to the correct handler still holds. Also note that the curling_accumulator handler starts automatically along with the scoreboard.

Now let's put together the module itself. It should be able to hold state for the curling game. So far, we have teams, score, and rounds to track. This information can all be held in a state record and changed on each event received. Then we will only need to reply to the game_data call, as follows:

```
-module(curling_accumulator).
-behavior(gen_event).

-export([init/1, handle_event/2, handle_call/2, handle_info/2, code_change/3,
terminate/2]).

-record(state, {teams=orddict:new(), round=0}).

init([]) -> {ok, #state{}}.

handle_event({set_teams, TeamA, TeamB}, S=#state{teams=T}) ->
    Teams = orddict:store(TeamA, 0, orddict:store(TeamB, 0, T)),
    {ok, S#state{teams=Teams}};
handle_event({add_points, Team, N}, S=#state{teams=T}) ->
    Teams = orddict:update_counter(Team, N, T),
    {ok, S#state{teams=Teams}};
```

```
handle_event(next_round, S=#state{}) ->
    {ok, S#state{round = S#state.round+1}};
handle_event(_Event, State=#state{}) ->
    {ok, State}.

handle_call(game_data, S=#state{teams=T, round=R}) ->
    {ok, {orddict:to_list(T), {round, R}}, S};
handle_call(_, State) ->
    {ok, ok, State}.

handle_info(_, State) -> {ok, State}.

code_change(_OldVsn, State, _Extra) -> {ok, State}.

terminate(_Reason, _State) -> ok.
```

So, we basically just update the state until someone asks for game details, at which point, we'll be sending them back. We did this in a very basic way. Perhaps a smarter way to organize the code would have been to simply keep a list of all the events to ever happen in the game so we could send them all back each time a new process subscribed to the feed. That's not necessary for our purposes here, however, so let's focus on using our new code. Give it a try:

```
1> c(curling), c(curling_accumulator).
{ok,curling_accumulator}
2> {ok, Pid} = curling:start_link("Pigeons", "Eagles").
Scoreboard: Team Pigeons vs. Team Eagles
{ok,<0.242.0>}
3> curling:add_points(Pid, "Pigeons", 2).
Scoreboard: increased score of team Pigeons by 1
ok
Scoreboard: increased score of team Pigeons by 1
4> curling:next_round(Pid).
Scoreboard: round over
ok
5> curling:add_points(Pid, "Eagles", 3).
Scoreboard: increased score of team Eagles by 1
ok
Scoreboard: increased score of team Eagles by 1
Scoreboard: increased score of team Eagles by 1
6> curling:next_round(Pid).
Scoreboard: round over
ok
7> curling:game_info(Pid).
{[{"Eagles",3},{"Pigeons",2}],{round,2}}
```

Enthralling! Surely the Olympic Committee will love our code. We can pat ourselves on the back, cash in a fat check, and go play video games all night.

We haven't covered all that can be done with gen_event as a module. In fact, we haven't discussed the most common use of event handlers: logging and system alarms. I decided against showing them because pretty much any other source on Erlang out there uses gen_event strictly for those purposes. If you're interested in learning more about these uses, check out the error_logger module of the standard library first.

Even though we have not covered the most common uses of gen_event, we've explored all the concepts necessary to understand them, to build our own, and to integrate them into our applications. More important, we've finally covered the three main OTP behaviors used in active code development. We still have a few behaviors left to visit—those that act as a bunch of glue between all of our worker processes, such as the supervisor, which is what Chapter 17 is all about.

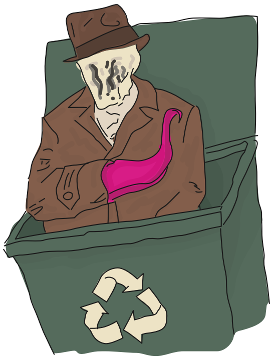
17

WHO SUPERVISES THE SUPERVISORS?

Supervisors are one of the most useful parts of OTP. We've encountered basic supervisors in Chapters 12 and 13, where they offered a way to keep our software going in case of errors by just restarting the faulty processes. This chapter introduces OTP's take on supervisors, which is much better than ours.

In our earlier examples, our supervisors would start a worker process, link to it, and trap exit signals with `process_flag(trap_exit,true)` to know when the process died and restart it. This is fine when we want restarts, but it's also pretty dumb. Imagine that you're using the remote control to turn on the TV. If it doesn't work the first time, you might try again once or twice, just in case you didn't press the right button or the signal went wrong. But our supervisor would keep trying to turn on that TV forever, even if it turned out that the remote had no batteries or didn't belong to that TV. That's a pretty dumb supervisor.

Something else that was dumb about our supervisors was that they could watch only one worker at a time. Although it's sometimes useful to have one supervisor for a single worker, in large applications, this would mean you could have only a chain of supervisors, not a tree. How would you supervise a task where you need two or three workers at once? With our implementation, it just couldn't be done.



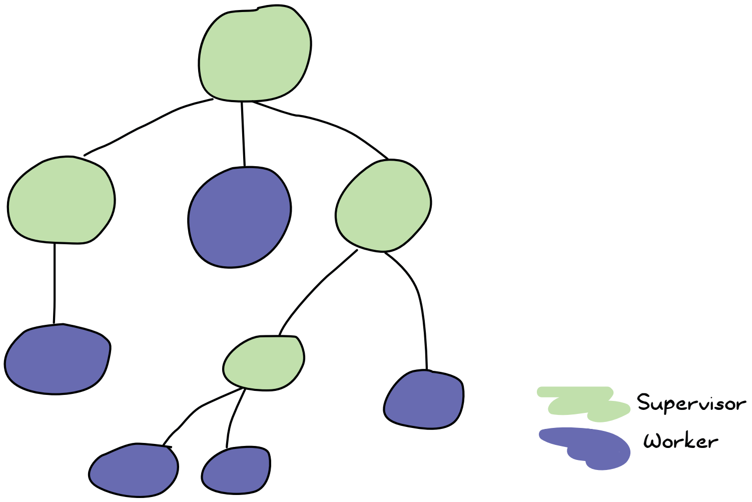
The OTP supervisors, fortunately, provide the flexibility to handle such cases (and more). As you'll see in this chapter, they let you define how many times a worker should be restarted in a given period before giving up. They let you have more than one worker per supervisor, and even let you pick from a few patterns to determine how they should depend on each other in case of a failure.

Supervisor Concepts

Supervisors are one of the simplest behaviors to use and understand, but one of the hardest behaviors to write a good design with. There are various strategies related to supervisors and application design, but before getting to the hard stuff, we need to cover some basic concepts.

One of the terms I've used previously in this book without much of a definition is *worker*. Workers are defined a bit in opposition of supervisors. If supervisors are supposed to be processes that do nothing but make sure their children are restarted when they die, workers are processes that are in charge of doing actual work and that may die while doing so. They are usually not trusted to be safe.

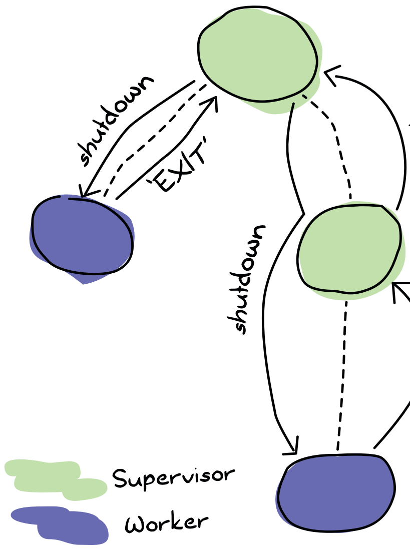
Supervisors can supervise workers and other supervisors. Workers should never be used in any position except under a supervisor:



Why should every process be supervised? Well, the idea is simple: If you're spawning unsupervised processes, how can you be sure they are gone? If you can't measure something, it doesn't exist. If a process resides in the void away from all your supervision trees, how do you know whether it actually exists? How did it get there? Will it happen again?

If it does happen, you'll find yourself leaking memory very slowly—so slowly your VM might suddenly die because the VM no longer has memory, and so slowly you might not be able to easily track the problem until it happens again and again. Of course, you might say, "If I take care and know what I'm doing, things will be fine." Maybe they will be fine, but maybe they won't. In a production system, you don't want to be taking chances. And in the case of Erlang, it's why you have garbage collection to begin with. Keeping things supervised is pretty useful.

Supervision is also useful because it allows you to terminate applications in good order. You'll write Erlang software that is not meant to run forever, but you'll still want it to terminate cleanly. How do you know everything is ready to be shut down? With supervisors, it's easy. Whenever you want to terminate an application, you have the top supervisor of the VM shut down (this is done for you with functions like `init:stop/1`). Then that supervisor asks each of its children to terminate. If some of the children are supervisors, they do the same:



This gives you a well-ordered VM shutdown, which is very hard to achieve without having all of your processes being part of the tree. Of course, there are times when your process will be stuck for some reason and won't terminate correctly. When that happens, supervisors have a way to brutally kill the process.

So, we have workers, supervisors, supervision trees, ways to specify dependencies, ways to tell supervisors when to give up on trying or waiting for their children, and so on. This is not all that supervisors can do; but for now, we have enough information to start looking at how to use them.

Using Supervisors

This has been a very violent chapter so far: Parents spend their time binding their children to trees, forcing them to work before brutally killing them. But we wouldn't be real sadists without actually implementing it all.

When I said supervisors were simple to use, I wasn't kidding. There is a single callback function to provide: `init/1`. The catch is that its return is quite complex. Here's an example return from a supervisor:

```
{ok, {{one_for_all, 5, 60},
      [{fake_id,
        {fake_mod, start_link, [SomeArg]},
        permanent,
        5000,
        worker,
        [fake_mod]},
       {other_id,
        {event_manager_mod, start_link, []},
        transient,
        infinity,
        worker,
        dynamic}]}}.
```

Say what? A general definition might be a bit simpler to work with:

```
{ok, {{RestartStrategy, MaxRestart, MaxTime},[ChildSpec]}}.
```

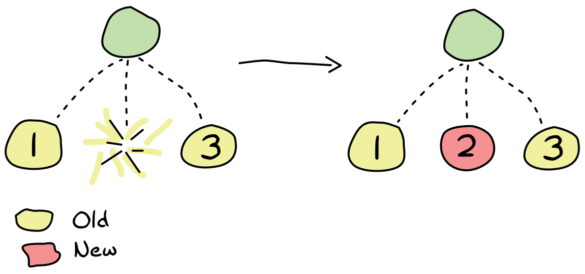
Let's take a look at each of these pieces.

Restart Strategies

The `RestartStrategy` part of the definition can be `one_for_one`, `one_for_all`, `rest_for_one`, or `simple_one_for_one`.

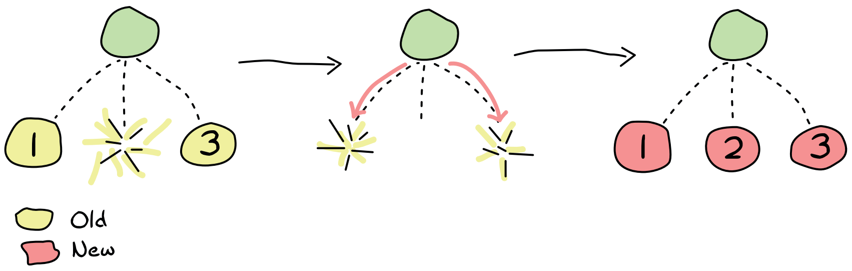
one_for_one

`one_for_one` is an intuitive restart strategy. It basically means that if your supervisor supervises many workers and one of them fails, only that one should be restarted. You should use `one_for_one` whenever the processes being supervised are independent and not really related to each other, or when the process can restart and lose its state without impacting its siblings.



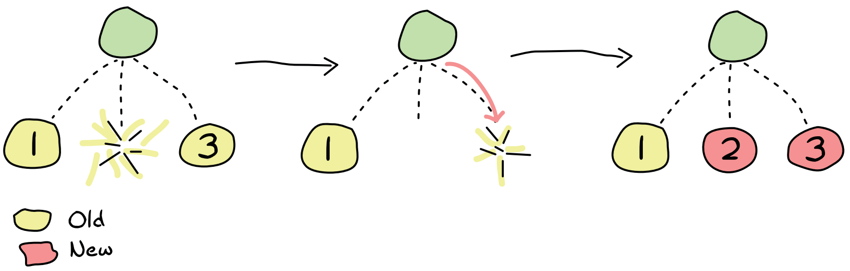
one_for_all

one_for_all has little to do with musketeers. It's to be used whenever all your processes under a single supervisor heavily depend on each other to be able to work normally. Let's say you have decided to add a supervisor on top of the trading system we implemented in Chapter 15. If a trader crashed, it wouldn't make sense to restart only that one of the two traders, because the traders' states would be out of sync. Restarting both of them at once would be a saner choice, and one_for_all is the strategy for that.



rest_for_one

rest_for_one is a more specific kind of strategy. Whenever you need to start processes that depend on each other in a chain (A starts B, which starts C, which starts D, and so on), you can use rest_for_one. It's also useful in the case of services where you have similar dependencies (X works alone, but Y depends on X and Z depends on both). Basically, with a rest_for_one restarting strategy, if a process dies, all the processes that were started after it (depend on it) are restarted, but not the other way around.



simple_one_for_one

Despite its name, the `simple_one_for_one` restart strategy isn't all that simple. This type of supervisor takes only one kind of child, and it's used when you want to dynamically add children to the supervisor, rather than having them started statically.

To say it a bit differently, a `simple_one_for_one` supervisor just sits around, and it knows it can produce one kind of child only. Whenever you want a new child, you ask for it and you get it. This could theoretically be done with the standard `one_for_one` supervisor, but there are practical advantages to using the simple version, as you'll see when we look at dynamic supervision later in the chapter.

NOTE *One of the big differences between* `one_for_one` *and* `simple_one_for_one` *is that* `one_for_one` *holds a list of all the children it has started (and* had *started, if you don't clear it after manipulating it manually), ordered by their starting order, while* `simple_one_for_one` *holds a single definition for all its children and works using a dictionary to hold its data. Basically, when a process crashes, the* `simple_one_for_one` *supervisor will be much faster if you have a large number of children.*

Restart Limits

The last part of the `RestartStrategy` tuple contains the variables `MaxRestart` and `MaxTime`. The idea is that if more than the `MaxRestart` limit happens within `MaxTime` (in seconds), the supervisor just gives up on your code, shuts it down, and then kills itself, never to return. And that is based on restarts for *all* children of the supervisor, not any one of them individually. Fortunately, that supervisor's supervisor might still have hope in its children and start them all over again.

Child Specifications

And now for the `ChildSpec` part of the return value. `ChildSpec` stands for *child specification*. Earlier we had the following two child specifications:

```
[{fake_id,
  {fake_mod, start_link, [SomeArg]},
  permanent,
  5000,
  worker,
  [fake_mod]},
 {other_id,
  {event_manager_mod, start_link, []},
  transient,
  infinity,
  worker,
  dynamic}]
```

The child specification can be described in a more abstract form, as follows:

```
{ChildId, StartFunc, Restart, Shutdown, Type, Modules}.
```

And now we can look at how each part works.

ChildId

`ChildId` is just a name used by the supervisor internally. You will rarely need to use it yourself, although it might happen to be useful for debugging purposes, and sometimes when you decide to actually get a list of all the children of a supervisor. Any term can be used for this identifier, but I suggest making it something readable, just in case you do need it for debugging.

StartFunc

`StartFunc` is a tuple that specifies how to start the supervisor. It's the standard `{M,F,A}` format we've used a few times already. Note that it is *very* important that the starting function here is OTP-compliant and links to its caller when executed. (Hint: Use `gen_*:start_link()` wrapped in your own module, all the time.)

Restart

`Restart` tells the supervisor how to react when that particular child dies. This can take one of three values:

- `permanent`
- `temporary`
- `transient`

A permanent process should always be restarted, no matter what. The supervisors we implemented in our previous applications used this strategy only. This is usually used by vital, long-living processes (or services) running on your node.

On the other hand, a temporary process is a process that should never be restarted. These processes are for short-lived workers that are expected to fail and have few bits of code that depend on them. You usually still want to supervise them to know where they are, and to be able to shut them down cleanly via the supervisor.

Transient processes are a bit of an in-between breed. They're meant to run until they terminate normally, and then they won't be restarted. However, if they die of abnormal causes (the exit reason is anything but `normal`, `shutdown`, or `{shutdown, *Reason*}`), they will be restarted. This restart option is often used for workers that need to succeed at their task, but it won't be used after they do so.

You can have children of all three kinds mixed under a single supervisor. This might affect the restart strategy. A `one_for_all` restart won't be triggered by a temporary process dying, but that temporary process might be restarted under the same supervisor if a permanent process dies first!

Shutdown

Earlier in the chapter, I mentioned being able to shut down entire applications with the help of supervisors. Here's how it's done: When the top-level supervisor is asked to terminate, it calls `exit(ChildPid, shutdown)` on each of the pids. If the child is a worker and trapping exits, it will call its own `terminate` function; otherwise, it's just going to die. When a supervisor gets the `shutdown` signal, it will forward that signal to its own children in the same way.

The `Shutdown` value of a child specification is thus used to give a deadline for the termination. On certain workers, you know you might need to do things like properly close files, notify a service that you're leaving, and so on. In these cases, you might want to use a certain cutoff time, either in milliseconds or set as `infinity` if you are really patient. If the time passes and nothing happens, the process is then brutally killed with `exit(Pid, kill)`. If you don't care about the child and it can die without any consequences without any timeout needed, the atom `brutal_kill` is also an acceptable value. `brutal_kill` will make it so the child is killed with `exit(Pid, kill)`, which is untrappable and instantaneous.

Choosing a good `Shutdown` value is sometimes complex or tricky. If you have a chain of supervisors with `Shutdown` values like 5 → 2 → 5 → 5, the two last ones will likely end up brutally killed, because the second one had a shorter cutoff time. The proper value is entirely application-dependent, and few general tips can be given on the subject.

NOTE *Before Erlang R14B03,* `simple_one_for_one` *children did* not *respect this rule with the* `Shutdown` *time. In the case of* `simple_one_for_one`*, the supervisor would just exit, and it would be left to each of the workers to terminate on its own after its supervisor was gone.*

Type

`Type` lets the supervisor know whether the child is a `supervisor` (it implements either the `supervisor` or `supervisor_bridge` behavior) or a `worker` (any other OTP process). This will be important when upgrading applications with more advanced OTP features, but you do not really need to care about it at the moment—just tell the truth and everything should be fine. You have to trust your supervisors!

Modules

`Modules` is a list of one element: the name of the callback module used by the child behavior. The exception is when you have callback modules whose identity you do not know beforehand (such as event handlers in an event manager). In this case, the value of Modules should be dynamic so that the whole OTP system knows who to contact when using more advanced features, such as releases.

Hooray, we now have covered the basic knowledge required to start supervised processes. You can take a break and digest it all, or move forward to see how supervisors work in practice.

Band Practice

Some practice is in order. And speaking of practice, the perfect example in this case is a band practice! (Well, not *that* perfect, but bear with me for a while.)

Let's say we're managing a band named *RSYNC, made up of a handful of musically inclined programmers: a drummer, a singer, a bass player, and a keytar player (in memory of all the forgotten 1980s glory). Despite a few retro hit cover songs, such as "Thread Safety Dance" and "Saturday Night Coder," the band has a hard time getting a venue. Annoyed with the whole situation, I storm into your office with yet another sugar rush-induced idea of simulating a band in Erlang. You're tired because you live in the same apartment as the drummer (who is the weakest link in this band to be honest, but they stick with him because they don't know any other drummers), so you accept.

Musicians

The first thing we can do is write the individual band members. For our use case, the musicians module will implement a gen_server. Each musician will take an instrument and a skill level as a parameter (so we can say the drummer sucks, while the others are all right). Once a musician has spawned, it will start playing. We'll also have an option to stop musicians, if needed. This gives us the following module and interface:

```
-module(musicians).
-behavior(gen_server).

-export([start_link/2, stop/1]).
-export([init/1, handle_call/3, handle_cast/2,
         handle_info/2, code_change/3, terminate/2]).
```

```
-record(state, {name="", role, skill=good}).
-define(DELAY, 750).

start_link(Role, Skill) ->
    gen_server:start_link({local, Role}, ?MODULE, [Role, Skill], []).

stop(Role) -> gen_server:call(Role, stop).
```

We've defined a ?DELAY macro that we'll use as the standard time span between each time a musician will show himself as playing. As the record definition shows, we'll also need to give each of the musicians a name, as follows:

```
init([Role, Skill]) ->
    %% To know when the parent shuts down.
    process_flag(trap_exit, true),
    %% Sets a seed for random number generation for the life of the process.
    %% Uses the current time to do it. Unique value guaranteed by now().
    random:seed(now()),
    TimeToPlay = random:uniform(3000),
    Name = pick_name(),
    StrRole = atom_to_list(Role),
    io:format("Musician ~s, playing the ~s entered the room~n",
              [Name, StrRole]),
    {ok, #state{name=Name, role=StrRole, skill=Skill}, TimeToPlay}.
```

Two things go on in the init/1 function. First, we start trapping exits. As you'll recall from the description of the terminate/2 function for generic servers in Chapter 14, we need to do this if we want terminate/2 to be called when the server's parent shuts down its children. The rest of the init/1 function sets a random seed (so that each process gets different random numbers) and then creates a random name for itself. The following are the functions to create the names:

```
%% Yes, the names are based off the magic school bus characters'
%% 10 names!
pick_name() ->
    %% The seed must be set for the random functions. Use within the
    %% process that started with init/1.
    lists:nth(random:uniform(10), firstnames())
    ++ " " ++
    lists:nth(random:uniform(10), lastnames()).

firstnames() ->
    ["Valerie", "Arnold", "Carlos", "Dorothy", "Keesha",
     "Phoebe", "Ralphie", "Tim", "Wanda", "Janet"].

lastnames() ->
    ["Frizzle", "Perlstein", "Ramon", "Ann", "Franklin",
     "Terese", "Tennelli", "Jamal", "Li", "Perlstein"].
```

Now we can move on to the implementation. This one is going to be pretty trivial for handle_call and handle_cast.

```
handle_call(stop, _From, S=#state{}) ->
    {stop, normal, ok, S};
handle_call(_Message, _From, S) ->
    {noreply, S, ?DELAY}.

handle_cast(_Message, S) ->
    {noreply, S, ?DELAY}.
```

The only call we have is to stop the musician server, which we agree to do pretty quickly. If we receive an unexpected message, we do not reply to it, and the caller will crash. This is not our problem. We set the timeout in the {noreply, S, ?DELAY} tuples, for one simple reason that we'll see right now.

```
handle_info(timeout, S = #state{name=N, skill=good}) ->
    io:format("~s produced sound!~n",[N]),
    {noreply, S, ?DELAY};
handle_info(timeout, S = #state{name=N, skill=bad}) ->
    case random:uniform(5) of
        1 ->
            io:format("~s played a false note. Uh oh~n",[N]),
            {stop, bad_note, S};
        _ ->
            io:format("~s produced sound!~n",[N]),
            {noreply, S, ?DELAY}
    end;
handle_info(_Message, S) ->
    {noreply, S, ?DELAY}.
```

Each time the server times out, our musicians are going to play a note. If they're good, everything will be completely fine. If they're bad, they will have one chance out of five to miss and play a bad note, which will make them crash. Again, we set the ?DELAY timeout at the end of each nonterminating call.

Then we add an empty code_change/3 callback, as required by the gen_server behavior.

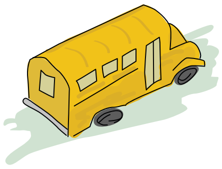
```
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

And we can set the terminate function, as follows:

```
terminate(normal, S) ->
    io:format("~s left the room (~s)~n",[S#state.name, S#state.role]);
terminate(bad_note, S) ->
    io:format("~s sucks! kicked that member out of the band! (~s)~n",
    [S#state.name, S#state.role]);
terminate(shutdown, S) ->
    io:format("The manager is mad and fired the whole band! "
              "~s just got back to playing in the subway~n",
              [S#state.name]);
terminate(_Reason, S) ->
    io:format("~s has been kicked out (~s)~n", [S#state.name, S#state.role]).
```

We have many different messages here. If we terminate with a `normal` reason, it means we've called the `stop/1` function, and so we display the musician left on his own free will. In the case of a `bad_note` message, the musician will crash, and we'll say that it's because the manager (the supervisor we'll soon add) kicked him out of the band.

Then we have the `shutdown` message, which will come from the supervisor. Whenever that happens, it means the supervisor decided to kill all of its children, or in our case, fire all of the musicians. We then add a generic error message for the rest.



Here's a simple use case of a musician:

```
1> c(musicians).
{ok,musicians}
2> musicians:start_link(bass, bad).
Musician Ralphie Franklin, playing the bass entered the room
{ok,<0.615.0>}
Ralphie Franklin produced sound!
Ralphie Franklin produced sound!
Ralphie Franklin played a false note. Uh oh
Ralphie Franklin sucks! kicked that member out of the band! (bass)
3>
=ERROR REPORT==== 6-June-2013::03:22:14 ===
** Generic server bass terminating
** Last message in was timeout
** When Server state == {state,"Ralphie Franklin","bass",bad}
** Reason for termination ==
** bad_note
** exception error: bad_note
```

So we have Ralphie playing and crashing after a bad note. If you try the same with a `good` musician, you'll need to call our `musicians:stop(Instrument)` function in order to stop all the playing.

Band Supervisor

We can now work with the band supervisor. We'll have three grades of supervisors: a lenient one, an angry one, and a total jerk. The lenient supervisor, while still a very pissy person, will fire a single member of the band at a time (`one_for_one`)—the one who fails—until he gets fed up, fires them all, and gives up on bands. The angry supervisor, on the other hand, will fire some of the musicians (`rest_for_one`) on each mistake and wait a shorter amount of time before firing them all and giving up. The jerk supervisor will fire the whole band each time someone makes a mistake, and give up if the band members fail even less often.

```
-module(band_supervisor).
-behavior(supervisor).

-export([start_link/1]).
-export([init/1]).

start_link(Type) ->
    supervisor:start_link({local,?MODULE}, ?MODULE, Type).

%% The band supervisor will allow its band members to make a few
%% mistakes before shutting down all operations, based on what
%% mood he's in. A lenient supervisor will tolerate more mistakes
%% than an angry supervisor, who'll tolerate more than a
%% complete jerk supervisor.
init(lenient) ->
    init({one_for_one, 3, 60});
init(angry) ->
    init({rest_for_one, 2, 60});
init(jerk) ->
    init({one_for_all, 1, 60});
```

The init definition doesn't finish here, but this lets us set the tone for each kind of supervisor we want. The lenient one will restart only one musician and will fail on the fourth failure in 60 seconds. The angry one will accept only two failures, and the jerk supervisor will have very strict standards!

Now let's finish the function and actually implement the band starting functions and whatnot.

```
init({RestartStrategy, MaxRestart, MaxTime}) ->
    {ok, {{RestartStrategy, MaxRestart, MaxTime},
          [{singer,
            {musicians, start_link, [singer, good]},
            permanent, 1000, worker, [musicians]},
           {bass,
            {musicians, start_link, [bass, good]},
            temporary, 1000, worker, [musicians]},
           {drum,
            {musicians, start_link, [drum, bad]},
            transient, 1000, worker, [musicians]},
           {keytar,
            {musicians, start_link, [keytar, good]},
            transient, 1000, worker, [musicians]}
          ]}}.
```

So we'll have three good musicians: the singer, bass player, and keytar player. The drummer is terrible (which makes you pretty mad). The musicians have different Restart values (permanent, transient, or temporary). The singer is permanent, so the band could never work without a singer, even if the current one left by choice. The bass player is temporary, because the band could still play fine without a bass player (frankly, who gives a crap about bass players?). Other musicians are transient, and so they can leave on their own, but they might still need to be replaced in case of errors.

That gives us a functional band_supervisor module, which we can now try.

```
3> c(band_supervisor).
{ok,band_supervisor}
4> band_supervisor:start_link(lenient).
Musician Carlos Terese, playing the singer entered the room
Musician Janet Terese, playing the bass entered the room
Musician Keesha Ramon, playing the drum entered the room
Musician Janet Ramon, playing the keytar entered the room
{ok,<0.623.0>}
Carlos Terese produced sound!
Janet Terese produced sound!
Keesha Ramon produced sound!
Janet Ramon produced sound!
Carlos Terese produced sound!
Keesha Ramon played a false note. Uh oh
Keesha Ramon sucks! kicked that member out of the band! (drum)
... <snip> ...
Musician Arnold Tennelli, playing the drum entered the room
Arnold Tennelli produced sound!
Carlos Terese produced sound!
Janet Terese produced sound!
Janet Ramon produced sound!
Arnold Tennelli played a false note. Uh oh
Arnold Tennelli sucks! kicked that member out of the band! (drum)
... <snip> ...
Musician Carlos Frizzle, playing the drum entered the room
... <snip for a few more firings> ...
Janet Jamal played a false note. Uh oh
Janet Jamal sucks! kicked that member out of the band! (drum)
The manager is mad and fired the whole band!
  Janet Ramon just got back to playing in the subway
The manager is mad and fired the whole band!
  Janet Terese just got back to playing in the subway
The manager is mad and fired the whole band!
  Carlos Terese just got back to playing in the subway
** exception error: shutdown
```

Magic! We can see that at first only the drummer is fired, and then after a while, everyone else gets kicked out, too. And off to the subway (tubes for the UK readers) they go!

You can try the code with other kinds of supervisors, and it will end the same. The only difference will be the restart strategy. Here's the angry supervisor at work:

```
5> band_supervisor:start_link(angry).
Musician Dorothy Frizzle, playing the singer entered the room
Musician Arnold Li, playing the bass entered the room
Musician Ralphie Perlstein, playing the drum entered the room
Musician Carlos Perlstein, playing the keytar entered the room
```

```
... <snip> ...
Ralphie Perlstein sucks! kicked that member out of the band! (drum)
... <snip> ...
The manager is mad and fired the whole band!
   Carlos Perlstein just got back to playing in the subway
```

With the angry supervisor, both the drummer and the keytar player get fired when the drummer makes a mistake. This is nothing compared to the jerk's behavior:

```
6> band_supervisor:start_link(jerk).
Musician Dorothy Franklin, playing the singer entered the room
Musician Wanda Tennelli, playing the bass entered the room
Musician Tim Perlstein, playing the drum entered the room
Musician Dorothy Frizzle, playing the keytar entered the room
... <snip> ...
Tim Perlstein played a false note. Uh oh
Tim Perlstein sucks! kicked that member out of the band! (drum)
The manager is mad and fired the whole band! Dorothy Franklin just got back to
playing in the subway
The manager is mad and fired the whole band! Wanda Tennelli just got back to
playing in the subway
The manager is mad and fired the whole band! Dorothy Frizzle just got back to
playing in the subway
```

And that's about it for static restart strategies.

Dynamic Supervision

So far, the kind of supervision we've covered has been static. We specified all the children we would have directly in the source code, and let everything run after that. This is how most of your supervisors might be set up in real-world applications, usually for the supervision of architectural components.

On the other hand, you may have supervisors who supervise undetermined workers. They're usually there on a per-demand basis. Think of a web server that spawns a process per connection it receives. In this case, you would want dynamic supervisors to look over all the different processes you'll have.

Using Standard Supervisors Dynamically

Every time a worker is added to a supervisor using the `one_for_one`, `rest_for_one`, or `one_for_all` strategy, the child specification is added to a list in the supervisor, along with a pid and some other information. The child specification can then be used to restart the child and perform other tasks. Because things work that way, the following interface exists:

`start_child(SupervisorNameOrPid, ChildSpec)`

Adds a child specification to the list and starts the child with it.

terminate_child(SupervisorNameOrPid, ChildId)
Terminates or `brutal_kill`s the child. The child specification is left in the supervisor.

restart_child(SupervisorNameOrPid, ChildId)
Uses the child specification to get things rolling.

delete_child(SupervisorNameOrPid, ChildId)
Gets rid of the `ChildSpec` of the specified child.

check_childspecs([ChildSpec])
Makes sure a child specification is valid. You can use this to try the specification before using `start_child/2`.

count_children(SupervisorNameOrPid)
Counts all the children under the supervisor and gives you a little comparative list of who is active, how many specs there are, how many are supervisors, and how many are workers.

which_children(SupervisorNameOrPid)
Gives you a list of all the children under the supervisor.

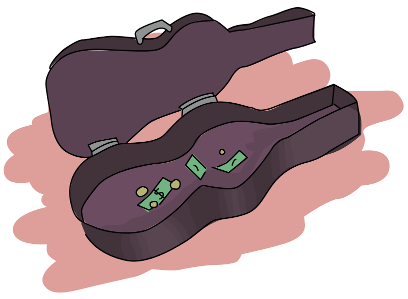
Let's see how this works with musicians, with the output removed (you need to be quick to outrace the failing drummer!).

```
1> band_supervisor:start_link(lenient).
{ok,0.709.0>}
2> supervisor:which_children(band_supervisor).
[{keytar,<0.713.0>,worker,[musicians]},
 {drum,<0.715.0>,worker,[musicians]},
 {bass,<0.711.0>,worker,[musicians]},
 {singer,<0.710.0>,worker,[musicians]}]
3> supervisor:terminate_child(band_supervisor, drum).
ok
4> supervisor:terminate_child(band_supervisor, singer).
ok
5> supervisor:restart_child(band_supervisor, singer).
{ok,<0.730.0>}
6> supervisor:count_children(band_supervisor).
[{specs,4},{active,3},{supervisors,0},{workers,4}]
7> supervisor:delete_child(band_supervisor, drum).
ok
8> supervisor:restart_child(band_supervisor, drum).
{error,not_found}
9> supervisor:count_children(band_supervisor).
[{specs,3},{active,3},{supervisors,0},{workers,3}]
```

And you can see how this could work well for anything dynamic that you need to manage (start, terminate, and so on) and when few children are involved. Because the internal representation is a list, this won't work well when you need quick access to many children.

In those cases, what you want is `simple_one_for_one`.

Using a simple_one_for_one Supervisor



With a supervisor that uses the `simple_one_for_one` strategy, all the children are held in a dictionary, which makes looking them up faster. There is also a single child specification for all children under the supervisor. This will save you memory and time—you will never need to delete a child yourself or store any child specifications.

For the most part, writing a `simple_one_for_one` supervisor is similar to writing any other type of supervisor, except for one thing: The argument list in the `{M,F,A}` tuple is not the whole thing, but will be appended to what you call it with when you do `supervisor:start_child(Sup, Args)`. That's right—`supervisor:start_child/2` changes meaning. So instead of doing `supervisor:start_child(Sup, Spec)`, which would call `erlang:apply(M,F,A)`, we now have `supervisor:start_child(Sup, Args)`, which calls `erlang:apply(M,F,Args++A)`.

We could use this strategy with our `band_supervisor` just by adding the following clause somewhere in it:

```
init(jamband) ->
    {ok, {{simple_one_for_one, 3, 60},
          [{jam_musician,
            {musicians, start_link, []},
            temporary, 1000, worker, [musicians]}
          ]}};
```

We've made all the musicians `temporary` in this case, and the supervisor is quite lenient:

```
1> supervisor:start_child(band_supervisor, [djembe, good]).
Musician Janet Tennelli, playing the djembe entered the room
{ok,<0.690.0>}
2> supervisor:start_child(band_supervisor, [djembe, good]).
{error,{already_started,<0.690.0>}}
```

Whoops! This happens because we register the djembe player as `djembe` as part of the start call to our `gen_server`. If we didn't name the child processes or used a different name for each, it wouldn't cause a problem. Here's one with the name `drum` instead:

```
3> supervisor:start_child(band_supervisor, [drum, good]).
Musician Arnold Ramon, playing the drum entered the room
{ok,<0.696.0>}
3> supervisor:start_child(band_supervisor, [guitar, good]).
Musician Wanda Perlstein, playing the guitar entered the room
{ok,<0.698.0>}
```

```
4> supervisor:terminate_child(band_supervisor, djembe).
ok
```

That seems right.

DON'T DRINK TOO MUCH KOOL-AID

Before Erlang version R14B03, it wasn't possible to terminate children with the function `supervisor:terminate_child(SupRef, Pid)`. The function would instead return `{error,simple_one_for_one}` and fail to terminate children. Instead, the following would have been the best way to terminate a child with a `simple_one_for_one` supervisor:

```
5> musicians:stop(drum).
Arnold Ramon left the room (drum)
ok
```

Backward-compatible code should take this kind of behavior into account.

As a general (though sometimes wrong) recommendation, use standard supervisors dynamically only when you know with certainty that you will have few children to supervise and/or they won't need to be manipulated frequently or with any high speed requirement. For other kinds of dynamic supervision, use `simple_one_for_one` where possible.

That's about it for the supervision strategies and child specifications. Right now, you might be having doubts and thinking. "How the hell am I going to get a working application out of that?" If that's the case, you'll be happy to get to Chapter 18, which actually builds a simple application with a short supervision tree to demonstrate how it could be done in the real world.

18

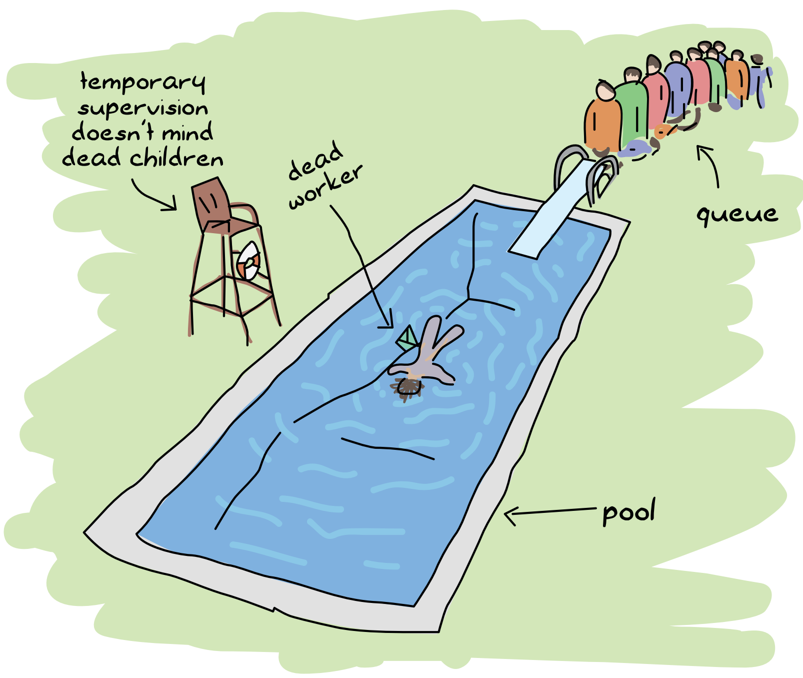
BUILDING AN APPLICATION

We've now covered how to use generic servers, FSMs, event handlers, and supervisors. However, we haven't gotten to how to put them all together to build complete applications and tools.

An Erlang application is a group of related code and processes. An OTP application specifically uses OTP behaviors for its processes, and then wraps them in a very specific structure that tells the VM how to set everything up and then tear it down. In this chapter, we're going to build an application with OTP components, although it won't be a full OTP application because we won't do the "wrapping up" just yet. The details of complete OTP applications are a bit complex and warrant their own chapter (the next one). This chapter is about using OTP components to implement an application, in our case, a process pool. The idea behind such a process pool is to manage and limit resources running in a system in a generic manner.

A Pool of Processes

A pool allows us to limit how many processes run at once. A pool can also queue up jobs when the running workers' limit is hit. The jobs can then be run as soon as resources are freed up, or they can simply block by telling the user they can't do anything else.



We might want to use process pools for several purposes, such as the following:

- Limit a server to at most *N* concurrent connections.
- Limit how many files can be opened by an application.
- Give different priorities to different subsystems of a release by allowing more resources for some and fewer for others. For example, you might want to allow more processes for client requests than processes in charge of generating reports for management.
- Allow an application under occasional heavy loads coming in bursts to remain more stable during its entire life by queuing the tasks.

The process pool application we'll build in this chapter will need to implement a few functions to handle the following:

- Start and stop the application.
- Start and stop a particular process pool (all the pools sit within the process pool application).
- Run a task in the pool and tell you it can't be started if the pool is full.

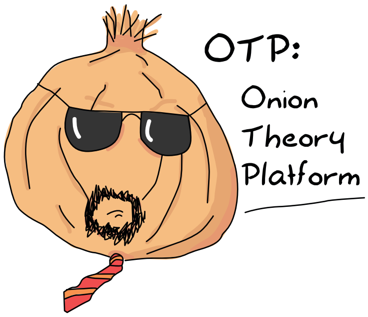
- Run a task in the pool if there is room; otherwise, keep the calling process waiting while the task is in the queue. Free the caller once the task can be run.
- Run a task asynchronously in the pool, as soon as possible. If no place is available, queue it up and run it whenever.

These needs will help drive our program design. Also keep in mind that we can now use supervisors, and, of course, we want to. However, though supervisors give us new powers in terms of robustness, they also impose a certain limit on flexibility. We'll explore that trade-off next.

The Onion Layer Theory

To help ourselves design an application with supervisors, it helps to have an idea of what needs supervision and how it needs to be supervised. As you'll recall from Chapter 17, we have different supervision strategies with different settings, which will fit for different kinds of code with different kinds of errors. A rainbow of mistakes can be made!

One thing newcomers and even experienced Erlang programmers have trouble dealing with is how to cope with the loss of state. Supervisors kill processes; state is lost; woe is me. To help with this, we will identify different kinds of states:



- A static state that can easily be fetched from a configuration file, another process, or the supervisor restarting the application.
- A dynamic state that is composed of data you can recompute. This includes state that you needed to transform from its initial form to get where it is right now.
- A dynamic state that you cannot recompute. This might include user input, live data, sequences of external events, and so on.

Static data is somewhat easy to deal with; most of the time, you can get it straight from the supervisor. The same is true for the dynamic but recomputable data. In this case, you might want to grab it and compute it within the `init/1` function (or anywhere else in your code, really). The most problematic kind of state is the dynamic data you can't recompute and that you just hope not to lose. In some cases, you'll be pushing that data to a database, although that won't always be a good option.

The idea of an onion-layered system is to allow all of these different states to be protected correctly by isolating different kinds of code from each other. In other words, it's process segregation. The static state can be handled by supervisors, as it is generally known as soon as the system starts up. Each time a child dies, the supervisor restarts it and can inject it with some form of static state, which is always available. Because most supervisor definitions are rather static by nature, each layer of supervision you add acts as a shield protecting your application against their failure and the loss of their state.

The dynamic state that can be recomputed has a whole lot of available solutions. For example, you can build it from the static data sent by the supervisors, or you could go fetch it back from some other process, database, text file, the current environment, or whatever. It should be relatively easy to get the data back on each restart. The fact that you have supervisors that do a restarting job can be enough to help you keep that state alive.

The dynamic non-recomputable kind of state needs a more thoughtful solution. The real nature of an onion-layered approach takes shape here. The idea is that the most important data (or the data that is most annoying to lose) must be the most protected type. The place where you are actually not allowed to fail is called the *error kernel* of your application.

The error kernel is likely the place you'll want to use `try ... catch` expressions more than anywhere else, since handling exceptional cases is vital there. This is the area that you want to be error-free. Careful testing must be done around the error kernel, especially in cases where there is no way to go back. You don't want to lose a customer's order halfway through processing it, do you?

Some operations are going to be considered safer than others. Because of this, we want to keep vital data in the safest core possible and keep everything somewhat dangerous outside it. In specific terms, this means that all related operations should be part of the same supervision trees, and the unrelated ones should be kept in different trees. Within the same tree, operations that are more failure-prone can be placed deeper in the tree, and the processes that cannot afford to crash are closer to the root of the tree.

These principles result in systems where all related pieces of software are part of the same trees, with the riskiest operations low in the tree, decreasing the risk of the core processes dying until the system can't cope with the errors properly anymore. We'll see an example of this when designing our actual process pool's supervision tree.

A Pool's Tree

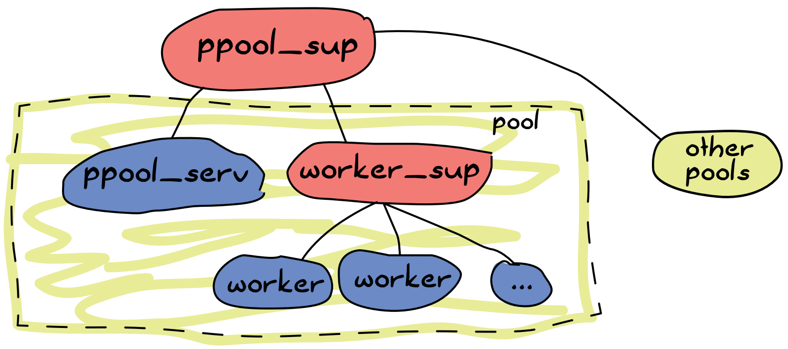
So how should we organize these process pools? There are two schools of thought here. One tells people to design bottom-up (write all individual components, and put them together as required), and another one tells us to write things top-down (design as if all the parts were there, and then build them). Both approaches are equally valid depending on the circumstances and your personal style. For the sake of making things understandable, we're going to do things top-down here.

So what should our tree look like? Well, our requirements include being able to start the pool application as a whole, having many pools, and having many workers that can be queued for each pool. This already suggests a few possible design constraints.

We will need one `gen_server` per pool. The server's job will be to maintain the counter of how many workers are in the pool. For convenience, the same server should also hold the queue of tasks. Who should be in charge of overlooking each of the workers, though? The server itself?

Doing it with the server is interesting. After all, the server needs to track the processes to count them, and supervising them itself is a nifty way to do this. Moreover, neither the server nor the processes can crash without losing the state of all the others (otherwise, the server can't track the tasks after it has restarted). It has a few disadvantages, too: The server has many responsibilities, can be seen as more fragile, and duplicates the functionality of existing, better-tested modules.

A good way to make sure all workers are properly accounted for would be to use a supervisor just for them:

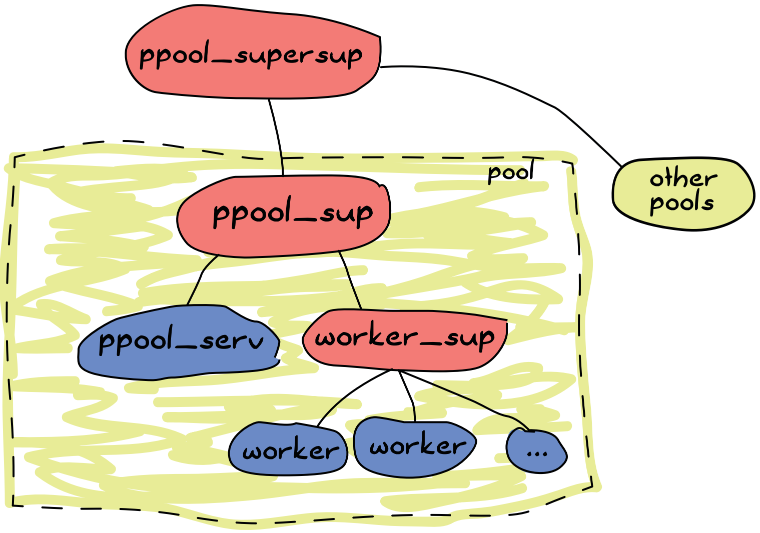


In this example, there is a single supervisor for all of the pools. Each pool is a set of a pool server and a supervisor for workers. The pool server knows of the existence of its worker supervisor and asks it to add items. Given that adding children is a very dynamic thing with unknown limits so far, we'll use a `simple_one_for_one` supervisor.

NOTE *The name* `ppool` *was chosen because the Erlang standard library already has a* `pool` *module. Plus it's a terrible pool-related pun.*

The advantage of this approach is that because the `worker_sup` supervisor will need to track only OTP workers of a single type, each pool is guaranteed to be about a well-defined kind of worker, with simple management and restart strategies that are easy to define. This is one example of an error kernel being better defined. If we're using a pool of sockets for web connections and another pool of servers in charge of log files, we are making sure that incorrect code or messy permissions in the log file section of our application won't be drowning out the processes in charge of the sockets. If the log files' pool crashes too much, they will be shut down and their supervisor will stop. Oh wait—their supervisor stopping is a serious problem!

Because all pools are under the same supervisor, a given pool or server restarting too many times in a short time span can take all the other pools down. A solution might be to add one level of supervision. This will also make it much simpler to handle more than one pool at a time, so let's say the following will be our application architecture:



And that makes a bit more sense. From the onion-layer perspective, all pools are independent, the workers are independent from each other and the `ppool_serv` server is going to be isolated from all the workers. That's good enough for the architecture. Everything we need seems to be there. We can start working on the implementation—again, using a top-to-bottom approach.

Implementing the Supervisors

We can start with the top-level supervisor, `ppool_supersup`. All this one needs to do is start the supervisor of a pool when required. We'll give it a few functions: `start_link/0`, which starts the whole application; `stop/0`, which stops it; `start_pool/3`, which creates a specific pool; and `stop_pool/1`, which does the opposite. We also can't forget `init/1`, which is the only callback required by the supervisor behavior.

```
-module(ppool_supersup).
-behavior(supervisor).
-export([start_link/0, stop/0, start_pool/3, stop_pool/1]).
-export([init/1]).
```

```
start_link() ->
    supervisor:start_link({local, ppool}, ?MODULE, []).
```

Here, we gave the top-level process pool supervisor the name ppool (this explains the use of {local, Name}, an OTP convention about registering gen_* processes on a node; another one exists for distributed registration). This is because we know we will have only one ppool per Erlang node, and we can give it a name without worrying about clashes. Fortunately, the same name can then be used to stop the whole set of pools, like this:

```
%% Technically, a supervisor cannot be killed in an easy way.
%% Let's do it brutally!
stop() ->
    case whereis(ppool) of
        P when is_pid(P) ->
            exit(P, kill);
        _ -> ok
    end.
```

As the comments in the code explain, we cannot terminate a supervisor gracefully. The OTP framework provides a well-defined shutdown procedure for all supervisors, but we can't use it from where we are right now. We'll address how to do that in Chapter 19; but for now, brutally killing the supervisor is the best we can do.

What is the top-level supervisor exactly? Well, its only task is to hold pools in memory and supervise them. In this case, it will be a childless supervisor.

```
init([]) ->
    MaxRestart = 6,
    MaxTime = 3600,
    {ok, {{one_for_one, MaxRestart, MaxTime}, []}}.
```

We can now focus on starting each individual pool's supervisor and attaching them to ppool. Given our initial requirements, we can determine that we'll need two parameters: the number of workers the pool will accept and the {M,F,A} tuple that the worker supervisor will need to start each worker. We'll also add a name for good measure. We then pass this ChildSpec to the process pool's supervisor as we start it.

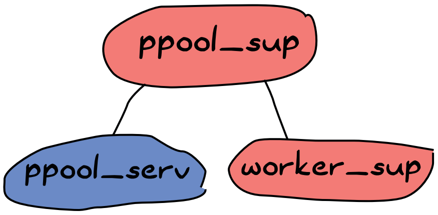
```
start_pool(Name, Limit, MFA) ->
    ChildSpec = {Name,
                 {ppool_sup, start_link, [Name, Limit, MFA]},
                 permanent, 10500, supervisor, [ppool_sup]},
    supervisor:start_child(ppool, ChildSpec).
```

You can see each pool supervisor is asked to be permanent and has the arguments needed (notice how we're changing programmer-submitted data into static data this way). The name of the pool is both passed to the supervisor and used as an identifier in the child specification. There's also a maximum shutdown time of 10500. There is no easy way to pick this value—just make sure it's large enough that all the children will have time to stop, if they need any. Play with the value according to your needs, and test and adapt to your application. If you're really not sure what value to use, you can try the infinity option.

To stop the pool, we need to ask the ppool super supervisor (the *supersup*!) to kill its matching child.

```
stop_pool(Name) ->
    supervisor:terminate_child(ppool, Name),
    supervisor:delete_child(ppool, Name).
```

This is possible because we gave the pool's name as the child specification identifier. Great! We can now focus on each pool's direct supervisor! Each ppool_sup will be in charge of the pool server and the worker supervisor:



Can you see the funny thing here? The ppool_serv process should be able to contact the worker_sup process. If they're both started by the same supervisor at the same time, we won't have any way to let ppool_serv know about worker_sup, unless we were to do some trickery with supervisor:which_children/1 (which would be sensitive to timing and somewhat risky), or unless we give a name to both the ppool_serv process (so that users can call it) and the supervisor. However, we don't want to give names to the supervisors for several reasons:

- The users don't need to call them directly.
- We would need to dynamically generate atoms, and that makes me nervous.
- There is a better way.

The solution is to get the pool server to dynamically attach the worker supervisor to its ppool_sup. Don't worry if this sounds vague—you'll get it soon. For now, we only start the server.

```
-module(ppool_sup).
-export([start_link/3, init/1]).
-behavior(supervisor).

start_link(Name, Limit, MFA) ->
    supervisor:start_link(?MODULE, {Name, Limit, MFA}).
```

```
init({Name, Limit, MFA}) ->
    MaxRestart = 1,
    MaxTime = 3600,
    {ok, {{one_for_all, MaxRestart, MaxTime},
          [{serv,
            {ppool_serv, start_link, [Name, Limit, self(), MFA]},
            permanent,
            5000, % Shutdown time
            worker,
            [ppool_serv]}]}}.
```

And that's about it. Note that the Name is passed to the server, along with self(), the supervisor's own pid. This will let the server call for the spawning of the worker supervisor; the MFA variable will be used in that call to let the simple_one_for_one supervisor know which kind of workers to run.

We'll get to how the server handles everything in the next section. For now, we'll finish creating all of the application's supervisors by writing ppool_worker_sup, which is in charge of all the workers.

```
-module(ppool_worker_sup).
-export([start_link/1, init/1]).
-behavior(supervisor).

start_link(MFA = {_,_,_}) ->
    supervisor:start_link(?MODULE, MFA).

init({M,F,A}) ->
    MaxRestart = 5,
    MaxTime = 3600,
    {ok, {{simple_one_for_one, MaxRestart, MaxTime},
          [{ppool_worker,
            {M,F,A},
            temporary, 5000, worker, [M]}]}}.
```

This is simple stuff. We picked a simple_one_for_one supervisor because workers could be added in very high numbers with a requirement for speed, plus we want to restrict their type. All the workers are temporary, and because we use an {M,F,A} tuple to start the worker, we can use any kind of OTP behavior there.

The reason to make the workers temporary is twofold. First, we cannot know for sure whether they need to be restarted in case of failure, or what kind of restart strategy would be required for them. Second, the pool might be useful only if the worker's creator can have access to the worker's pid, depending on the use case. For this to work in any safe and simple manner, we can't just restart workers as we please without tracking their creator and sending it a notification. This would make things quite complex just to grab a pid. Of course, you are free to write your own ppool_worker_sup that doesn't return pids but restarts them. There's nothing inherently wrong in that design.

Working on the Workers

The pool server is the most complex part of the application, where all the clever business logic happens. Here's a reminder of the operations we must support:

- Running a task in the pool and indicating that it can't be started if the pool is full
- Running a task in the pool if there's room; otherwise, keeping the calling process waiting while the task is in the queue, until it can be run
- Running a task asynchronously in the pool, as soon as possible; if no place is available, queuing it up and running it whenever

The first operation will be done by a function named run/2, the second by sync_queue/2, and the last one by async_queue/2.

```
-module(ppool_serv).
-behavior(gen_server).
-export([start/4, start_link/4, run/2, sync_queue/2, async_queue/2, stop/1]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         code_change/3, terminate/2]).

start(Name, Limit, Sup, MFA) when is_atom(Name), is_integer(Limit) ->
    gen_server:start({local, Name}, ?MODULE, {Limit, MFA, Sup}, []).

start_link(Name, Limit, Sup, MFA) when is_atom(Name), is_integer(Limit) ->
    gen_server:start_link({local, Name}, ?MODULE, {Limit, MFA, Sup}, []).

run(Name, Args) ->
    gen_server:call(Name, {run, Args}).

sync_queue(Name, Args) ->
    gen_server:call(Name, {sync, Args}, infinity).

async_queue(Name, Args) ->
    gen_server:cast(Name, {async, Args}).
```

```
stop(Name) ->
    gen_server:call(Name, stop).
```

For `start/4` and `start_link/4`, `Args` will be the additional arguments passed to the `A` part of the `{M,F,A}` tuple sent to the supervisor. Note that for the synchronous queue, we've set the waiting time to `infinity`.

As mentioned earlier, we must start the supervisor from within the server. If you're adding the code as we go, you might want to include an empty `gen_server` template (or use the completed file available online) to follow along, because we'll do things on a per-feature basis, rather than from top to bottom.

The first thing we do is handle the creation of the supervisor. As discussed in "Dynamic Supervision" on page 277, we do not need a `simple_one_for_one` for cases where only a few children will be added, so `supervisor:start_child/2` ought to do it. We'll first define the child specification of the worker supervisor.

```
%% The friendly supervisor is started dynamically!
-define(SPEC(MFA),
        {worker_sup,
         {ppool_worker_sup, start_link, [MFA]},
         permanent,
         10000,
         supervisor,
         [ppool_worker_sup]}).
```

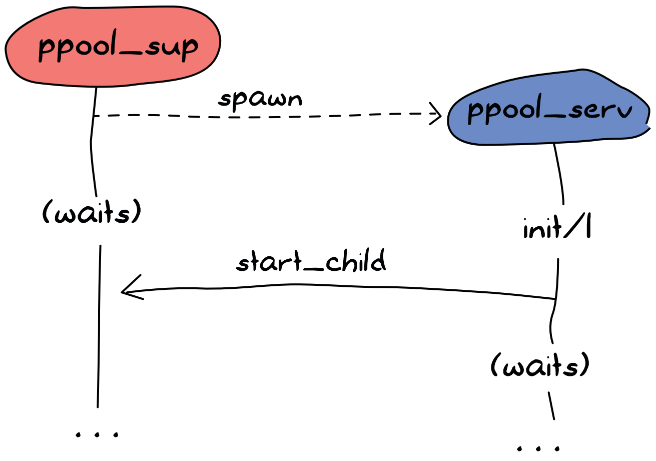
We can then define the inner state of the server. We know we will need to track a few pieces of data: the number of processes that can be running, the pid of the supervisor, and a queue for all the jobs. To know when a worker's finished running and to fetch one from the queue to start it, we will need to track each worker from the server. The sane way to do this is with monitors, so we'll also add a `refs` field to our state record to keep all the monitor references in memory.

```
-record(state, {limit=0,
                sup,
                refs,
                queue=queue:new()}).
```

With this ready, we can start implementing the `init` function. The natural thing to try is this:

```
init({Limit, MFA, Sup}) ->
    {ok, Pid} = supervisor:start_child(Sup, ?SPEC(MFA)),
    {ok, #state{limit=Limit, refs=gb_sets:empty()}}.
```

However, this code is wrong. With gen_* behaviors, the process that spawns the behavior waits until the init/1 function returns before resuming its processing. This means that by calling supervisor:start_child/2 in there, we create the following deadlock:



Both processes will keep waiting for each other until there is a crash. The cleanest way to get around this is to create a special message that the server will send to itself and later handle in handle_info/2 as soon as the init function has returned (and the pool supervisor has become free):

```
init({Limit, MFA, Sup}) ->
    %% We need to find the Pid of the worker supervisor from here,
    %% but alas, this would be calling the supervisor while it waits for us!
    self() ! {start_worker_supervisor, Sup, MFA},
    {ok, #state{limit=Limit, refs=gb_sets:empty()}}.
```

This one is cleaner. We can then head out to the handle_info/2 function and add the following clauses:

```
handle_info({start_worker_supervisor, Sup, MFA}, S = #state{}) ->
    {ok, Pid} = supervisor:start_child(Sup, ?SPEC(MFA)),
    {noreply, S#state{sup=Pid}};
handle_info(Msg, State) ->
    io:format("Unknown msg: ~p~n", [Msg]),
    {noreply, State}.
```

The first clause is the interesting one here. We find the message we sent ourselves (which will necessarily be the first one received), ask the pool supervisor to add the worker supervisor, track this pid, and *voilà*! Our tree is now fully initialized. You can try compiling everything to make sure no mistake has been made so far. Unfortunately, we still can't test the application because too much stuff is missing.

NOTE *Don't worry if you do not like the idea of building the whole application before running it. Things are being done this way to show a cleaner reasoning of the whole thing. While I did have the general design in mind (the same one I illustrated earlier), I started writing this pool application in a little test-driven manner, with a few tests here and there and a bunch of refactoring to get everything to a functional state. Few Erlang programmers (much like programmers of most other languages) will be able to produce production-ready code on their first try, and the author is not as clever as the examples might make him seem.*

Now that we've solved this bit, we'll take care of the `run/2` function. This one is a synchronous call with the message of the form `{run, Args}` and works as follows:

```
handle_call({run, Args}, _From, S = #state{limit=N, sup=Sup, refs=R}) when N > 0 ->
    {ok, Pid} = supervisor:start_child(Sup, Args),
    Ref = erlang:monitor(process, Pid),
    {reply, {ok,Pid}, S#state{limit=N-1, refs=gb_sets:add(Ref,R)}};
handle_call({run, _Args}, _From, S=#state{limit=N}) when N =< 0 ->
    {reply, noalloc, S};
```

We have a long function head, but we can see most of the management taking place there. Whenever there are places left in the pool (the original limit `N` being decided by the programmer adding the pool in the first place), we accept to start the worker. We then set up a monitor to know when it's done, store all of this in our state, decrement the counter, and off we go. In the case no space is available, we simply reply with `noalloc`.

The calls to `sync_queue/2` will give a very similar implementation:

```
handle_call({sync, Args}, _From, S = #state{limit=N, sup=Sup, refs=R}) when N > 0 ->
    {ok, Pid} = supervisor:start_child(Sup, Args),
    Ref = erlang:monitor(process, Pid),
    {reply, {ok,Pid}, S#state{limit=N-1, refs=gb_sets:add(Ref,R)}};
handle_call({sync, Args},  From, S = #state{queue=Q}) ->
    {noreply, S#state{queue=queue:in({From, Args}, Q)}};
```

If there is space for more workers, then the first clause will do exactly the same thing as we did for `run/2`. The difference comes in the case where no workers can run. Rather than replying with `noalloc` as we did with `run/2`, this one doesn't reply to the caller, keeps the `From` information, and queues it for a later time when there is space for the worker to be run. We'll see how we dequeue workers and handle them soon enough, but for now, we'll finish the `handle_call/3` callback with the following clauses:

```
handle_call(stop, _From, State) ->
    {stop, normal, ok, State};
handle_call(_Msg, _From, State) ->
    {noreply, State}.
```

These handle the unknown cases and the stop/1 call. We can now focus on getting async_queue/2 working. Because async_queue/2 basically does not care when the worker is run and expects absolutely no reply, we'll make it a cast rather than a call. You'll find the logic of it very similar to the two previous options.

```
handle_cast({async, Args}, S=#state{limit=N, sup=Sup, refs=R}) when N > 0 ->
    {ok, Pid} = supervisor:start_child(Sup, Args),
    Ref = erlang:monitor(process, Pid),
    {noreply, S#state{limit=N-1, refs=gb_sets:add(Ref,R)}};
handle_cast({async, Args}, S=#state{limit=N, queue=Q}) when N =< 0 ->
    {noreply, S#state{queue=queue:in(Args,Q)}};
%% Not going to explain the one below!
handle_cast(_Msg, State) ->
    {noreply, State}.
```

Again, the only big difference apart from not replying is that when there is no place left for a worker, it is queued. This time though, we have no From information and just send the worker to the queue without it. The limit doesn't change in this case.

When do we know it's time to dequeue something? Well, we have monitors set all over the place, and we're storing their references in a gb_sets. Whenever a worker goes down, we're notified of it. Let's work from there.

```
handle_info({'DOWN', Ref, process, _Pid, _}, S = #state{limit=L, sup=Sup, refs=Refs}) ->
    io:format("received down msg~n"),
    case gb_sets:is_element(Ref, Refs) of
        true ->
            handle_down_worker(Ref, S);
        false -> %% not our responsibility
            {noreply, S}
    end;
handle_info({start_worker_supervisor, Sup, MFA}, S = #state{}) ->
    ...
handle_info(Msg, State) ->
    ...
```

In this snippet, we make sure the 'DOWN' message we get comes from a worker. If it doesn't come from one (which would be surprising), we just ignore it. However, if the message really is what we want, we call a function named handle_down_worker/2:

```
handle_down_worker(Ref, S = #state{limit=L, sup=Sup, refs=Refs}) ->
    case queue:out(S#state.queue) of
        {{value, {From, Args}}, Q} ->
            {ok, Pid} = supervisor:start_child(Sup, Args),
            NewRef = erlang:monitor(process, Pid),
            NewRefs = gb_sets:insert(NewRef, gb_sets:delete(Ref,Refs)),
            gen_server:reply(From, {ok, Pid}),
            {noreply, S#state{refs=NewRefs, queue=Q}};
        {{value, Args}, Q} ->
            {ok, Pid} = supervisor:start_child(Sup, Args),
            NewRef = erlang:monitor(process, Pid),
            NewRefs = gb_sets:insert(NewRef, gb_sets:delete(Ref,Refs)),
            {noreply, S#state{refs=NewRefs, queue=Q}};
        {empty, _} ->
            {noreply, S#state{limit=L+1, refs=gb_sets:delete(Ref,Refs)}}
    end.
```

This is quite a complex function. Because our worker is dead, we can look in the queue for the next one to run. We do this by popping one element out of the queue and looking at the result. If there is at least one element in the queue, it will be of the form `{{value, Item}, NewQueue}`. If the queue is empty, it returns `{empty, SameQueue}`. Furthermore, we know that when we have the value `{From, Args}`, it means this came from `sync_queue/2`; otherwise, it came from `async_queue/2`.

Both cases where the queue has tasks in it will behave roughly the same: A new worker is attached to the worker supervisor, and the reference of the old worker's monitor is removed and replaced with the new worker's monitor reference. The only different aspect is that in the case of the synchronous call, we send a manual reply, and in the other case, we can remain silent. In the case the queue was empty, we need to do nothing but increment the worker limit by one.

The last thing to do is add the standard OTP callbacks:

```
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

terminate(_Reason, _State) ->
    ok.
```

That's it—our pool is ready to be used! It is a very unfriendly pool, though. All the functions we need to use are scattered throughout the code. Some are in `ppool_supersup`; some are in `ppool_serv`. Also, the module names are long for no reason. To make things nicer, add the following API module (just abstracting the calls away) to the application's directory:

```
%%% API module for the pool
-module(ppool).
-export([start_link/0, stop/0, start_pool/3,
         run/2, sync_queue/2, async_queue/2, stop_pool/1]).

start_link() ->
    ppool_supersup:start_link().

stop() ->
    ppool_supersup:stop().
```

```
start_pool(Name, Limit, {M,F,A}) ->
    ppool_supersup:start_pool(Name, Limit, {M,F,A}).

stop_pool(Name) ->
    ppool_supersup:stop_pool(Name).

run(Name, Args) ->
    ppool_serv:run(Name, Args).

async_queue(Name, Args) ->
    ppool_serv:async_queue(Name, Args).

sync_queue(Name, Args) ->
    ppool_serv:sync_queue(Name, Args).
```

And now we're finished with our pool for real!

POOL LIMITS

You'll have noticed that our process pool doesn't limit the number of items that can be stored in the queue. In some cases, a real server application will need to put a ceiling on how many things can be queued to avoid crashing when too much memory is used, although the problem can be circumvented if you use `run/2` and `sync_queue/2` only with a fixed number of callers (if all content producers are stuck waiting for free space in the pool, they stop producing so much content in the first place).

Adding a limit to the queue size is left as an exercise to the reader, but fear not because it is relatively simple to do. You will need to pass a new parameter to all functions up to the server, which will then check the limit before any queuing.

Additionally, to control the load of your system, you sometimes want to impose limits closer to their source by using synchronous calls. Synchronous calls allow you to block incoming queries when the system is getting swamped by producers faster than consumers. This approach generally helps keep it more responsive than a free-for-all load.

Writing a Worker

Look at me go—I'm lying all the time! The pool isn't really ready to be used. We don't have a worker at the moment. I forgot. This is a shame because we all know that in the chapter about writing a concurrent application (Chapter 13), we built a nice task reminder. It apparently isn't enough for me, so for this example, we'll write a *nagger*.

It will basically be a worker for each task, and the worker will keep nagging us by sending repeated messages until a given deadline.

The nagger will be able to take the following elements:

- A time delay for which to nag
- An address (pid) to say where the messages should be sent
- A nagging message to be sent in the process mailbox, including the nagger's own pid to be able to call
- A stop function to say the task is done and that the nagger can stop nagging

Here we go:

```
%% demo module, a nagger for tasks,
%% because the previous one wasn't good enough
-module(ppool_nagger).
-behavior(gen_server).
-export([start_link/4, stop/1]).
-export([init/1, handle_call/3, handle_cast/2,
         handle_info/2, code_change/3, terminate/2]).

start_link(Task, Delay, Max, SendTo) ->
    gen_server:start_link(?MODULE, {Task, Delay, Max, SendTo} , []).

stop(Pid) ->
    gen_server:call(Pid, stop).
```

Yes, we're going to be using yet another `gen_server`. You'll find out that people use them all the time—sometimes even when not appropriate! It's important to remember that our pool can accept any OTP-compliant process, not just a `gen_server`.

```
init({Task, Delay, Max, SendTo}) ->
    {ok, {Task, Delay, Max, SendTo}, Delay}.
```

This just takes the basic data and forwards it:

- `Task` is the thing to send as a message.
- `Delay` is the time spent in between each sending.
- `Max` is the number of times it's going to be sent.
- `SendTo` is a pid or a name where the message will go.

Note that `Delay` is passed as a third element of the tuple, which means the timeout will be sent to `handle_info/2` after `Delay` milliseconds.

Given our API, most of the server is rather straightforward.

```
%%% OTP Callbacks
handle_call(stop, _From, State) ->
    {stop, normal, ok, State};
```

```
handle_call(_Msg, _From, State) ->
    {noreply, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(timeout, {Task, Delay, Max, SendTo}) ->
    SendTo ! {self(), Task},
    if Max =:= infinity ->
        {noreply, {Task, Delay, Max, SendTo}, Delay};
       Max =< 1 ->
        {stop, normal, {Task, Delay, 0, SendTo}};
       Max > 1  ->
        {noreply, {Task, Delay, Max-1, SendTo}, Delay}
    end.
%% We cannot use handle_info below: if that ever happens,
%% we cancel the timeouts (Delay) and basically zombify
%% the entire process. It's better to crash in this case.
%% handle_info(_Msg, State) ->
%%    {noreply, State}.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

terminate(_Reason, _State) -> ok.
```

The only somewhat complex part here lies in the `handle_info/2` function. As seen back in Chapter 14's introduction to `gen_server`, every time a timeout is hit (in this case, after `Delay` milliseconds), the `timeout` message is sent to the process. Based on this, we check how many nags were sent to know if we should send more or just quit.

With this worker complete, we can actually try this process pool!

Run Pool Run

We can now play with the pool. Compile all the files and start the pool top-level supervisor itself:

```
$ erlc *.erl
$ erl
... <snip> ...
1> ppool:start_link().
{ok,<0.33.0>}
```

From this point, we can try a bunch of different features of the nagger as a pool:

```
2> ppool:start_pool(nagger, 2, {ppool_nagger, start_link, []}).
{ok,<0.35.0>}
```

```
3> ppool:run(nagger, ["finish the chapter!", 10000, 10, self()]).
{ok,<0.39.0>}
4> ppool:run(nagger, ["Watch a good movie", 10000, 10, self()]).
{ok,<0.41.0>}
5> flush().
Shell got {<0.39.0>,"finish the chapter!"}
Shell got {<0.39.0>,"finish the chapter!"}
ok
6> ppool:run(nagger, ["clean up a bit", 10000, 10, self()]).
noalloc
7> flush().
Shell got {<0.41.0>,"Watch a good movie"}
Shell got {<0.39.0>,"finish the chapter!"}
Shell got {<0.41.0>,"Watch a good movie"}
Shell got {<0.39.0>,"finish the chapter!"}
Shell got {<0.41.0>,"Watch a good movie"}
... <snip> ...
```

Everything seems to work rather well for the synchronous nonqueued runs. The pool is started, tasks are added, and messages are sent to the right destination. When we try to run more tasks than allowed, allocation is denied to us. No time for cleaning up, sorry! The others still run fine though.

NOTE *The* `ppool` *is started with* `start_link/0`*. If at any time you make an error in the shell, you take down the whole pool and need to start over again. This issue will be addressed in Chapter 19.*

Now let's try the queuing facilities (asynchronous):

```
8> ppool:async_queue(nagger, ["Pay the bills", 30000, 1, self()]).
ok
9> ppool:async_queue(nagger, ["Take a shower", 30000, 1, self()]).
ok
10> ppool:async_queue(nagger, ["Plant a tree", 30000, 1, self()]).
ok
❶ ... <wait a bit> ...
received down msg
received down msg
11> flush().
Shell got {<0.70.0>,"Pay the bills"}
Shell got {<0.72.0>,"Take a shower"}
❷ ... <wait some more> ...
received down msg
12> flush().
Shell got {<0.74.0>,"Plant a tree"}
ok
```

Great! So the queuing works. The log here doesn't show everything in a very clear manner (although you should wait at ❶ and ❷ for best effect).

What happens is that the two first naggers run as soon as possible. Then the worker limit is hit, and we need to queue the third one (planting a tree). When the nags for paying the bills are finished, the tree nagger is scheduled and sends the message a bit later.

The synchronous one will behave differently:

```
13> ppool:sync_queue(nagger, ["Pet a dog", 20000, 1, self()]).
{ok,<0.108.0>}
14> ppool:sync_queue(nagger, ["Make some noise", 20000, 1, self()]).
{ok,<0.110.0>}
15> ppool:sync_queue(nagger, ["Chase a tornado", 20000, 1, self()]).
received down msg
{ok,<0.112.0>}
received down msg
16> flush().
Shell got {<0.108.0>,"Pet a dog"}
Shell got {<0.110.0>,"Make some noise"}
ok
received down msg
17> flush().
Shell got {<0.112.0>,"Chase a tornado"}
ok
```

Again, the log isn't as clear as if you tried it yourself (which is recommended). The basic sequence of events is that two workers are added to the pool. They aren't finished running, and when we try to add a third one, the shell gets locked up until `ppool_serv` (under the process name `nagger`) receives a worker's down message (`received down msg`). After this, our call to `sync_queue/2` can return and give us the pid of our brand-new worker.

We can now get rid of the pool as a whole:

```
18> ppool:stop_pool(nagger).
ok
19> ppool:stop().
** exception exit: killed
```

All pools will be terminated if you decide to just call `ppool:stop()`, but you'll receive a bunch of error messages. This is because we brutally kill the `ppool_supersup` process, rather than taking it down correctly, which in turns crashes all child pools. In Chapter 19, I will cover how to terminate the pool cleanly.

Cleaning the Pool

In this chapter, we managed to write a process pool to do some resource allocation in a somewhat simple manner. Everything can be handled in parallel, can be limited, and can be called from other processes. Pieces of your application that crash can, with the help of supervisors, be replaced transparently without breaking the entirety of it. Once the pool application was ready, we even rewrote a surprisingly large part of our reminder app with very little code.

Failure isolation for a single computer has been taken into account, and concurrency is handled. We now have enough architectural blocks to write some pretty solid server-side software, even though we still haven't really covered good ways to run them from the shell.

Chapter 19 will show how to package the `ppool` application into a real OTP application, ready to be shipped and used by other products. Although we haven't explored all the advanced features of OTP, you're now on a level where you should be able to understand most intermediate to somewhat advanced discussions on OTP and Erlang (the nondistributed part, at least). That's pretty good!