19

BUILDING APPLICATIONS THE OTP WAY

After seeing our whole application's supervision tree start at once with a simple function call, you might wonder why we would want to package it in any way. What could be simpler than a single function call?

The concepts behind supervision trees are a bit complex already, and I could see myself just starting all of these trees and subtrees manually with a script when the system is first set up. Then after that, I would be free to go outside and try to find clouds that look like animals for the rest of the afternoon.

This is entirely true, yes. This is an acceptable way to do things (especially the part about clouds, because these days, everything is about

cloud computing). However, like most abstractions made by programmers and engineers, OTP applications are the result of many ad hoc systems being generalized and made clean.

If you were to make an array of scripts and commands to start your supervision trees, and other developers you work with had their own, you would quickly run into massive issues. Then someone would ask something like, "Wouldn't it be nice if everyone used the same kind of system to start everything? And wouldn't it even be nicer if they all had the same kind of application structure?"

OTP applications attempt to solve this type of problem by providing the following:

- A directory structure
- A way to handle configurations
- A way to create environment variables and configurations
- Ways to start and stop applications while respecting dependencies
- A lot of safe control in detecting conflicts and handling live upgrades without shutting your applications down

So unless you don't want these aspects (nor the niceties they give, like consistent structures and tools), this chapter should be of some interest to you, as it introduces all the necessary concepts to get a good understanding of OTP applications.

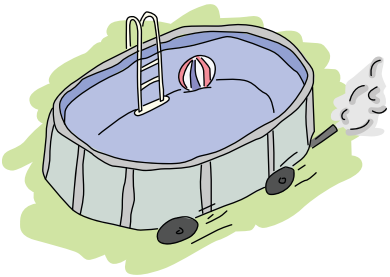
My Other Car Is a Pool

Merely using OTP components isn't enough to guarantee we're creating an *OTP application*, much like putting pieces of humans together won't guarantee you get a human being instead of some kind of Frankenstein's monster. We're going to reuse the `ppool` application we wrote in Chapter 18 and turn it into a proper OTP application.

The first step in doing so is to copy all the `ppool`-related files into a neat directory structure:

```
ebin/
include/
priv/
src/
 - ppool.erl
 - ppool_sup.erl
 - ppool_supersup.erl
 - ppool_worker_sup.erl
 - ppool_serv.erl
 - ppool_nagger.erl
test/
 - ppool_tests.erl
```

Most directories will remain empty for now. As explained in Chapter 13, the *ebin/* directory will hold compiled files; the *include/* directory will contain public Erlang header (*.hrl*) files; *priv/* will hold executables, other programs, and various specific files needed for the application to work; and *src/* will hold the Erlang source files you will need (as well as private *.hrl* files).



You'll note that we added a *test/* directory, which holds the test file *ppool_tests.erl* that I wrote for Chapter 18, if you downloaded the related code. Tests are somewhat common, but you don't necessarily want them distributed as part of your application. You just need the tests when developing your code and justifying yourself to your manager ("Tests pass. I don't understand why the app killed people."). Other directories end up being added as required, depending on the case. One example is the *doc/* directory, created whenever you have EDoc documentation (which is a way to annotate Erlang code to generate documentation) to add to your application. For more information on EDoc, see *http://www.erlang.org/doc/apps/edoc/chapter.html*.

The four basic directories to have are *ebin/*, *include/*, *priv/*, and *src/*. These are common to pretty much every OTP application, although only *ebin/* and *priv/* will be exported when real OTP systems are deployed.

The Application Resource File

Where do we go from here? Well, the first thing to do is add an application file. This file will tell the Erlang VM what the application is, where it begins, and where it ends. This file lives in the *ebin/* directory, along with all the compiled modules.

This file is usually named *yourapp.app* (in our case *ppool.app*) and contains a bunch of Erlang terms defining the application in a way that the VM can understand. (The VM is pretty bad at guessing stuff!)

NOTE *Some people prefer to keep the application file outside* ebin/ *and instead have a file named* myapp.app.src *as part of* src/*. Whatever build system they use then copies this file over to* ebin/ *or even generates an app file in order to keep everything clean.*

The basic structure of the application file is simply as follows:

```
{application, ApplicationName, Properties}.
```

`ApplicationName` is an atom, and `Properties` is a list of `{Key, Value}` tuples describing the application. They're used by OTP to figure out what your application does. They're all optional, but they can be useful to debug running systems and make sure different applications interact in an orderly

manner. They're also necessary for some tools. We'll look at a subset of them for now, and introduce the others as we need them.

`{description, "Some description of your application"}`
: This gives the system a short description of the application. The field defaults to an empty string. Although this field is optional, I suggest always defining a description, if only because it makes things easier to read.

`{vsn, "1.2.3"}`
: This is the version of your application. This string can take any format you want. It's usually a good idea to stick to a scheme of the form *`Major.Minor.Patch`*, or something similar. When you start using tools to help with upgrades and downgrades, this string is used to identify your application's version.

`{modules, ModuleList}`
: This contains a list of all the modules that your application introduces to the system. A module always belongs to at most one application and cannot be present in two applications' app files at once. This list lets the system and tools look at dependencies of your application, making sure everything is where it needs to be and that you have no conflicts with other applications already loaded in the system. If you're using a standard OTP structure and are using an Erlang build tool like rebar, this is handled for you.

NOTE *Rebar is an Erlang build tool used by the community in general. It understands the principles behind OTP applications and can act the way Emakefiles do. It can also fetch dependencies from git and mercurial repositories as needed.*

`{registered, AtomList}`
: This contains a list of all the names registered by the application. It lets OTP know if there will be name clashes when you try to bundle a bunch of applications together, but is entirely based on trusting the developers to give good data. We all know this isn't always the case, so blind faith shouldn't be used in this case, and some testing is always recommended.

`{env, [{Key, Val}]}`
: This is a list of key/values that can be used as a configuration for your application. They can be obtained at runtime by calling `application:get_env(Key)` or `application:get_env(AppName, Key)`. The former will try to find the value in the application file of whatever application you are in at the moment of the call. The latter allows you to specify a particular application. These values can be overwritten as required (either at boot time or by using `application:set_env(Application, Key, Value)`). Because

it's possible to overwrite these values, the `env` part of the application resource file is usually used for default values. This helps make the application usable with minimal user configuration.

All in all, this is a pretty useful place to store configuration data rather than having a bunch of configuration files to read in some arbitrary format, without really knowing where to store them and whatnot. People often tend to roll their own system to handle it anyway, since not everyone is a fan of using Erlang syntax in their configuration files.

`{maxT, Milliseconds}`

This is the maximum time that the application can run, after which it will be shut down. This is a rarely used item. `Milliseconds` defaults to `infinity`, so you often don't need to bother with this one at all.

`{applications, AtomList}`

This is a list of applications on which yours depends. The application system of Erlang will make sure they were loaded and/or started before allowing yours to do so. All applications depend at least on `kernel` and `stdlib`, but if your application were to depend on `ppool` being started, then you should add `ppool` to the list. It is important to add your dependencies, given OTP has mechanisms to know whether an application can be loaded or started based on this list. Not adding them is doing a disservice to yourself.

NOTE *Yes, the standard library and the VM's kernel are applications themselves, which means that Erlang is a language used to build OTP, but whose runtime environment depends on OTP to work. It's circular. This gives you some idea of why the language is officially named Erlang/OTP.*

`{mod, {CallbackMod, Args}}`

This defines a callback module for the application, using the application behavior (described shortly). This tells OTP that when starting your application, it should call `CallbackMod:start(normal, Args)`. It will also call `CallbackMod:stop(Args)` when stopping it. People will tend to name `CallbackMod` after their application.

And this covers most of what we need for now (and for most applications you'll ever write).

Converting the Pool

Now let's put this into practice! We'll turn the `ppool` set of processes from Chapter 18 into a basic OTP application. The first step for this is to redistribute everything under the correct directory structure:

```
ebin/
include/
```

```
priv/
src/
     - ppool.erl
     - ppool_serv.erl
     - ppool_sup.erl
     - ppool_supersup.erl
     - ppool_worker_sup.erl
test/
     - ppool_tests.erl
     - ppool_nagger.erl
```

You'll notice we moved the *ppool_nagger.erl* to the *test* directory. This is for a good reason: It is not much more than a demo case and will have nothing to do with our application, but is still necessary for the tests. We can actually try it later on once the app has been packaged to make sure everything still works, but for the moment, it's kind of useless.

We'll add an Emakefile (appropriately named `Emakefile`, placed in the app's base directory) to help us compile and run things later on.

```
{"src/*", [debug_info, {i,"include/"}, {outdir, "ebin/"}]}.
{"test/*", [debug_info, {i,"include/"}, {outdir, "ebin/"}]}.
```

This just tells the compiler to include `debug_info` for all files in *src/* and *test/*, go look in the *include/* directory (if it's ever needed), and then shove the files up its *ebin/* directory.

Speaking of which, let's add the app file in the *ebin/* directory.

```
{application, ppool,
 [{vsn, "1.0.0"},
  {modules, [ppool, ppool_serv, ppool_sup, ppool_supersup, ppool_worker_sup]},
  {registered, [ppool]},
  {mod, {ppool, []}}
]}.
```

This one contains only fields we find necessary; `env`, `maxT`, and `applications` are not used.

We now need to change how the callback module (`ppool`) works. How do we do that exactly?

First, let's see the application behavior.

NOTE *Even though all applications depend on the `kernel` and the `stdlib` applications, I haven't included them. `ppool` will still work because starting the Erlang VM starts these applications automatically. You might feel like adding them for the sake of explicitness, but there's no need for it right now.*

The Application Behavior

Remember that behaviors are always about splitting generic code away from specific code. They denote the idea that your specific code gives up its own execution flow and inserts itself as a bunch of callbacks to be used by the generic code. To put it simply, behaviors handle the boring parts while you connect the dots. In the case of applications, this generic part is quite complex and not nearly as simple as other behaviors.

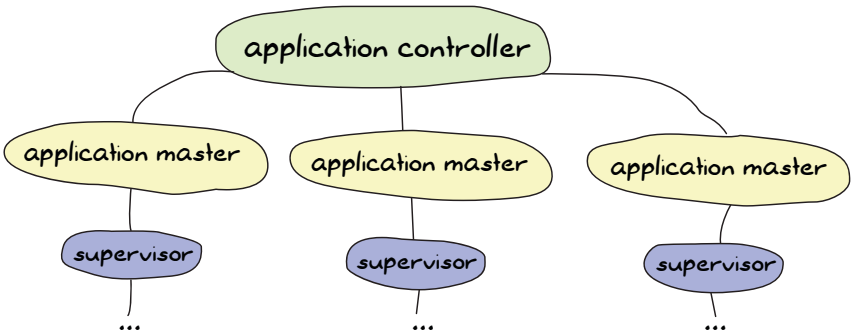
Whenever the VM first starts up, a process called the *application controller* is started (with the name `application_controller`). It starts all other applications and sits on top of most of them. In fact, you could say the application controller acts a bit like a supervisor for all applications. We'll cover the available supervision strategies in the next section.

> **THE EXCEPTION THAT CONFIRMS THE RULE**
>
> The application controller technically doesn't sit over all the applications. One exception is the `kernel` application, which itself starts a process named `user`. The `user` process acts as a group leader to the application controller, and the `kernel` application thus needs some special treatment. We don't need to care about this, but I felt like it should be included for the sake of precision.
>
> In Erlang, the I/O system depends on a concept called a *group leader*. The group leader represents standard input and output and is inherited by all processes. There is a hidden I/O protocol (*http://erlang.org/doc/apps/stdlib/io_protocol.html*) that the group leader and any process-calling I/O functions use to communicate. The group leader then takes the responsibility of forwarding these messages to whatever I/O channels there are, weaving some magic that doesn't concern us within the confines of this text.

When someone decides to start an application, the application controller (often referred to as *AC* in OTP parlance) starts an *application master*. The application master is two processes taking charge of each individual application. They set up the application and act like a middleman between your application's top supervisor and the application controller. OTP is a bureaucracy, and we have many layers of middle management! I won't get into the details of what happens in there, as most Erlang developers never actually need to care about this, and very little documentation exists (the code is the documentation). Just know that the application

master acts a bit like the app's nanny (well, a pretty insane nanny). It looks over its children and grandchildren, and when things go awry, it goes berserk and terminates its whole family tree. Brutally killing children is a common topic among Erlangers.

An Erlang VM with a bunch of applications might look a bit like this:



Up to now, we've been looking at the generic part of the behavior, but what about the specific stuff? After all, this is all we actually have to program. Well, the application callback module requires very few functions to be functional: `start/2` and `stop/1`.

The `start/2` function takes the form `YourMod:start(Type, Args)`. For now, the `Type` will always be `normal` (the other possibilities accepted have to do with distributed applications, which we'll cover in Chapter 27). `Args` is what is coming from your app file (in the `{mod, {YourMod, Args}}` tuple). The function initializes everything for your app and needs to return only the pid of the application's top-level supervisor in one of the two following forms: `{ok, Pid}` or `{ok, Pid, SomeState}`. If you don't return `SomeState`, it simply defaults to `[]`.

The `stop/1` function takes the state returned by `start/2` as an argument. It runs after the application is finished running and does only the necessary cleanup.

That's it—a huge generic part and a tiny specific one. Be thankful for that, because you wouldn't want to write the generic part yourself too often. (Just look at the source if you feel like it!) There are a few more functions that you can optionally use to have more control over the application, but we don't need them for now. This means we can move forward with our `ppool` application!

From Chaos to Application

We have the app file and a general idea of how applications work. Two simple callbacks are needed. Open *ppool.erl* and locate these lines:

```
-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().
```

Change this code to the following instead:

```
-behavior(application).
-export([start/2, stop/1, start_pool/3,
         run/2, sync_queue/2, async_queue/2, stop_pool/1]).

start(normal, _Args) ->
    ppool_supersup:start_link().

stop(_State) ->
    ok.
```

We can then make sure the tests are still valid. Open the old *ppool_tests.erl* file and replace the single call to `ppool:start_link/0` with `application:start(ppool)`, as follows:

```
find_unique_name() ->
    application:start(ppool),
    Name = list_to_atom(lists:flatten(io_lib:format("~p",[now()]))),
    ?assertEqual(undefined, whereis(Name)),
    Name.
```

You should also take the time to remove `stop/0` from `ppool_supersup` (and remove the export), because the OTP application tools will take care of that for us.

We can finally recompile the code and run all the tests to make sure everything still works (we'll cover how that `eunit` thing works in Chapter 24).

```
$ erl -make
Recompile: src/ppool_worker_sup
Recompile: src/ppool_supersup
... <snip> ...
$ erl -pa ebin/
... <snip> ...
1> make:all([load]).
Recompile: src/ppool_worker_sup
Recompile: src/ppool_supersup
Recompile: src/ppool_sup
Recompile: src/ppool_serv
Recompile: src/ppool
Recompile: test/ppool_tests
Recompile: test/ppool_nagger
up_to_date
2> eunit:test(ppool_tests).
  All 14 tests passed.
ok
```

The tests take a while to run due to `timer:sleep(X)` being used to synchronize everything in a few places, but you should find that they work, as shown here. Good news: Our app is healthy!

We can now study the wonders of OTP applications by using our new awesome callbacks:

```
3> application:start(ppool).
ok
4> ppool:start_pool(nag, 2, {ppool_nagger, start_link, []}).
{ok,<0.142.0>}
5> ppool:run(nag, [make_ref(), 500, 10, self()]).
{ok,<0.146.0>}
6> ppool:run(nag, [make_ref(), 500, 10, self()]).
{ok,<0.148.0>}
7> ppool:run(nag, [make_ref(), 500, 10, self()]).
noalloc
9> flush().
Shell got {<0.146.0>,#Ref<0.0.0.625>}
Shell got {<0.148.0>,#Ref<0.0.0.632>}
... <snip> ...
received down msg
received down msg
```

The magic command here is `application:start(ppool)`. This tells the application controller to launch our `ppool` application. It starts the `ppool_supersup` supervisor, and from that point on, everything can be used as normal. We can see all the applications currently running by calling `application:which_applications()`:

```
10> application:which_applications().
[{ppool,[],"1.0.0"},
 {stdlib,"ERTS  CXC 138 10","1.17.4"},
 {kernel,"ERTS  CXC 138 10","2.14.4"}]
```

What a surprise—`ppool` is running (the [] means we have put no description in the *.app* file). As mentioned earlier, we can see that all applications depend on `kernel` and `stdlib`, which are both running. We can close the pool as follows:

```
11> application:stop(ppool).

=INFO REPORT==== DD-MM-YYYY::23:14:50 ===
    application: ppool
    exited: stopped
    type: temporary
ok
```

And it is finished. You should notice that we now get a clean shutdown with a little informative report rather than the messy `** exception exit: killed` from Chapter 18.

NOTE *You'll sometimes see people do something like* `MyApp:start(...)` *instead of* `application:start(MyApp)`*. While this works for testing purposes, it loses a lot of the advantages of actually having an application. It's no longer part of the VM's supervision tree, cannot access its environment variables, will not check dependencies before being started, and so on. Try to stick to* `application:start/1` *if possible.*

But wait! What's that thing about our app being *temporary*? We write Erlang and OTP stuff because it's supposed to run forever, not just for a while! How dare the VM say this? The secret is that we can give different arguments to `application:start/1`. Depending on the arguments, the VM will react differently to termination of one of its applications. In some cases, the VM will be a loving beast ready to die for its children. In other cases, it's rather a cold, heartless, and pragmatic machine willing to tolerate many of its children dying for the survival of its species.

Application started with `application:start(AppName, temporary)`
: If it ends normally, nothing special happens, and the application has stopped.

  If it ends abnormally, the error is reported, and the application terminates without restarting.

Application started with `application:start(AppName, transient)`
: If it ends normally, nothing special happens, and the application has stopped.

  If it ends abnormally, the error is reported, all the other applications are stopped, and the VM shuts down.

Application started with `application:start(AppName, permanent)`
: If it ends normally, all other applications are terminated, and the VM shuts down.

  If it ends abnormally, the same thing happens: All applications are terminated, and the VM shuts down.

You can see something new in the supervision strategies when it comes to applications. No longer will the VM try to save you. At this point, something has gone very, very wrong to cause it to travel up the whole supervision tree of one of its vital applications—enough to crash it. When this does happen, the VM has lost all hope in your program. Given the definition of insanity is to keep doing the same thing while expecting different outcomes each time, the VM prefers to die sanely and just give up. Of course, the real reason has to do with something being broken that needs to be fixed, but you catch my drift. Note that all applications can be terminated by calling `application:stop(AppName)` without affecting others, as if a crash had occurred.

Library Applications

What happens when we want to wrap flat modules in an application but we have no process to start and thus no need for an application callback module?

After pulling our hair and crying in rage for a few minutes, the only other thing left to do is to remove the tuple `{mod, {Module, Args}}` from the application file. That's it. This is called a *library application*. The Erlang `stdlib` (standard library) application is an example of one of these.

If you have the source package of Erlang, you can go to *otp_src_<release>/lib/stdlib/src/stdlib.app.src* and see the following:

```
{application, stdlib,
 [{description, "ERTS  CXC 138 10"},
  {vsn, "%VSN%"},
  {modules, [array,
             ...
             gen_event,
             gen_fsm,
             gen_server,
             io,
             ...
             lists,
             ...
             zip]},
  {registered,[timer_server,rsh_starter,take_over_monitor,pool_master,
               dets]},
  {applications, [kernel]},
  {env, []}]}.
```

You can see it's a pretty standard application file, but without the callback module. Again, it's a library application.

How about we go deeper with applications and try building more complex ones?

20

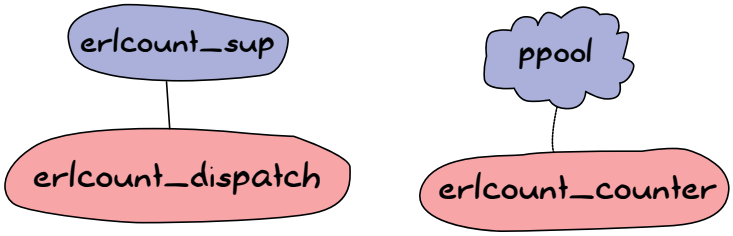
THE COUNT OF APPLICATIONS

Our `ppool` app has become a valid OTP application, and we now understand what this means. But wouldn't it be nice to build an application that actually uses our process pool to do something useful? To push our knowledge of applications a bit further, we will write a second application. This one will depend on `ppool` but will be able to benefit from some more automation than our "nagger" from Chapter 19.

This application, which we will name `erlcount`, will have a somewhat simple objective: recursively look into some directory, find all Erlang files (ending in *.erl*), and then run a regular expression over the result to count all instances of a given string within the modules. The results are then accumulated to give the final result, which will be output to the screen.

From OTP Application to Real Application

The `erlcount` application will be relatively simple, relying heavily on our process pool to do the work. It will have a structure as follows:



In this diagram, `ppool` represents the whole application but only means to show that `erlcount_counter` will be the worker for the process pool. It will open files, run the regular expression, and return the count. The process/module `erlcount_sup` will be our supervisor. `erlcount_dispatch` will be a single server in charge of browsing the directories, asking `ppool` to schedule workers, and compiling the results. We'll also add an `erlcount_lib` module, taking charge of hosting all the functions to read directories, compile data, and so on, leaving the other modules with the responsibility of coordinating these calls. Last will be an `erlcount` module, with the single purpose of being the application callback module.

The first step is to create the required directory structure. You can also add a few file stubs if you feel like doing so.

```
ebin/
 - erlcount.app
include/
priv/
src/
 - erlcount.erl
 - erlcount_counter.erl
 - erlcount_dispatch.erl
 - erlcount_lib.erl
 - erlcount_sup.erl
test/
Emakefile
```

This has nothing too different from the structure we used in Chapter 19, and you can even copy our old Emakefile.

We can probably start writing most parts of the application pretty quickly. The *.app* file, counter, library, and supervisor should be relatively simple. On the other hand, the dispatch module will need to accomplish some complex tasks if we want the application to be useful.

The Application File

Let's start with the app file, which looks like this:

```
{application, erlcount,
 [{vsn, "1.0.0"},
 {modules, [erlcount, erlcount_sup, erlcount_lib,
            erlcount_dispatch, erlcount_counter]},
 {applications, [ppool]},
 {registered, [erlcount]},
 {mod, {erlcount, []}},
 {env,
  [{directory, "."},
   {regex, ["if\\s.+->", "case\\s.+\\sof"]},
   {max_files, 10}]}
 ]}.
```

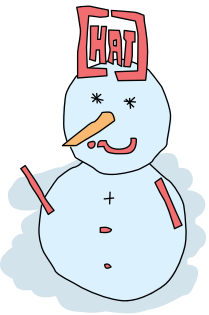
This app file is a bit more complex than the `ppool` one. We can still recognize some of the fields as being the same: this app will also be in version 1.0.0, and the modules are listed. The next part is something we didn't have for `ppool`: an application dependency. As explained earlier, the `applications` tuple gives a list of all the applications that should be started before `erlcount`. If you try to start it without that, you'll get an error message.

We then need to count the registered processes with `{registered, [erlcount]}`. Technically, none of our modules started as part of the `erlcount` app will need a name. Everything we do can be anonymous. However, because we know `ppool` registers the `ppool_serv` to the name we give it, and because we know we will use a process pool, we're going to call it `erlcount` and note it here. If all applications that use `ppool` do the same, we should be able to detect conflicts in the future. The `mod` tuple is similar to the one we used earlier. In it, we define the application behavior callback module.

The last new thing in here is the `env` tuple. This entire tuple gives us a key/value store for application-specific configuration variables. These variables will be accessible from all the processes running within the application, stored in memory for your convenience. They can basically be used as a substitute configuration file for your app.

In this case, we define three variables:

- `directory` tells the app where to look for *.erl* files. Assuming we run the app in an Erlang virtual machine started in the *learn-you-some-erlang* root (if you downloaded the package of files for this book), `"."` will refer to that directory.
- `max_files` tells us how many file descriptors should be opened at once. We don't want to open 10,000 files at once if we end up having that many, so this variable will match the maximum number of workers in `ppool`.
- `regex`, the most complex variable, will contain a list of all regular expressions we want to run over each of the files to count the results.



We won't get into the syntax of Perl Compatible Regular Expressions (PCRE) here (if you're interested, the `re` module's documentation contains full details), but we will look at what we're doing for our application. In this case, the first regular expression says, "Look for a string that contains `if` followed by any single whitespace character (`\s`, with a second backslash for escaping purposes) and finishes with `->`. Moreover, there can be anything in between the `if` and the `->` (`.+`)." The second regular expression says, "Look for a string that contains `case` followed by any single whitespace character (`\s`), and finishes with `of` preceded by a single whitespace character. Between the `case` and the `of`, there can be anything (`.+`)." To make things simple, we'll try to count how many times we use `case ... of` versus how many times we use `if ... end` in our libraries.

DON'T DRINK TOO MUCH KOOL-AID

Using regular expressions is not an optimal choice to analyze Erlang code. The problem is there are a lot of cases that will make your results inaccurate, including strings in the text and comments that match the patterns you're looking for but are technically not code.

To get more accurate results, you would need to look at the parsed and expanded version of your modules directly in Erlang. While more complex, this would make sure that you handle things like macros, exclude comments, and just generally do it the right way. You can look into `erl_syntax` or `xref` if this is something you wish to explore.

The Application Callback Module and Supervisor

With the app file out of the way, we can start writing the application callback module.

```
-module(erlcount).
-behavior(application).
-export([start/2, stop/1]).

start(normal, _Args) ->
    erlcount_sup:start_link().

stop(_State) ->
    ok.
```

It's not complex—basically it just starts the supervisor. And now let's set up the supervisor itself:

```
-module(erlcount_sup).
-behavior(supervisor).
-export([start_link/0, init/1]).

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

init([]) ->
    MaxRestart = 5,
    MaxTime = 100,
    {ok, {{one_for_one, MaxRestart, MaxTime},
          [{dispatch,
            {erlcount_dispatch, start_link, []},
            transient,
            60000,
            worker,
            [erlcount_dispatch]}]}}.
```

This is a standard supervisor, which will be in charge of only `erlcount_dispatch`, as it was shown in the previous little schema. The `MaxRestart`, `MaxTime`, and 60 seconds value for shutdown were chosen pretty randomly, but in real cases, you would want to study the needs you have. Because this is a demo application, it didn't seem that important. The author reserves the right to laziness.

The Dispatcher

The next process and module in the chain is the dispatcher. The dispatcher will have a few complex requirements to fulfill for it to be useful:

- When we search the directories to find files ending in *.erl*, we should go through the whole list of directories only once, even when we apply multiple regular expressions.
- We should be able to start scheduling files for result counting as soon as we find there is one that matches our criteria. We should not need to wait for a complete list to do so.
- We need to hold a counter per regular expression so we can compare the results in the end.
- It is possible that we will start getting results from the `erlcount_counter` workers before we're finished looking for *.erl* files.
- It is possible that many `erlcount_counters` will be running at once.
- It is likely we will keep getting results after we have finished looking up files in the directories (especially if we have many files or complex regular expressions).

The two big points we must consider right now are how we're going to go through a directory recursively while still being able to get results from there in order to schedule them, and then accept results back while that goes on, without getting confused.

Returning Results through CPS

At a first glance, the easiest way to gain the ability to return results while in the middle of recursion would be to use a process to do it. However, it's a bit annoying to change our previous structure just to be able to add another process to the supervision tree, and then to get the processes working together. There is, in fact, a simpler way to do things: Use a style of programming called *continuation-passing style (CPS)*.

The basic idea behind CPS is to take one function that's usually deeply recursive and break down every step. We return each step (which would usually be the accumulator), and then call a function that will allow us to keep going after that. In our case, our function will have two possible return values:

```
{continue, Name, NextFun}
done
```

Whenever we receive the first one, we can schedule `FileName` into `ppool` and then call `NextFun` to keep looking for more files. We can implement this function in `erlcount_lib`, like this:

```
-module(erlcount_lib).
-export([find_erl/1]).
-include_lib("kernel/include/file.hrl").

%% Finds all files ending in .erl.
find_erl(Directory) ->
    find_erl(Directory, queue:new()).
```

Ah, something new here! What a surprise; my heart is racing and my blood is pumping. The include file up there is something given to us by the `file` module. It contains a record (`#file_info{}`) with a bunch of fields explaining details about the file, including its type, size, permissions, and so on.

Our design includes a queue. Why is that? Well, it is entirely possible that a directory contains more than one file. So when we hit a directory and it contains something like 15 files, we want to handle the first one (and if it's a directory, open it, look inside, and so on), and then handle the 14 others

later. In order to do this, we will just store their names in memory until we have the time to process them. We use a queue for that, but a stack or any other data structure would still be fine, given we don't really care about the order in which we read files. The point is that this queue acts a bit like a to-do list for files in our algorithm.

Let's start by reading the first file passed from the first call:

```
%%% Private
%% Dispatches based on file type.
find_erl(Name, Queue) ->
    {ok, F = #file_info{}} = file:read_file_info(Name),
    case F#file_info.type of
        directory -> handle_directory(Name, Queue);
        regular -> handle_regular_file(Name, Queue);
        _Other -> dequeue_and_run(Queue)
    end.
```

This function tells us a few things. One is that we want to deal with only regular files and directories. In each case, we will write a function to handle these specific occurrences (`handle_directory/2` and `handle_regular_file/2`). For other files, we will dequeue anything we had prepared before with the help of `dequeue_and_run/2`. For now, we first start dealing with directories, as follows:

```
%% Opens directories and enqueues files in there.
handle_directory(Dir, Queue) ->
    case file:list_dir(Dir) of
        {ok, []} ->
            dequeue_and_run(Queue);
        {ok, Files} ->
            dequeue_and_run(enqueue_many(Dir, Files, Queue))
    end.
```

So if there are no files, we keep searching with `dequeue_and_run/1`. If there are many files, we enqueue them before searching further. The function `dequeue_and_run` will take the queue of filenames and get one element out of it. The filename it fetches from there will be used by calling `find_erl(Name, Queue)`, and we just keep going as if we were just getting started.

```
%% Pops an item from the queue and runs it.
dequeue_and_run(Queue) ->
    case queue:out(Queue) of
        {empty, _} -> done;
        {{value, File}, NewQueue} -> find_erl(File, NewQueue)
    end.
```

Note that if the queue is empty (`{empty, _}`), the function considers itself `done` (a keyword chosen for our CPS function); otherwise, we keep going over again.

The other function we need to consider is `enqueue_many/3`. This one is designed to enqueue all the files found in a given directory. It works as follows:

```
%% Adds a bunch of items to the queue.
enqueue_many(Path, Files, Queue) ->
    F = fun(File, Q) -> queue:in(filename:join(Path,File), Q) end,
    lists:foldl(F, Queue, Files).
```

Basically, we use the function `filename:join/2` to merge the directory's path to each filename (so that we get a complete path). We then add this new full path to a file to the queue. We use a fold to repeat the same procedure with all the files in a given directory. The new queue we get out of it is then used to run `find_erl/2` again, but this time with all the new files we found added to the to-do list.

We digressed a bit. Where were we? Oh yes, we were handling directories, and now we're finished with them. We then need to check for regular files and whether they end in *.erl*.

```
%% Checks if the file finishes in .erl.
handle_regular_file(Name, Queue) ->
    case filename:extension(Name) of
        ".erl" ->
            {continue, Name, fun() -> dequeue_and_run(Queue) end};
        _NonErl ->
            dequeue_and_run(Queue)
    end.
```

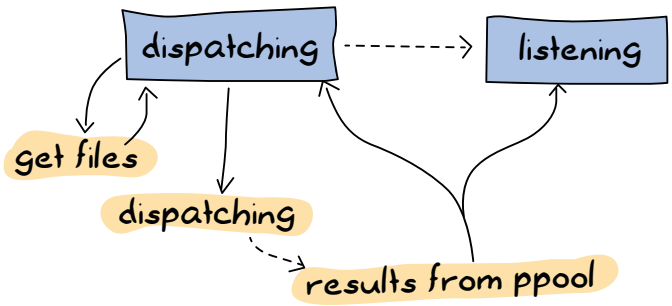
You can see that if the name matches (according to `filename:extension/1`), we return our continuation. The continuation gives the `Name` to the caller, and then wraps the operation `dequeue_and_run/1` with the queue of files left to visit into an anonymous function. That way, the user can call that anonymous function and keep going as if we were still in the recursive call, while still getting results in the meantime. In the case where the filename doesn't end in *.erl*, the user has no interest in us returning yet, and we keep going by dequeuing more files. That's it.

Hooray, the CPS thing is complete. We can now focus on the other issue.

Dispatching and Receiving

How are we going to design the dispatcher so that it can both dispatch and receive at once? My suggestion, which you will no doubt accept because I'm the one writing the text, is to use an FSM.

The FSM will have two states. The first one will be the "dispatching" state. It's the one used whenever we're waiting for our `find_erl` CPS function to hit the `done` entry. While we're in there, we will never think about being finished with the counting. That will happen in only the second and final state, "listening," but we will still receive notices from `ppool` all the time:



This will thus require us to have the following:

- A dispatching state with an asynchronous event for when we get new files to dispatch
- A dispatching state with an asynchronous event for when we are finished getting new files
- A listening state with an asynchronous event for when we are finished getting new files
- A global event to be sent by the `ppool` workers when they are finished running their regular expression

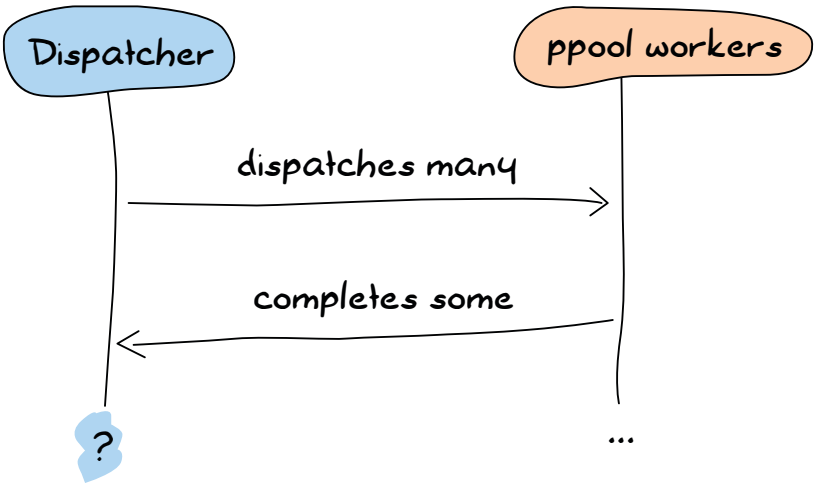
We'll slowly start building our `gen_fsm`:

```
-module(erlcount_dispatch).
-behavior(gen_fsm).
-export([start_link/0, complete/4]).
-export([init/1, dispatching/2, listening/2, handle_event/3,
         handle_sync_event/4, handle_info/3, terminate/3, code_change/4]).

-define(POOL, erlcount).
```

Our API will have two functions: one for the supervisor (`start_link/0`) and one for the `ppool` callers (`complete/4`, which we'll get to soon). The other functions are the standard `gen_fsm` callbacks, including our `listening/2` and `dispatching/2` asynchronous state handlers. We also defined a `?POOL` macro, used to give our `ppool` server the name `erlcount`.

What should the gen_fsm's data look like, though? Because we're going asynchronous and we will always call ppool:run_async/2 instead of anything else, we will have no real way of knowing if we're ever done scheduling files. Basically, we could have a timeline like this:



One way to solve the problem could be to use a timeout, but this is always annoying. Is the timeout too long or too short? Has something crashed? This much uncertainty is probably as much fun as a toothbrush made of lemon. Instead, we could use a concept where each worker is given some kind of identity, which we can track and associate with a reply, a bit like a secret password to enter the private club of "workers who succeeded." This concept will let us match one-on-one whatever message we get and let us know when we are absolutely finished. We now know what our state data might look like:

```
-record(data, {regex=[], refs=[]}).
```

The first list will be tuples of the form {RegularExpression, NumberOfOccurrences}, and the second will be a list of some kind of references to the messages. Anything will do, as long as it's unique. We can then add the two following API functions:

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

complete(Pid, Regex, Ref, Count) ->
    gen_fsm:send_all_state_event(Pid, {complete, Regex, Ref, Count}).
```

And here is our secret `complete/4` function. Unsurprisingly, the workers will need to send back only three pieces of data: what regular expression they were running, what their associated score was, and then the reference mentioned earlier. Awesome—now we can get into the real interesting stuff!

```
init([]) ->
    {ok, Re} = application:get_env(regex),
    {ok, Dir} = application:get_env(directory),
    {ok, MaxFiles} = application:get_env(max_files),
    ppool:start_pool(?POOL, MaxFiles, {erlcount_counter, start_link, []}),
    case lists:all(fun valid_regex/1, Re) of
        true ->
            self() ! {start, Dir},
            {ok, dispatching, #data{regex=[{R,0} || R <- Re]}};
        false ->
            {stop, invalid_regex}
    end.
```

The `init` function first loads all the information we need to run from the application file. Once that's done, we plan on starting the process pool with `erlcount_counter` as a callback module. The last step before actually starting to dispatch workers is to make sure all regular expressions are valid. The reason for this is simple: If we do not check it right now, then we will need to add an error-handling call somewhere else instead. This is likely going to be in the `erlcount_counter` worker. If it happens there, we now need to define what we do to handle workers crashing when regular expressions are invalid. It's just simpler to handle when starting the app and crash early. Here's the `valid_regex/1` function:

```
valid_regex(Re) ->
    try re:run("", Re) of
        _ -> true
    catch
        error:badarg -> false
    end.
```

We try to run the regular expression on only an empty string. This will take no time and let the `re` module try to run things. So the regular expressions are valid, and we start the app by sending ourselves `{start, Directory}` and with a state defined by `[{R,0} || R <- Re]`. This will basically change a list of the form `[a,b,c]` to the form `[{a,0},{b,0},{c,0}]`, the idea being to add a counter set to 0 to each of the regular expressions.

The first message we need to handle is `{start, Dir}` in `handle_info/2`. Remember that because Erlang's behaviors are pretty much all based on messages, we need to take the step of sending ourselves messages if we want to trigger a function call and do things our way. This is annoying but manageable.

```
handle_info({start, Dir}, State, Data) ->
    gen_fsm:send_event(self(), erlcount_lib:find_erl(Dir)),
    {next_state, State, Data}.
```

We send ourselves the result of `erlcount_lib:find_erl(Dir)`. It will be received in the `dispatching` callback, given that's the value of `State` as it was set by the `init` function of the FSM. This snippet solves our problem, but also illustrates the general pattern we'll have during the whole FSM. Because our `find_erl/1` function is written in CPS, we can just send ourselves an asynchronous event and deal with it in each of the correct callback states. It is likely that the first result of our continuation will be `{continue, File, Fun}`. We will also be in the "dispatching" state, because that's what we put as the initial state in the `init` function:

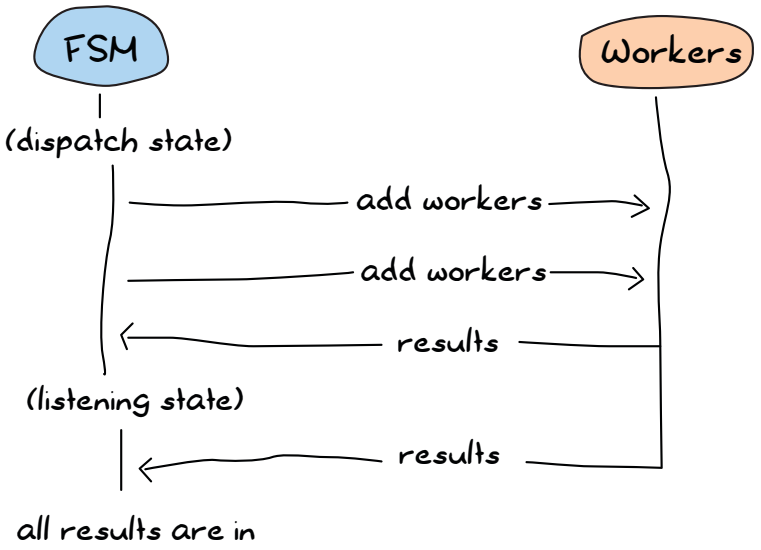
```
dispatching({continue, File, Continuation}, Data = #data{regex=Re, refs=Refs}) ->
    F = fun({Regex, _Count}, NewRefs) ->
        Ref = make_ref(),
        ppool:async_queue(?POOL, [self(), Ref, File, Regex]),
        [Ref|NewRefs]
    end,
    NewRefs = lists:foldl(F, Refs, Re),
    gen_fsm:send_event(self(), Continuation()),
    {next_state, dispatching, Data#data{refs = NewRefs}};
```

That's a bit ugly. For each of the regular expressions, we create a unique reference, schedule a `ppool` worker that knows this reference, and then store this reference (to know if a worker has finished). Doing this in a `foldl` makes it easier to accumulate all the new references. Once that dispatching is complete, we call the continuation again to get more results, and then wait for the next message with the new references as our state.

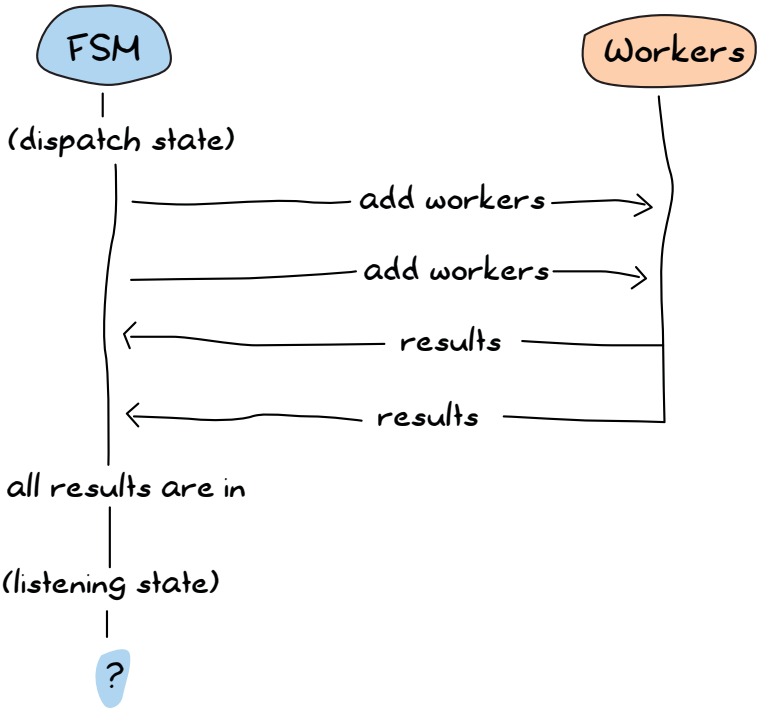
What's the next kind of message we can get? We have two choices here: Either none of the workers have given us our results back (even though they have not been implemented yet) or we get the `done` message because all files have been looked up. Let's go with the second type to finish implementing the `dispatching/2` function:

```
dispatching(done, Data) ->
    %% This is a special case. We cannot assume that all messages have NOT
    %% been received by the time we hit 'done'. As such, we directly move to
    %% listening/2 without waiting for an external event.
    listening(done, Data).
```

The comment is pretty explicit about what is going on. When we schedule jobs, we can receive results while in `dispatching/2` or while in `listening/2`. This can take the following form:



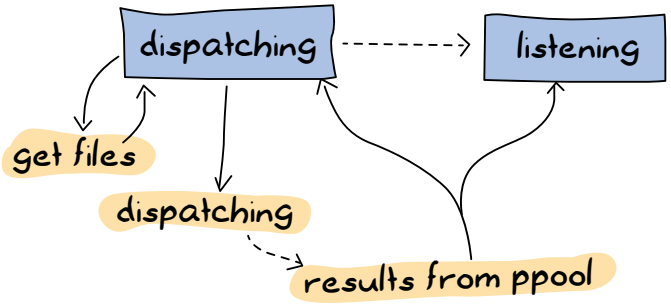
In this case, the listening state can just wait for results and declare everything is in. But remember that this is Erlang Land (Erland), and we work in parallel and asynchronously! This scenario is as probable:



Ouch. Our application would then be hanging forever, waiting for messages. This is why we need to manually call `listening/2`. We will force it to do some kind of result detection to make sure everything has been received, just in case we already have all the results. Here's what this looks like:

```
listening(done, #data{regex=Re, refs=[]}) -> % all received!
    [io:format("Regex ~s has ~p results~n", [R,C]) || {R, C} <- Re],
    {stop, normal, done};
listening(done, Data) -> % entries still missing
    {next_state, listening, Data}.
```

If no `refs` are left, then everything was received and we can output the results. Otherwise, we can keep listening to messages. Take another look at `complete/4` and our events diagram:



The result messages are global because they can be received in either dispatching or listening states. Here's the implementation:

```
handle_event({complete, Regex, Ref, Count}, State, Data = #data{regex=Re, refs=Refs}) ->
    {Regex, OldCount} = lists:keyfind(Regex, 1, Re),
    NewRe = lists:keyreplace(Regex, 1, Re, {Regex, OldCount+Count}),
    NewData = Data#data{regex=NewRe, refs=Refs--[Ref]},
    case State of
        dispatching ->
            {next_state, dispatching, NewData};
        listening ->
            listening(done, NewData)
    end.
```

The first thing this does is find the regular expression that just completed in the `Re` list, which also contains the count for all of them. We extract that value (`OldCount`) and update it with the new count (`OldCount+Count`) with the help of `lists:keyreplace/4`. We update our `Data` record with the new scores while removing the `Ref` of the worker, and then send ourselves to the next state.

In normal FSMs, we would just have done `{next_state, State, NewData}`, but here, because of the problem with knowing when we're finished, we must manually call `listening/2` again. This may be a pain, but alas, it's a necessary step.

And that's it for the dispatcher. We just add in the rest of the filler behavior functions, as follows:

```
handle_sync_event(Event, _From, State, Data) ->
    io:format("Unexpected event: ~p~n", [Event]),
    {next_state, State, Data}.

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

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

Next, we'll move on to the counter. You might want to take a little break before then. Hard-core readers can go bench-press their own weight a few times to relax themselves and then come back for more.

The Counter

The counter is simpler than the dispatcher. While we still need a behavior to do things (in this case, a `gen_server`), it will be quite minimalist. We need it to do only three things:

- Open a file
- Run a regular expression on it and count the instances
- Return the result

For the first task, we have plenty of functions in `file` to help us. For the third task, we defined `erlcount_dispatch:complete/4` to handle it. For the second, we can use the `re` module with `run/2-3`, but it doesn't quite do what we need, as you can see here:

```
1> re:run(<<"brutally kill your children (in Erlang)">>, "a").
{match,[{4,1}]}
2> re:run(<<"brutally kill your children (in Erlang)">>, "a",
2>        [global]).
{match,[[{4,1}],[{35,1}]]}
3> re:run(<<"brutally kill your children (in Erlang)">>, "a",
3>        [global, {capture, all, list}]).
{match,[["a"],["a"]]}
4> re:run(<<"brutally kill your children (in Erlang)">>, "child",
4>        [global, {capture, all, list}]).
{match,[["child"]]}
```

While the function does take the arguments we need (`re:run(String, Pattern, Options)`), it doesn't give us the correct count. Let's add the following function to `erlcount_lib` so we can start writing the counter:

```
regex_count(Re, Str) ->
    case re:run(Str, Re, [global]) of
        nomatch -> 0;
        {match, List} -> length(List)
    end.
```

This one basically just counts the results and returns that value. Don't forget to add it to the export attribute.

Now let's continue by defining the worker, as follows:

```
-module(erlcount_counter).
-behavior(gen_server).
-export([start_link/4]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-record(state, {dispatcher, ref, file, re}).

start_link(DispatcherPid, Ref, FileName, Regex) ->
    gen_server:start_link(?MODULE, [DispatcherPid, Ref, FileName, Regex], []).

init([DispatcherPid, Ref, FileName, Regex]) ->
    self() ! start,
    {ok, #state{dispatcher=DispatcherPid,
                ref = Ref,
                file = FileName,
                re = Regex}}.

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

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

handle_info(start, S = #state{re=Re, ref=Ref}) ->
    {ok, Bin} = file:read_file(S#state.file),
    Count = erlcount_lib:regex_count(Re, Bin),
    erlcount_dispatch:complete(S#state.dispatcher, Re, Ref, Count),
    {stop, normal, S}.

terminate(_Reason, _State) ->
    ok.

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

The two interesting sections here are the `init/1` callback, where we order ourselves to start, and then a single `handle_info/2` clause, where we open the file (`file:read_file(Name)`) and get back a binary, which we pass to our new `regex_count/2` function, and then send it back with `complete/4`. We then stop the worker. The rest is just standard OTP callback stuff.

We can now compile and run the whole thing!

```
$ erl -make
Recompile: src/erlcount_sup
Recompile: src/erlcount_lib
Recompile: src/erlcount_dispatch
Recompile: src/erlcount_counter
Recompile: src/erlcount
Recompile: test/erlcount_tests
```

Hell, yes. Pop the champagne because we have no whine!

Run App Run

There are many ways to get our app running. Make sure you're in a directory where you somehow have these two directories next to each other:

```
erlcount-1.0
ppool-1.0
```

Now start Erlang the following way:

```
$ erl -env ERL_LIBS "."
```

`ERL_LIBS` is a special variable defined in your environment that lets you specify where Erlang can find OTP applications. The VM is then able to automatically look there to find the *ebin/* directories for you. The `erl` executable can also take an argument of the form `-env NameOfVar Value` to override this setting quickly, so that's what we used here. The `ERL_LIBS` variable is pretty useful, especially when installing libraries, so try to remember it!

With the VM we started, we can test that the modules are all there:

```
1> application:load(ppool).
ok
```

This function will try to load all the application modules in memory if they can be found. If you don't call it, the loading will be done automatically when starting the application, but this provides an easy way to test our paths. We can start the apps:

```
2> application:start(ppool), application:start(erlcount).
ok
```

```
Regex if\s.+-> has 20 results
Regex case\s.+\sof has 26 results
```

Your results may vary depending on what you have in your directories. Note that depending how many files you have, this can take a while.

What if we want different variables to be set for our applications, though? Do we need to change the application file all the time? No, we don't! Erlang also supports that. So let's say we wanted to see how many times the developers of Erlang are angry in their source files?

The `erl` executable supports a special set of arguments of the form `-AppName Key1 Val1 Key2 Val2 ... KeyN ValN`. In this case, we could then run the following regular expression over the Erlang source code from the R15B01 distribution with two regular expressions, as follows:

```
$ erl -env ERL_LIBS "." -erlcount directory '"/home/ferd/otp_src_R15B01/lib/"' regex '["shit","damn"]'
... <snip> ...
1> application:start(ppool), application:start(erlcount).
ok
Regex shit has 3 results
Regex damn has 1 results
2> q().
ok
```

Note that in this case, all expressions given as arguments are wrapped in single quotation marks (`'`). That's because I want them to be taken literally by my Unix shell. Different shells might have different rules.

We could also try our search with more general expressions, allowing values to start with capital letters and more file descriptors:

```
$ erl -env ERL_LIBS "." -erlcount directory '"/home/ferd/otp_src_R15B01/lib/"'
  regex '["[Ss]hit","[Dd]amn"]' max_files 50
... <snip> ...
1> application:start(ppool), application:start(erlcount).
ok
Regex [Ss]hit has 13 results
Regex [Dd]amn has 6 results
2> q().
ok
```

Oh, OTP programmers, what makes you so angry ("working with Erlang" not being an acceptable answer)?

This one might take even longer to run due to the more complex checks required over the hundreds of files there. This all works pretty well, but there are a few annoying things here. Why are we always manually starting both applications? Isn't there a better way?

Included Applications

Included applications are one way to get things working. The basic idea of an included application is that you define an application (in this case `ppool`) as an application that is part of another one (`erlcount` here). To do this, a bunch of changes need to be made to both applications.

The gist of this approach is that you modify your application file a bit, and then you need to add something called *start phases* to them, and respect a given protocol described at length in the Erlang documentation.

It is more and more recommended *not* to use included applications for a simple reason: They seriously limit code reuse. We've spent a lot of time working on `ppool`'s architecture to make it so anyone can use it, get their own pool, and be free to do whatever they want with it. If we were to push it into an included application, then it can no longer be included in any other application on this VM. Also, if `erlcount` dies, then `ppool` will be taken down with it, ruining the work of any third-party application that wanted to use `ppool`.

For these reasons, included applications are usually excluded from many Erlang programmers' toolbox, although some still love them. As we will discuss in the following chapter, releases can basically help us do the same (and much more) in a more generic manner.

But before we get to that, we have a one more application topic left to discuss.

Complex Terminations

In some cases, we need more steps to be taken before terminating our application. The `stop/1` function from the application callback module might not be enough, especially since it is called *after* the application has already terminated. What do we do if we need to clean up things before the application is actually gone?

The trick is simple: Just add a function `prep_stop(State)` to your application callback module. `State` will be the state returned by your `start/2` function, and whatever `prep_stop/1` returns will be passed to `stop/1`. The function `prep_stop/1` thus technically inserts itself between `start/2` and `stop/1`, and is executed while your application is still alive, but just before it shuts down. For your own code, you will know when you need to use this kind of callback. We don't require it for our application right now.

Now that we have basic applications working, we're going to start thinking about packaging our applications into releases.

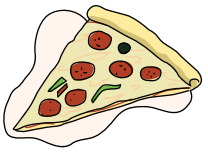
21

RELEASE IS THE WORD

How far have we really gotten? All this work, all these concepts, and we haven't shipped a single Erlang executable yet! You might agree with me that getting an Erlang system up and running requires a lot of effort, especially compared to many languages where you call the compiler and off you go.

Of course this is entirely right. We can compile files, run applications, check for some dependencies, handle crashes, and so on, but it's not very useful without a functioning Erlang system you can easily deploy or ship with the code we wrote. What use is having a great pizza when it can only be delivered cold? (People who enjoy cold pizza might feel excluded here. I am sorry.)

The OTP team didn't leave us on our own when it comes to making sure real systems come to life. OTP releases are part of a system made to help package applications with the minimal resources and dependencies. In this chapter, we'll look at the two major ways to handle releases, Systools and Reltool.



Fixing the Leaky Pipes

For our first release, we will reuse our `ppool` and `erlcount` applications from the previous chapters. However, before we do so, we'll need to change a few things here and there.

If you're following along with the book and writing your own code, you might want to copy both of our apps into a new directory called *release/*, which I will assume you have done for the rest of the chapter.

Terminating the VM

The first thing that's really bothersome about `erlcount` is that once it's finished running, the VM stays up, doing nothing. We might want most applications to stay running forever, but that's not the case here. Keeping it running makes sense during development, because we might want to play with a few things in the shell and need to manually start applications, but this should no longer be necessary.

For this reason, we'll add a command that will shut down the Erlang VM in an orderly manner. The best place to do this is within `erlcount_dispatch.erl`'s own `terminate` function, given it's called after we obtain the results. The perfect function to tear everything down is `init:stop/0`. This function is quite complex, but will take care of terminating our applications in order. It will get rid of file descriptors, sockets, and so on for us. The new `stop` function should now look like this:

```
terminate(_Reason, _State, _Data) ->
    init:stop().
```

And that's it for the code itself. However, we still have a bit more work to do.

Updating the Application Files

When we defined our app files in the preceding chapters, we did so while using the absolute minimal amount of information necessary to get them running. A few more fields are required for releases so that Erlang isn't completely mad at us.

First, the Erlang tools to build releases require us to be a little more precise in our application descriptions. You see, although tools for releases don't understand documentation, they still have this intuitive fear of code where the developers were too impolite to at least leave an idea of what the application does. For this reason, we'll need to add a `description` tuple to both our *ppool.app* and *erlcount.app* files.

For `ppool`, add the following:

```
{description, "Run and enqueue different concurrent tasks"}
```

For erlcount, add this:

```
{description, "Run regular expressions on Erlang source files"}
```

Now we'll be able to get a better idea of what's going on when we inspect our different systems.

The most attentive readers will also remember I've mentioned at some point that *all* applications depend on stdlib and kernel. However, our two app files do not mention these. Let's add both applications to each of our app files. Add the following tuple to the ppool app file:

```
{applications, [stdlib, kernel]}
```

Also add the two applications to the existing erlcount app file, giving us:

```
{applications, [stdlib, kernel, ppool]}.
```

DON'T DRINK TOO MUCH KOOL-AID

While adding the stdlib and kernel applications to the list in the app file might have virtually no impact when we start releases manually (and even when we generate them with systools, which we'll explore soon), it is absolutely vital to do so.

People who generate releases with Reltool (the other tool we'll cover in this chapter) will definitely need these applications in order for their release to run well, and even to be able to shut down the VM in a respectable manner. I'm not kidding—it's *really* necessary. I forgot to do it when writing this chapter, and lost a night of work trying to find what was wrong, only to discover that it was just me not doing things right in the first place.

It could be argued that, ideally, the release systems of Erlang could implicitly add these applications, given pretty much all of them (except very special cases) will depend on them. Alas, they don't.

Compiling the Applications

We have a termination in place and have updated the app files. The last step before we start working with releases is to *compile all your applications*.

Successively run your Emakefile files (with erl -make) in each directory containing one. Otherwise, Erlang's tools won't do it for you, and you'll end up with a release without code to run. Ouch.

Releases with systools

The systools application is the simplest one to use to build Erlang releases. It's the Easy-Bake Oven of Erlang releases. To get your delicious releases out of the systools oven, you first need a basic recipe and list of ingredients. Here's a list of the ingredients of a successful minimal Erlang release for our erlcount application (erlcount 1.0.0):

- An Erlang Run-Time System (ERTS) of your choice
- A standard library
- A kernel library
- The ppool application, which should not fail
- The erlcount application

Did I mention that I'm a terrible cook? I'm not sure I can even make pancakes, but at least I know how to build an OTP release. The ingredient list for an OTP release with systools looks like this file, named *erlcount-1.0.rel* and placed at the top level of the *release/* directory:

```
{release,
 {"erlcount", "1.0.0"},
 {erts, "5.9.1"},
 [{kernel, "2.15.1"},
  {stdlib, "1.18.1"},
  {ppool, "1.0.0", permanent},
  {erlcount, "1.0.0", transient}]}.
```

This has the same content as the textual list of ingredients for the recipe, although we can specify how we want the applications to be started (temporary, transient, or permanent). We can also specify versions so we can mix and match different libraries from different Erlang versions depending on our needs. To get all the version numbers in there, we can just make the following sequence of calls:

```
$ erl
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:4:4] [async-threads:0]
[hipe] [kernel-poll:false]

Eshell V5.9.1  (abort with ^G)
1> application:which_applications().
[{stdlib,"ERTS  CXC 138 10","1.18.1"},
 {kernel,"ERTS  CXC 138 10","2.15.1"}]
```

For this example, I was running R15B01. You can see the ERTS version in there right after the release number (the version is 5.9.1). Then by calling application:which_applications() on a running system, I can see the two versions I need from kernel (2.15.1) and stdlib (1.18.1). The numbers will vary from Erlang version to version. However, being explicit about the versions you need is helpful because it means that if you have many different

Erlang installations on a system, you can still use an older version of `stdlib` that won't badly influence whatever you're doing.

You'll also note that I chose to name the *release* as `erlcount` and make it version 1.0.0. This is unrelated to the `ppool` and `erlcount` applications, which are both also running version 1.0.0, as specified in their app file.

So now we have all our applications compiled, our list of ingredients, and the wonderful concept of a metaphorical Easy-Bake Oven. What we need is the actual recipe.

A recipe will tell you a few things: in what order to add ingredients, how to mix them, how to cook them, and so on. The part about the order used to add them is covered by our list of dependencies in each app file. The `systools` application will be clever enough to look at the app files and figure out what needs to run before what. But we do need to handle relaying the other instructions.

Creating a Boot File

Erlang's VM can start itself with a basic configuration taken from something called a *boot file*. In fact, when you start your own `erl` executable from the shell, it implicitly calls the ERTS with a default boot file. That boot file will give basic instructions such as "load the standard library," "load the kernel application," "run a given function," and so on. That boot file is a binary file created from a *boot script* (*http://www.erlang.org/doc/man/script.html*), which contains tuples that will represent these instructions. We'll write such a boot script now.

First we start with the following:

```
{script, {Name, Vsn},
 [
  {progress, loading},
  {preLoaded, [Mod1, Mod2, ...]},
  {path, [Dir1,"$ROOT/Dir",...]}.
  {primLoad, [Mod1, Mod2, ...]},
  ...
```

I'm just kidding. No one really takes the time to do that, and we won't either. The boot script is something easy to generate from the *.rel* file. Just start an Erlang VM from the *release/* directory and call the following line:

```
$ erl -env ERL_LIBS .
... <snip> ...
1> systools:make_script("erlcount-1.0", [local]).
ok
```

Now if you look in your directory, you will have a bunch of new files, including *erlcount-1.0.script* and *erlcount-1.0.boot*. Here, the `local` option means that we want the release to be able to run from anywhere, and not just the current installation. The `systools` application has many more options (see *http://www.erlang.org/doc/man/systools.html*), but because `systools` isn't as powerful as Reltool (which we'll discuss in the next section), we won't look into them with too much depth.

At this point, we have the boot script, but not enough to distribute our code yet.

Packaging the Release

Go back to your Erlang shell and run the following command:

```
2> systools:make_tar("erlcount-1.0", [{erts, "/usr/local/lib/erlang/"}]).
ok
```

Or, on Windows 7, run this:

```
2> systools:make_tar("erlcount-1.0",
2>                   [{erts, "C:/Program Files (x86)/erl5.9.1"}]).
ok
```

Here, `systools` will look for your release files and the ERTS (because of the `erts` option). If you omit the `erts` option, the release won't be self-executable and will depend on Erlang already being installed on a system.

Running this function call creates an archive file named *erlcount-1.0.tar.gz*. Unarchive the files inside the archive, and you should see a directory like this:

```
erts-5.9.1/
lib/
releases/
```

The *erts-5.9.1/* directory will contain the ERTS. The *lib/* directory holds all the applications we need, and the *releases/* directory has the boot files and other files related to releases.

Move into the directory where you extracted these files. From there, we can build a command-line call for `erl`. First, we specify where to find the `erl` executable and the boot file (without the *.boot* extension). In Linux, this gives us the following:

```
$ ./erts-5.9.1/bin/erl -boot releases/1.0.0/start
```

The command is the same on Windows 7, using Windows PowerShell.

You can optionally use absolute paths if you want the command to work from anywhere on your computer. Don't run it right now, though. It's going to be useless because there is no source file to analyze in the current directory. If you use absolute paths, you can go to the directory you want to analyze and call the file from there.

DON'T DRINK TOO MUCH KOOL-AID

There is no guarantee that a release will work on any system ever. If you're using pure Erlang code without native compiling with HiPE (a native compiler for Erlang code, which gives somewhat faster code, especially for CPU-bound applications), then that code will be portable. The issue is that the ERTS you ship with it might itself not work. You will need to either create many binary packages for many different platforms for large-scale distribution or just ship the BEAM files without the associated ERTS and ask people to run them with an Erlang system they have on their own computer.

The `erlcount` application's implementation would use the current directory as its default point to start searching. It is, however, possible to configure which directory to scan by overriding the application's `env` variables. Let's add `-erlcount directory "'<path to the directory>'"` to the command. Then because we want this to not look like Erlang, let's add the `-noshell` argument. This gives me something like this on my own computer:

```
$ ./erts-5.9.1/bin/erl -boot releases/1.0.0/start -erlcount directory
'"/home/ferd/code/otp_src_R14B03/"' -noshell
Regex if\s.+-> has 3846 results
Regex case\s.+\sof has 55894 results
```

I was running `erlcount` on old Erlang and OTP releases. You can try it on more recent ones. Using absolute file paths, I get something like this long command:

```
$ /home/ferd/code/learn-you-some-erlang/release/rel/erts-5.9.1/bin/erl -boot
/home/ferd/code/learn-you-some-erlang/release/rel/releases/1.0.0/start -noshell
```

Wherever I run it from, that's the directory that's going to be scanned. Wrap this up in a shell script or a batch file, and you should be good to go.

Releases with Reltool

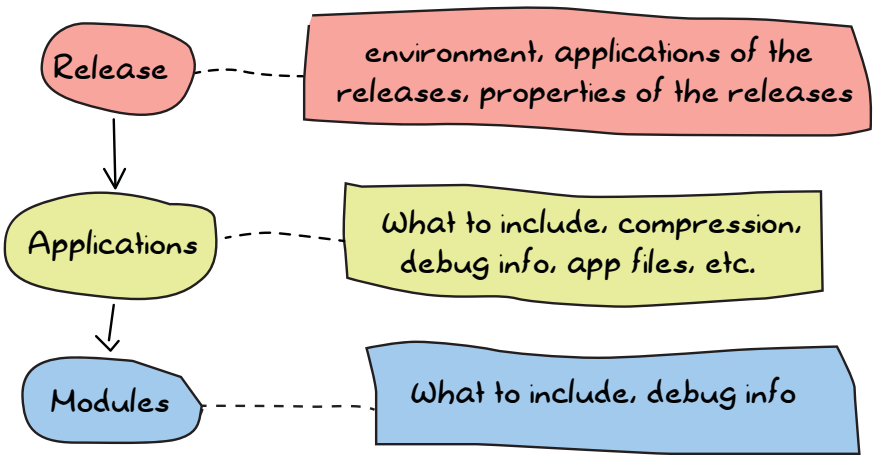
There are a bunch of aspects of `systools` that are annoying. We have very little control over how things are done. Manually specifying the path to the boot file and whatnot is kind of painful. Moreover, the files are a bit large. The whole release takes more than 20MB on disk, and it would be a lot worse if we were to package more applications. It is possible to do better with Reltool, as we get a lot more power, although the trade-off is increased complexity.

Reltool works from a configuration file that looks like this:

```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
    {rel, "erlcount", "1.0.0",
     [kernel,
      stdlib,
      {ppool, permanent},
      {erlcount, transient}
      % {LibraryApp, load} is also an option for stuff that never starts.
     ]},
    {boot_rel, "erlcount"},
    {relocatable, true},
    {profile, standalone},
    {app, ppool, [{vsn, "1.0.0"},
                  {app_file, all},
                  {debug_info, keep}]},
    {app, erlcount, [{vsn, "1.0.0"},
                     {incl_cond, include},
                     {app_file, strip},
                     {debug_info, strip}]}
]}.
```

Behold the user-friendliness of Erlang! To be quite honest, there's no easy way to introduce Reltool. You need a bunch of these options at once or nothing will work. It might sound confusing, but there's logic behind it.

First of all, Reltool will take different levels of information. The first level will contain release-wide information. The second level will be application-specific, before allowing fine-grained control at a module-specific level.



For each of these levels, different options will be available. Rather than taking the encyclopedic approach with all the options possible, we'll visit a few essential options and then a few possible configurations.

The first option is one that helps us get rid of the somewhat annoying need to be sitting in a given directory or to set the correct `-env` arguments

to the VM. The option is `lib_dirs`, and it takes a list of directories where applications reside. So instead of adding `-env ERL_LIBS` *`list:of:directories`*, you put in `{lib_dirs, [`*`ListOfDirectories`*`]}` and get the same result.

Another vital option for the Reltool configuration files is `rel`. This tuple is very similar to the *.rel* file we wrote for `systools`. In the demo file, we have this:

```
{rel, "erlcount", "1.0.0",
 [kernel,
  stdlib,
  {ppool, permanent},
  {erlcount, transient}
]},
```

This is what we need to define which apps must be started correctly. After that tuple, we want to add a tuple of this form:

```
{boot_rel, "erlcount"}
```

This will tell Reltool that whenever someone runs the `erl` binary included in the release, we want the apps from the `erlcount` release to be started. With just these three options—`lib_dirs`, `rel`, and `boot_rel`—we can get a valid release.

To do so, we'll put these tuples into a format Reltool can parse:

```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
    {rel, "erlcount", "1.0.0",
     [kernel,
      stdlib,
      {ppool, permanent},
      {erlcount, transient}
     ]},
    {boot_rel, "erlcount"}
]}.
```

We just wrap them into a `{sys, [Options]}` tuple. I saved this in a file named *erlcount-1.0.config* in the *release/* directory. You can put it anywhere you want (except */dev/null*, even though it has exceptional write speeds!).

Then we'll need to open an Erlang shell:

```
1> {ok, Conf} = file:consult("erlcount-1.0.config").
{ok,[{sys,[{lib_dirs,["/home/ferd/code/learn-you-some-erlang/release/"]},
     {rel,"erlcount","1.0.0",
          [kernel,stdlib,{ppool,permanent},{erlcount,transient}]},
     {boot_rel,"erlcount"}]}]}
2> {ok, Spec} = reltool:get_target_spec(Conf).
{ok,[{create_dir,"releases",
... <snip> ...
3> reltool:eval_target_spec(Spec, code:root_dir(), "rel").
ok
```

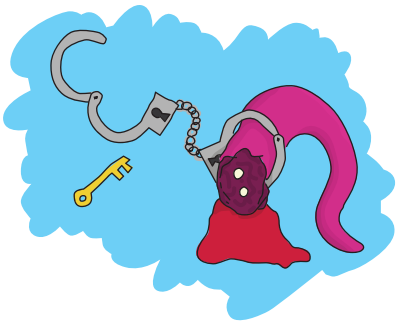
The first step here is to read the configuration and bind it to the `Conf` variable. Then we send that into `reltool:get_target_spec(Conf)`. The function will take a while to run and return way too much information for us to proceed. We don't care, so we just save the result in `Spec`.

The third command takes the specification and tells Reltool, "I want you to take my release specification, using whatever path where my Erlang installations are, and shove it into the *rel* directory." That's it. Look in the *rel* directory, and you should find a bunch of subdirectories there.

For now, we don't care and can just call this:

```
$ ./bin/erl -noshell
Regex if\s.+-> has 0 results
Regex case\s.+\sof has 0 results
```

Ah, this is a bit simpler to run. You can put these files pretty much anywhere, as long as they keep the same file tree, and run them from wherever you want.



Have you noticed something different? I hope you have. We didn't need to specify any version numbers. Reltool is a bit more clever than `systools` there. If you do not specify a version, it will automatically look for the newest one possible in the paths you have (either in the directory returned by `code:root_dir()` or what you put in the `lib_dirs` tuple).

But what if I'm not hip and cool and trendy and all about the latest apps, but rather a retro lover? I'm still wearing my disco pants, and I want to use older ERTS versions and older library versions (I've never stayed more alive than I was in 1977!).

Thankfully, Reltool can handle releases that need to work with older versions of Erlang. Respecting your elders is an important concept for Erlang tools.

If you have older versions of Erlang installed, you can add an `{erts, [{vsn, Version}]}` entry to the configuration file:

```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
    {erts, [{vsn, "5.8.3"}]},
    {rel, "erlcount", "1.0.0",
     [kernel,
      stdlib,
      {ppool, permanent},
      {erlcount, transient}
     ]},
    {boot_rel, "erlcount"}
]}.
```

Now you want to clear out the *rel/* directory to get rid of the newer release. Then you run the rather ugly sequence of calls again:

```
4> f(),
4> {ok, Conf} = file:consult("erlcount-1.0.config"),
4> {ok, Spec} = reltool:get_target_spec(Conf),
4> reltool:eval_target_spec(Spec, code:root_dir(), "rel").
ok
```

A quick reminder here: `f()` is used to unbind the variables in the shell.

Now if I go to the *rel* directory and call `$ ./bin/erl`, I get the following output:

```
Erlang R14B02 (erts-5.8.3) [source] ...

Eshell V5.8.3  (abort with ^G)
1> Regex if\s.+-> has 0 results
Regex case\s.+\sof has 0 results
```

Awesome! This runs on version 5.8.3, even though I have newer ones available (Ah, ha, ha, ha, stayin' alive . . .). For the preceding snippet to work for you, you must have Erlang R14B02 installed beforehand.

NOTE *If you look at the* rel/ *directory, you'll see things are kind of similar to what they were with* `systools`*. One difference will be in the* lib/ *directory, which will now contain a bunch of directories and* .ez *files. The directories in* lib/ *will contain the* include/ *files required when you want to do development using the libraries from the release, and the* priv/ *directories when there are files that need to be kept there, such as C drivers or specific files required by running applications. The* .ez *files, on the other hand, are just zipped BEAM files. The Erlang VM will unpack them for you come runtime; this setup is just to make things lighter.*

But wait, what about the other modules?

Ah, now we move away from the release-wide settings and enter the realm of settings that have to do with applications. There are still a lot of release-wide options to see, but we're on such a roll that we can't be asked to stop right now. We'll revisit them in the next section.

For applications, we can specify versions by adding more tuples:

```
{app, AppName, [{vsn, Version}]}
```

Put in one per app that needs it.

Reltool Options

Now we have many more options for everything. We can specify if we want the release to include debug information or strip it away, whether to try to make more compact app files or keep the existing ones, which stuff to include or exclude, how strict to be when it comes to including applications

and modules on which your own applications might depend, and so on. Moreover, these options can usually be defined both release-wide and application-wide, so you can specify defaults and then values to override.

Here, we'll take a quick look at the Reltool options. If you find these complex, just skip to the next section, where you'll find a few Reltool cookbook recipes to follow.

Release-Only Options

The following are Reltool release-only options.

`{lib_dirs, [ListOfDirs]}`
: This lets you specify which directories to search for libraries.

`{excl_lib, otp_root}`
: Added in R15B02, this option lets you specify OTP applications as part of your release, without including whatever comes from the standard Erlang/OTP path in the final release. This lets you create releases that are essentially libraries bootable from an existing virtual machine installed in a given system. When using this option, you must start the virtual machine as `$ erl -boot_var RELTOOL_EXT_LIB path/to/release directory/lib -boot path/to/boot/file`. This will allow the release to use the current Erlang/OTP install, but with your own libraries for your custom release.

`{app, AppName, [AppOptions]}`
: This lets you specify application-wide options, which are usually more specific than the release-wide options.

`{boot_rel, ReleaseName}`
: This lets you specify the default release to boot with the `erl` executable. This means you won't need to specify the boot file when calling `erl`.

`{rel, Name, Vsn, [Apps]}`
: This lets you specify the applications to be included in the release.

`{relocatable, true | false}`
: It is possible to run the release from everywhere or from a hard-coded path in your system. By default, this option is set to `true`, and I tend to leave it that way unless there is a good reason to do otherwise. You'll know when you need it.

`{profile, development | embedded | standalone}`
: This option serves as a way to specify default `*_filters` (described in the next list) based on your type of release. By default, `development` is used. That one will include more files from each app and ERTS blindly. The `standalone` profile will be more restrictive, and the `embedded` profile even more so, dropping more default ERTS applications and binaries.

Release-wide and Application-wide Options

The following are Reltool release-wide and application-wide options, Note that for all of these, setting the option on the level of an application will simply override the value you gave at a system level.

`{incl_sys_filters, [RegularExpressions]}` and `{excl_sys_filters, [RegularExpressions]}`
: These check whether a file matches the include filters without matching the exclude filters before including it. You might drop or include specific files this way.

`{incl_app_filters, [RegularExpressions]}` and `{excl_app_filters, [RegularExpressions]}`
: These are similar to `incl_sys_filters` and `excl_sys_filters`, but for application-specific files.

`{incl_archive_filters, [RegularExpressions]}` and `{excl_archive_filters, [RegularExpressions]}`
: These specify which top-level directories must be included or excluded in *.ez* archive files (more on this in the next section). Files not included in the archive may still be included in the release, but just not compressed.

`{incl_cond, include | exclude | derived}`
: This decides how to include applications not necessarily specified in the `rel` tuple. Picking `include` means that Reltool will include pretty much everything it can find. Picking `derived` means that Reltool will include only applications that it detects can be used by any of the applications in your `rel` tuple. This is the default value. Picking `exclude` means that you will include no apps at all by default. You usually set this on a release level when you want minimal includes, and then override it on an application-by-application basis for the stuff you feel like adding.

`{mod_cond, all | app | ebin | derived | none}`
: This controls the module inclusion policy. Picking `none` means no modules will be kept (which isn't very useful). The `derived` option means that Reltool will try to figure out which modules are used by other modules that are already included and add them. Setting the option to `app` means that Reltool keeps all the modules mentioned in the app file and those that were derived. Setting it to `ebin` keeps those in the *ebin/* directory and the derived ones. Using the option `all`, which is the default, is a mix of using `ebin` and `app`.

`{app_file, keep | strip | all}`
: This option handles how the app files are going to be managed for you when you include an application. Picking `keep` guarantees that the app file used in the release is the same one you wrote for your application. That's the default option. If you choose `strip`, Reltool will try to generate a new app file that removes the modules you don't want in there (those

that were excluded by filters and other options). Choosing `all` keeps the original file, but also adds specifically included modules. The nice thing with `all` is that it can generate app files for you if none are available.

Module-Specific Options

The following are Reltool module-specific options.

`{incl_cond, include | exclude | derived}`
: This lets you override the `mod_cond` option defined at the release level and application level.

All-levels Options

The following options work on all levels. The lower the level, the more precedence it takes.

`{debug_info, keep | strip}`
: Assuming your files were compiled with `debug_info` on (which I suggest), this option lets you decide whether to keep that information or drop it. The `debug_info` is useful when you want to decompile files or debug them, but will take some space.

That's Dense

Oh yes, we've covered a lot of information about Reltool options. I didn't include all the possible options, but it's still a decent reference. If you want the whole thing, check out the official documentation at *http://www.erlang.org/doc/man/reltool.html*.

Reltool Recipes

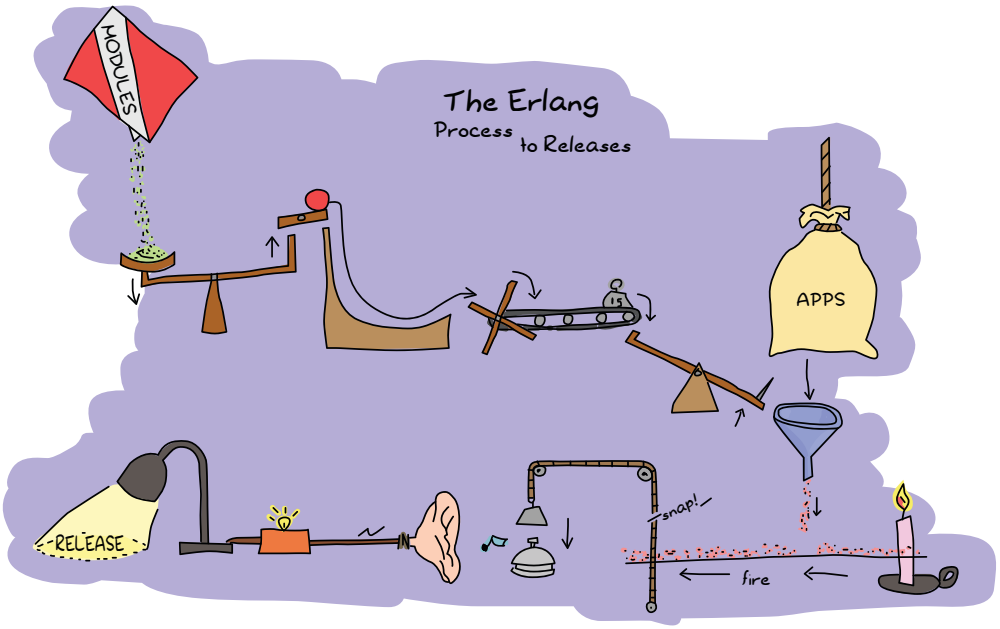
Now we'll consider a few general tips and tricks on how to write your *.rel* files in order to obtain specific results, such as small-sized releases or releases that contain enough libraries to allow development work.

Development Versions

Getting a release packed with libraries useful for development of specific projects should be relatively easy, and often the defaults are good enough. Just stick to getting the basic items we've covered so far, and you should be in good shape.

```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
    {rel, "erlcount", "1.0.0", [kernel, stdlib, ppool, erlcount]},
    {boot_rel, "erlcount"}
]}.
```

Reltool will take care of importing enough libraries for the new release to be fine. In some cases, you might want to have everything from a regular



VM as created by the OTP team, plus some of your own libraries. You might be distributing an entire VM for a team, with some libraries included. In that case, what you want to do is something more like this:

```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
    {rel, "start_clean", "1.0.0", [kernel, stdlib]},
    {incl_cond, include},
    {debug_info, keep}
]}.
```

By setting `incl_cond` to `include`, all applications found in the current ERTS installation and the `lib_dirs` will be part of your release.

NOTE *When no `boot_rel` is specified, you need to have a release named `start_clean` for Reltool to be happy. That one will be picked by default when you start the associated `erl` executable.*

If we want to exclude a specific application—let's say `megaco` because I never looked into it—we can instead get a file, like this:

```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
    {rel, "start_clean", "1.0.0", [kernel, stdlib]},
    {incl_cond, include},
    {debug_info, keep},
    {app, megaco, [{incl_cond, exclude}]}
]}.
```

Here, we can specify one or more applications (each having its own `app` tuple), and each of them overrides the `incl_cond` setting put at the release level. So, in this case, we will include everything except `megaco`.

Importing or Exporting Only Part of a Library

In our release, one annoying thing that happened was that apps like `ppool` and others also kept their test files in the release, even though they didn't need them. You can see them by going into *rel/lib/* and unzipping *ppool-1.0.0.ez* (you might need to change the extension first).

The easiest way to get rid of these files is to specify exclusion filters, such as the following:

```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
    {rel, "start_clean", "1.0.0", [kernel, stdlib, ppool, erlcount]},
    {excl_app_filters, ["_tests.beam$"]}
]}.
```

When you want to import only specific files of an application, things get a bit more complex. Here's an example of importing only `erlcount_lib` for its functionality, but nothing else from `erlcount`:

```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
    {rel, "start_clean", "1.0.0", [kernel, stdlib]},
    {incl_cond, derived}, % Exclude would also work, but not include.
    {app, erlcount, [{incl_app_filters, ["^ebin/erlcount_lib.beam$"]},
                     {incl_cond, include}]}
]}.
```

In this case, we switched from `{incl_cond, include}` to the more restrictive `incl_conds`. This is because if you go large and rake everything in, then the only way to include a single library is to exclude all the others with an `excl_app_filters`. However, when our selection is more restrictive (in this case, we're `derived` and wouldn't include `erlcount` because it's not part of the `rel` tuple), we can specifically tell the release to include the `erlcount` app with only files that match the regular expression having to do with `erlcount_lib`. This prompts the question as to how to make the smallest release possible, right?

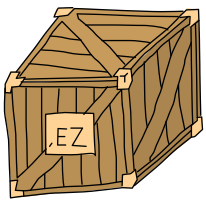
Smaller Apps for Programmers with Big Hearts

Release size reduction is where Reltool becomes a good bit more complex, with a rather verbose configuration file:

```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/release/"]},
    {erts, [{mod_cond, derived},
            {app_file, strip}]},
```

```
    {rel, "erlcount", "1.0.0", [kernel, stdlib, ppool, erlcount]},
    {boot_rel, "erlcount"},
    {relocatable, true},
    {profile, embedded},
    {app_file, strip},
    {debug_info, strip},
    {incl_cond, exclude},
    {excl_app_filters, ["_tests.beam$"]},
    {app, stdlib, [{mod_cond, derived}, {incl_cond, include}]},
    {app, kernel, [{incl_cond, include}]},
    {app, ppool, [{vsn, "1.0.0"}, {incl_cond, include}]},
    {app, erlcount, [{vsn, "1.0.0"}, {incl_cond, include}]}
]}.
```

A lot more stuff is going on here. We can see that in the case of `erts`, we ask for Reltool to keep only what's necessary. Setting `mod_cond` to `derived` and `app_file` to `strip` will ask Reltool to check and keep only what's used for something else. That's why `{app_file, strip}` is also used on the release level.



The profile is set to `embedded`. If you looked at the *.ez* archives in the previous cases, they contained the source files, test directories, and so on. When switching over to `embedded`, only include files, binaries, and the *priv/* directories are kept. We're also removing `debug_info` from all files, even if they were compiled with it. This means we're going to lose some debugging ability, but it will reduce the size of files.

We're still stripping away test files, and setting things so that no application is included until explicitly told to be (`{incl_cond, exclude}`). Then we override this setting in each app we do want to include. If something is missing, Reltool will warn you, so you can try to move things around and play with settings until you get the results you want. It might involve having some application settings with `{mod_cond, derived}`, as we did with the `stdlib`, so that the minimal files of some applications are what is kept.

What's the difference in the end? Some of our more general releases would weigh in at more than 35MB. The one described here is reduced to less than 20MB. We're shaving off a good part of it, although it's still fairly large. That's because of ERTS, which itself takes around 18.5MB. If you want to, you can dig deeper and really micromanage how ERTS is built to get something smaller. You can alternatively pick and delete some binary files in the ERTS that you know won't be used by your application: executables for scripts, remote running of Erlang, binaries from test frameworks, and different running commands (such as Erlang with or without SMP).

The lightest release will be the one that assumes that other users have Erlang installed already. When you pick this option, you need to add the *rel/* directory's content as part of your `ERL_LIBS` environment variable and call the boot file yourself (a bit like with `systools`), but it will work. Programmers might want to wrap this up in scripts to get things going.

NOTE *These days, Erlang programmers seem to really love the idea of having all these releases handled for them by a tool called rebar, which will act as a wrapper over Emakefile files and Reltool. There is no loss in understanding how Reltool works. The rebar tool uses configuration files that are nearly the same, and the gap between the two tools isn't that big.*

Released from Releases

Well, that's it for the two major ways to handle releases. It's a complex topic, but a standard way to handle distributions. Applications might be enough for many readers, and there's nothing wrong with sticking to them for a good while. However, now and then, releases might be useful if you want your operations guy to like you a bit better, given you know (or at least have some idea about) how to deploy Erlang applications when necessary.

Of course, what could make your operations guy happier than no downtime? The next challenge will be to do software upgrades while a release is running.

22

LEVELING UP IN THE PROCESS QUEST

Code hot-loading is simple in Erlang. You recompile, make a fully qualified function call, and then enjoy. However, doing it the right (and safe) way is much more difficult.

A plethora of things can go wrong in practice.

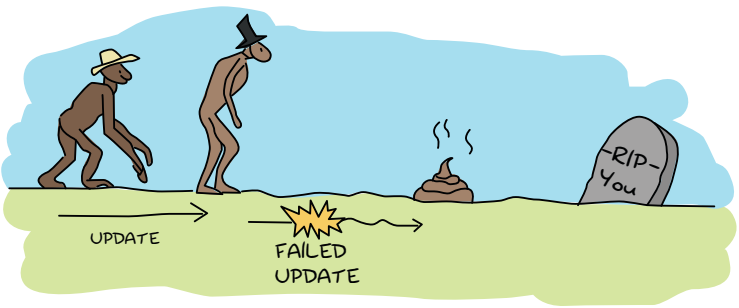
In this chapter, we'll explore the problems that hot code loading can bring and some principles that prove helpful to solve them. Then we'll go through a practical demonstration of how to take an existing OTP release, upgrade it, and reload the new version of it while it runs with the help of OTP mechanisms, through *appups* and *relups* (application and release upgrades).

NOTE *This chapter does not include all of the code used in the examples. Before you start this chapter, you might want to get the required code from* http://learnyousomeerlang.com/static/erlang/processquest.zip. *If you've downloaded the whole code package from the* Learn You Some Erlang *website, you already have everything you need.*

The Hiccups of Appups and Relups

One very simple challenge makes code reloading problematic. To understand it, let's use our amazing Erlang-programming brain and imagine a `gen_server` process. This process has a `handle_cast/2` function that accepts one kind of argument. You update it to one that takes a different kind of argument, compile it, and push it in production. All is fine and dandy, but because you have an application that you don't want to shut down, you decide to load it on the production VM to make it run.

Then a bunch of error reports start pouring in.



It turns out that your different `handle_cast` functions are incompatible. So when they were called a second time, no clause matched. The customer is pissed off and so is your boss. Then the operations guy is also angry because he needs to get on location and roll back the code, extinguish fires, and so on. If you're lucky, you're that operations guy. You're staying late and ruining the janitor's night (he likes to hum along with his music and dance a little as he works, but he is ashamed to do that in your presence). You come home late, your spouse/friends/World of Warcraft raid party/children are mad at you. They yell, scream, slam the door, and you're left alone. You had promised that nothing could go wrong—no downtime. You're using Erlang after all, right? But it didn't happen like that. You're alone, curled up in a ball in the corner of the kitchen, eating frozen Hot Pockets.

Of course, things aren't always that bad, but the point stands. Doing live code upgrades on a production system can be very dangerous if you're changing the interface your modules give to the world, changing internal data structures, changing function names, modifying records (remember that they're tuples!), and so on. These all have the potential to cause a crash.

When we were first playing with code reloading in Chapter 13, we had a process with some kind of hidden message to handle doing a fully qualified call. If you recall, a process could have looked like this:

```
loop(N) ->
    receive
        some_standard_message -> N+1;
```

```
        other_message -> N-1;
        {get_count, Pid} ->
            Pid ! N,
            loop(N);
        update -> ?MODULE:loop(N);
    end.
```

However, this way of doing things wouldn't fix our problems if we were to change the arguments to `loop/1`. We would need to extend it a bit, like this:

```
loop(N) ->
    receive
        some_standard_message -> N+1;
        other_message -> N-1;
        {get_count, Pid} ->
            Pid ! N,
            loop(N);
        update -> ?MODULE:code_change(N);
    end.
```

And then `code_change/1` could take care of calling a new version of the loop. But this kind of trick wouldn't work with generic loops. Consider this example:

```
loop(Mod, State) ->
    receive
        {call, From, Msg} ->
            {reply, Reply, NewState} = Mod:handle_call(Msg, State),
            From ! Reply,
            loop(Mod, NewState);
        update ->
            {ok, NewState} = Mod:code_change(State),
            loop(Mod, NewState)
    end.
```

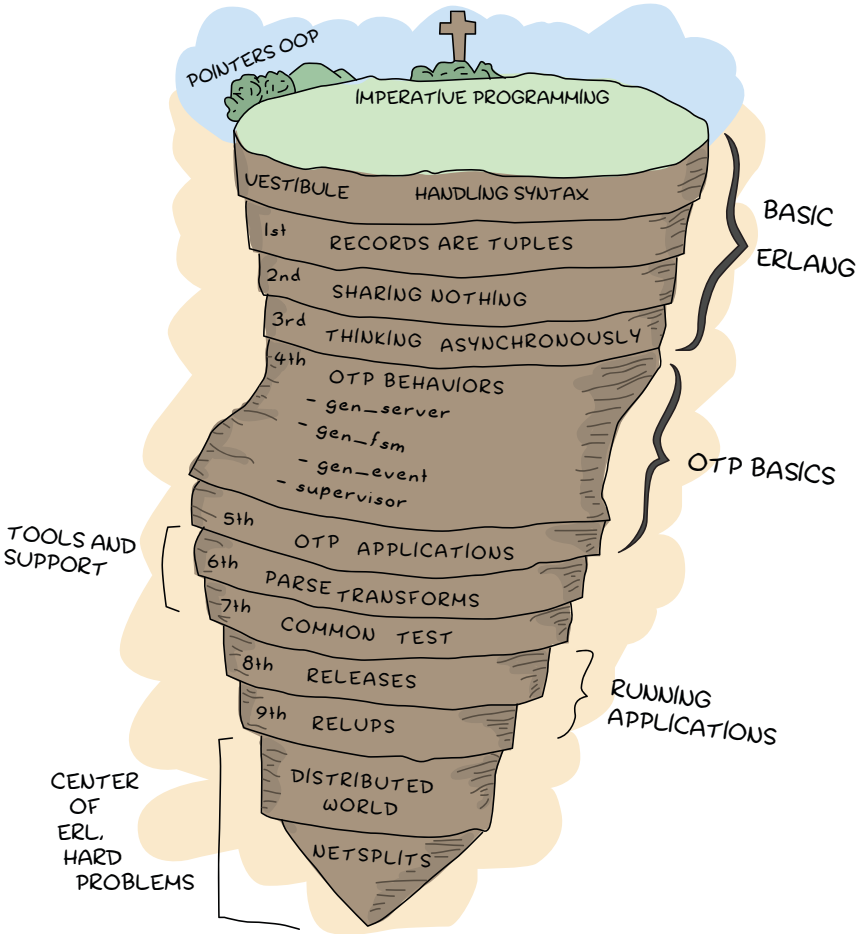
Do you see the problem? If we want to update `Mod` and load a new version, there is no way to do it safely with that implementation. The call `Mod:handle_call(Msg, State)` is already fully qualified, and it's possible that a message of the form `{call, From, Msg}` could be received in between the time we reload the code and handle the `update` message. In that case, we would update the module in an uncontrolled manner. Then we would crash.

The secret to getting it right is buried within the entrails of OTP. We must freeze the sands of time! To do so, we require more secret messages: messages to put a process on hold, messages to change the code, and then messages to resume the actions we had before. Deep inside OTP behaviors is hidden a special protocol to take care of all that kind of management. This is handled through the `sys` module and the `release_handler` module, which is part of the System Architecture Support Libraries (SASL) application. They take care of everything.

The trick is that you can call `sys:suspend(PidOrName)` to suspend OTP processes (you can find all of the processes by using the supervision trees and looking at the children each supervisor has). Then you use `sys:change_code(PidOrName, Mod, OldVsn, Extra)` to force the process to update itself. Finally, you call `sys:resume(PidOrName)` to make things go again.

It wouldn't be very practical for us to call these functions manually by writing ad hoc scripts all the time. Instead, we can look at how relups are done.

The Ninth Circle of Erl



The act of taking a running release, making a second version of it, and updating it while it runs is perilous. What seems like a simple assembly of *appups* (files containing instructions on how to update individual applications) and *relups* (files containing instructions to update an entire release) quickly turns into a struggle through APIs and undocumented assumptions.

We're getting into one of the most complex parts of OTP, a part that is difficult to comprehend and get right, on top of being time-consuming. In fact, if you can avoid the whole procedure (which will be called *relup* from now on) and do simple rolling upgrades by restarting VMs and booting new applications, I recommend you do so. Relups should be one of these "do or die" tools—something you use when you have few other choices.

There are a bunch of steps to execute when dealing with release upgrades, each of which can be more complex than the preceding one:

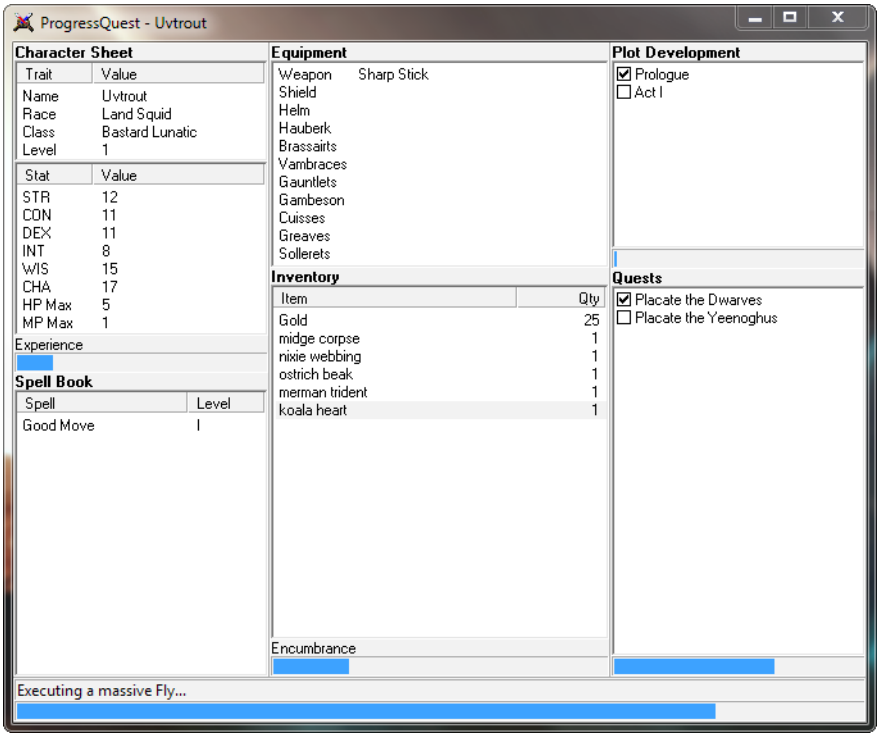
1. Write OTP applications.
2. Turn a bunch of them into a release.
3. Create new versions of one or more of the OTP applications.
4. Create an appup file that explains what to change to make the transition between the old and the new application work.
5. Create a new release with the new applications.
6. Generate an appup file from these releases.
7. Install the new app in a running Erlang shell.

We've only covered how to do the first three steps so far. To demonstrate how to work with an application that is more adapted to long-running upgrades than the previous ones (eh, who cares about running regular expressions without restarting?), we'll introduce a superb video game.

Process Quest

Progress Quest (*http://progressquest.com/*) is a revolutionary role-playing game (RPG). I would call it the OTP of RPGs, in fact. If you've ever played an RPG before, you'll notice that many steps are similar: run around, kill enemies, gain experience, get money, level up, get skills, complete quests—rinse and repeat forever. Power players will have shortcuts such as macros or even bots to go around and do their bidding for them.

Progress Quest took all of these generic steps and turned them into one streamlined game where you can sit back and enjoy your character doing all the work:



With the permission of the creator of this fantastic game, Eric Fredricksen, I've made a very minimal Erlang clone of it called Process Quest. Process Quest is similar in principle to Progress Quest, but rather than being a single-player application, it's a server able to hold many raw socket connections (usable through *telnet*) to let someone use a terminal and temporarily play the game.

The game is made up of the `regis`, `processquest`, and `sockserv` applications.

The regis-1.0.0 Application

The `regis` application is a process registry. It has an interface somewhat similar to the regular Erlang process registry, but it can accept any term at all and is meant to be dynamic. It might make things slower because all the calls will be serialized when they enter the server, but it will be better than using the regular process registry, which is not made for this kind of dynamic work. If this book could automatically update itself with external libraries (it's too much work), I would have used `gproc` instead.

The application has three modules: `regis.erl`, `regis_server.erl`, and `regis_sup.erl`. The first one is a wrapper around the other two (and an application callback module). `regis_server` is the main registration `gen_server`, and `regis_sup` is the application's supervisor.

The processquest-1.0.0 Application

The `processquest` application is the core of the release. It includes all the game logic—enemies, market, killing fields, and statistics. The player itself is a `gen_fsm` that sends messages to itself in order to keep going all the time. It contains the following modules:

`pq_enemy.erl`
: This module randomly picks an enemy to fight, of the form `{<<"Name">>, [{drop, {<<"DropName">>, Value}}, {experience, ExpPoints}]}`. This lets the player fight an enemy.

`pq_market.erl`
: This implements a market that allows players to find items of a given value and a given strength. All items returned are of the form `{<<"Name">>, Modifier, Strength, Value}`. There are functions to fetch weapons, armors, shields, and helmets.

`pq_stats.erl`
: This is a small attribute generator for your character.

`pq_events.erl`
: This is a wrapper around a `gen_event` event manager. This acts as a generic hub to which subscribers connect themselves with their own handlers to receive events from each player. It also takes care of waiting a given delay for the player's actions to avoid the game being instantaneous.

`pq_player.erl`
: This is the central module. It is a `gen_fsm` that goes through the state loop of killing, then going to the market, then killing again, and so on. It uses all of the preceding modules to function.

`pq_sup.erl`
: This is a supervisor that sits above a pair of `pq_event` and `pq_player` processes. They both need to be together in order to work; otherwise, the player process is useless and isolated or the event manager will never get any events.

pq_supersup.erl
: This is the top-level supervisor of the application. It sits over a bunch of `pq_sup` processes. This lets you spawn as many players as you like.

processquest.erl
: This is a wrapper and application callback module. It gives the basic interface to a player. You start one, and then subscribe to events.

The sockserv-1.0.0 Application

The `sockserv` application is a customized raw socket server, made to work only with the `processquest` app. It will spawn `gen_servers`, each in charge of a TCP socket that will push strings to some client. Again, you may use telnet to work with it. (Telnet was technically not made for raw socket connections and is its own protocol, but most modern clients accept it without any problems.) Here are `sockserv`'s modules:

sockserv_trans.erl
: This translates messages received from the player's event manager into printable strings.

sockserv_pq_events.erl
: This is a simple event handler that takes whatever events come from a player and casts them to the socket `gen_server`.

sockserv_serv.erl
: This is a `gen_server` in charge of accepting a connection, communicating with a client, and forwarding information to it.

sockserv_sup.erl
: This supervises a bunch of socket servers.

sockserv.erl
: This is an application callback module for the app as a whole.

The Release

I've set everything up in a directory called *processquest* with the following structure:

```
apps/
 - processquest-1.0.0
   - ebin/
   - src/
   - ...
 - regis-1.0.0
   - ...
```

```
 - sockserv-1.0.0
   - ...
rel/
  (will hold releases)
processquest-1.0.0.config
```

Based on that, we can build a release.

> **MORE RELEASE CONFIGURATION**
>
> If you look into *processquest-1.0.0.config*, you will see that applications such as Crypto and SASL are included. Crypto is necessary to have good initialization of pseudo-random number generators, and SASL is mandatory to be able to do appups on a system. *If you forget to include SASL in your release, it will be impossible to upgrade the system.*
>
> A new filter has appeared in the configuration file: `{excl_archive_filters, [".*"]}`. This filter makes sure that no *.ez* file is generated; only regular files and directories are created. This is necessary because the tools we're going to use cannot look into *.ez* files to find the items they need.
>
> You will also see that there are no instructions asking to strip the `debug_info`. Without `debug_info`, doing an appup will fail for some reason. It's always useful to have `debug_info` anyway.

Following Chapter 21's instructions for releases, we start by calling `erl -make` for all applications. Once this is done, start an Erlang shell from the *processquest* directory and enter the following:

```
1> {ok, Conf} = file:consult("processquest-1.0.0.config"),
1> {ok, Spec} = reltool:get_target_spec(Conf),
1> reltool:eval_target_spec(Spec, code:root_dir(), "rel").
ok
```

We should have a functional release. Let's try it. Start any version of the VM by entering `./rel/bin/erl -sockserv port 8888` (or any other port number you want; the default is 8082). This will show a lot of logs about processes being started (that's one of the functions of SASL), and then a regular Erlang shell. Start a telnet session on your localhost using whatever client you want:

```
$ telnet localhost 8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
What's your character's name?
hakvroot
```

```
Stats for your character:
  Charisma: 7
  Constitution: 12
  Dexterity: 9
  Intelligence: 8
  Strength: 5
  Wisdom: 16

Do you agree to these? y/n
```

That's a bit too much wisdom and charisma for me. So, I type **n** and hit ENTER:

```
n
Stats for your character:
  Charisma: 6
  Constitution: 12
  Dexterity: 12
  Intelligence: 4
  Strength: 6
  Wisdom: 10

Do you agree to these? y/n
```

Yes, that's ugly, dumb, and weak—exactly what I'm looking for in a hero based on me.

```
y
Executing a Wildcat...
Obtained Pelt.
Executing a Pig...
Obtained Bacon.
Executing a Wildcat...
Obtained Pelt.
Executing a Robot...
Obtained Chunks of Metal.
...
Executing a Ant...
Obtained Ant Egg.
Heading to the marketplace to sell loot...
Selling Ant Egg
Got 1 bucks.
Selling Chunks of Metal
Got 1 bucks.
...
Negotiating purchase of better equipment...
Bought a plastic knife
Heading to the killing fields...
Executing a Pig...
Obtained Bacon.
Executing a Ant...
```

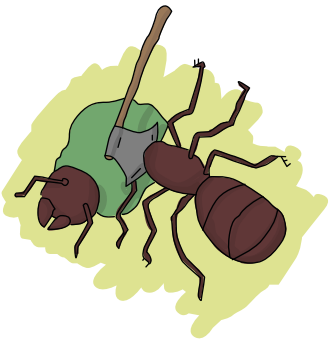
That's enough for me. I type **quit** and then press ENTER to close the connection:

```
quit
Connection closed by foreign host.
```

If you want, you can leave it open, see yourself level up, gain stats, and so on. The game basically works, and you can try it with many clients. It should keep going without a problem.

Awesome, right? Well, we can do better.

Making Process Quest Better



There are a few issues with the current versions of the applications of Process Quest. First, we have very little variety in terms of enemies to beat. Second, we have text that looks a bit weird (what is it with "Executing a Ant..."?). A third issue is that the game is a bit too simple; let's add a mode for quests! Another one is that the value of items you sell at the market is directly bound to your level in Progress Quest, while Process Quest doesn't do anything like that. Finally—and you can't see this unless you read the code and try to close the client on your own end—a client closing its connection will leave the player process alive on the server. Uh-oh, memory leaks!

We'll need to fix these problems!

NOTE *I started by making a new copy of both applications that need fixes. I then had* processquest-1.1.0 *and* sockserv-1.0.1 *on top of the others. (I use the version scheme of* MajorVersion.Enhancements.BugFixes.*) In the copies, I implemented all the changes I needed. I won't go through all of them here, because we're here to upgrade an app, not to walk through all the details and intricacies of this specific app. In case you do want to know all the little intricacies, I commented the code so you can find the information you need to understand it.*

Updating code_change Functions

In `processquest-1.1.0`, changes were made to *pq_enemy.erl*, *pq_events.erl*, and *pq_player.erl*. I also added a file named *pq_quest.erl*, which implements quests based on how many enemies were killed by a player. Of these files, only *pq_player.erl* had incompatible changes that required a time suspension. This record:

```
-record(state, {name, stats, exp=0, lvlexp=1000, lvl=1,
                equip=[], money=0, loot=[], bought=[], time=0}).
```

was changed to this one:

```
-record(state, {name, stats, exp=0, lvlexp=1000, lvl=1,
                equip=[], money=0, loot=[], bought=[],
                time=0, quest}).
```

where the `quest` field will hold a value given by `pq_quest:fetch/0`.

Because of that change, we'll need to modify the `code_change/4` function in version 1.1.0. In fact, we'll need to modify it twice: once in the case of an upgrade (moving from 1.0.0 to 1.1.0) and another in the case of a downgrade (1.1.0 to 1.0.0). Fortunately, OTP will pass different arguments in each case. When we upgrade, we get a version number for the module. We don't exactly care for that one at this point, and we'll likely just ignore it. When we downgrade, we get `{down, Version}`. This lets us easily match on each operation:

```
code_change({down, _}, StateName, State, _Extra) ->
    ...;
code_change(_OldVsn, StateName, State, _Extra) ->
    ...
```

But hold on a second! We can't just blindly take the state as we usually do. We need to upgrade it. The problem is we can't do something like this:

```
code_change(_OldVsn, StateName, S = #state{}, _Extra) ->
    ...
```

We have two options. The first is to declare a new state record that will have a new form and a new name. We would end up having something like this:

```
-record(state, {...}).
-record(new_state, {...}).
```

And then we would need to change the record in each of the function clauses of the module. That's annoying and not worth the risk. It will be simpler to expand the record to its underlying tuple form (remember the discussion of records in Chapter 9), as follows:

```
code_change({down, _},
            StateName,
            #state{name=N, stats=S, exp=E, lvlexp=LE, lvl=L, equip=Eq,
                   money=M, loot=Lo, bought=B, time=T},
            _Extra) ->
    Old = {state, N, S, E, LE, L, Eq, M, Lo, B, T},
    {ok, StateName, Old};
code_change(_OldVsn,
            StateName,
            {state, Name, Stats, Exp, LvlExp, Lvl, Equip, Money, Loot,
             Bought, Time},
```

```
        _Extra) ->
    State = #state{
        name=Name, stats=Stats, exp=Exp, lvlexp=LvlExp,
        lvl=Lvl, equip=Equip, money=Money, loot=Loot,
        bought=Bought, time=Time, quest=pq_quest:fetch()
    },
    {ok, StateName, State}.
```

And there's our `code_change/4` function! All it does is convert between both tuple forms. For new versions, we also take care of adding a new quest—it would be boring to add quests but have all our existing players unable to use them.

You'll notice that we still ignore the `_Extra` variable. This one is passed from the appup file (described next), and you'll be the one to pick its value. For now, we don't care because we can upgrade and downgrade to and from only one release. In some more complex cases, you might want to pass release-specific information in there.

For the `sockserv-1.0.1` application, only *sockserv_serv.erl* required changes. Fortunately, they didn't need a restart, and only added a new message to match on.

The two versions of the two applications have been fixed. That's not enough to go on our merry way, though. We need to find a way to let OTP know what kinds of changes require different kinds of actions.

Adding Appup Files

Appup files are lists of Erlang commands that need to be done to upgrade a given application. They contain lists of tuples and atoms telling what to do and in what case. The general format for them is as follows:

```
{NewVersion,
 [{VersionUpgradingFrom, [Instructions]}]
 [{VersionDownGradingTo, [Instructions]}]}.
```

They ask for lists of versions because it's possible to upgrade and downgrade to many different versions. In our case, for `processquest-1.1.0`, this would be the following:

```
{"1.1.0",
 [{"1.0.0", [Instructions]}],
 [{"1.0.0", [Instructions]}]}.
```

The instructions contain both high-level and low-level commands. We usually need to care about only high-level ones, though:

`{add_module, Mod}`
: The module `Mod` is loaded for the first time.

`{load_module, Mod}`
: The module `Mod` is already loaded in the VM and has been modified.

{delete_module, Mod}
: The module `Mod` is removed from the VM.

{update, Mod, {advanced, Extra}}
: This will suspend all processes running `Mod`, call the `code_change` function of your module with `Extra` as the last argument, and then resume all processes running `Mod`. `Extra` can be used to pass in arbitrary data to the `code_change` function, in case it's required for upgrades.

{update, Mod, supervisor}
: Calling this lets you redefine the `init` function of a supervisor to influence its restart strategy (`one_for_one`, `rest_for_one`, and so on) or change child specifications (this will not affect existing processes).

{apply, {M, F, A}}
: This will call `apply(M,F,A)`.

Module dependencies
: You can use `{load_module, Mod, [ModDependencies]}` or `{update, Mod, {advanced, Extra}, [ModDeps]}` to make sure that a command happens only after some other modules were handled beforehand. This is especially useful if `Mod` and its dependencies are *not* part of the same application. There is sadly no way to give similar dependencies to `delete_module` instructions.

Note that when generating relups, we won't need any special instructions to remove or add applications. The function that generates relup files (files to upgrade releases) will take care of detecting this for us.

Using these instructions, we can write the two following appup files for our applications. The file must be named *NameOfYourApp.appup* and be put in the app's *ebin/* directory.

Here's `processquest-1.1.0`'s appup file:

```
{"1.1.0",
 [{"1.0.0", [{add_module, pq_quest},
             {load_module, pq_enemy},
             {load_module, pq_events},
             {update, pq_player, {advanced, []}, [pq_quest, pq_events]}]}],
 [{"1.0.0", [{update, pq_player, {advanced, []}},
             {delete_module, pq_quest},
             {load_module, pq_enemy},
             {load_module, pq_events}]}]}.
```

You can see that we need to add the new module, load the two ones that require no suspension, and then update `pq_player` in a safe manner. When we downgrade the code, we do the same thing, but in reverse. The funny thing is that in one case, `{load_module, Mod}` will load a new version, and in the other, it will load the old version. It all depends on the context between an upgrade and a downgrade.

Because sockserv-1.0.1 had only one module to change and it required no suspension, its appup file is brief, as follows:

```
{"1.0.1",
 [{"1.0.0", [{load_module, sockserv_serv}]}],
 [{"1.0.0", [{load_module, sockserv_serv}]}]}.
```

The next step is to build a new release using the new modules. Here's the file *processquest-1.1.0.config*:

```
{sys, [
    {lib_dirs, ["/home/ferd/code/learn-you-some-erlang/processquest/apps"]},
    {erts, [{mod_cond, derived},
            {app_file, strip}]},
    {rel, "processquest", "1.1.0",
     [kernel, stdlib, sasl, crypto, regis, processquest, sockserv]},
    {boot_rel, "processquest"},
    {relocatable, true},
    {profile, embedded},
    {app_file, strip},
    {incl_cond, exclude},
    {excl_app_filters, ["_tests.beam"]},
    {excl_archive_filters, [".*"]},
    {app, stdlib, [{mod_cond, derived}, {incl_cond, include}]},
    {app, kernel, [{incl_cond, include}]},
    {app, sasl, [{incl_cond, include}]},
    {app, crypto, [{incl_cond, include}]},
    {app, regis, [{vsn, "1.0.0"}, {incl_cond, include}]},
    {app, sockserv, [{vsn, "1.0.1"}, {incl_cond, include}]},
    {app, processquest, [{vsn, "1.1.0"}, {incl_cond, include}]}
]}.
```

It's just a copy/paste of the old one with a few versions changed. First, compile both new applications with `erl -make` (the new versions are in the zip file mentioned earlier). Then we can generate a new release. First, compile the two new applications, and then enter the following:

```
$ erl -env ERL_LIBS apps/
1> {ok, Conf} = file:consult("processquest-1.1.0.config"),
1> {ok, Spec} = reltool:get_target_spec(Conf),
1> reltool:eval_target_spec(Spec, code:root_dir(), "rel").
ok
```

But wait, there's more manual work required!

1. Copy *rel/releases/1.1.0/processquest.rel* as *rel/releases/1.1.0/processquest-1.1.0.rel*.
2. Copy *rel/releases/1.1.0/processquest.boot* as *rel/releases/1.1.0/processquest-1.1.0.boot*.
3. Copy *rel/releases/1.1.0/processquest.boot* as *rel/releases/1.1.0/start.boot*.
4. Copy *rel/releases/1.0.0/processquest.rel* as *rel/releases/1.0.0/processquest-1.0.0.rel*.

5. Copy *rel/releases/1.0.0/processquest.boot* as *rel/releases/1.0.0/processquest-1.0.0.boot*.
6. Copy *rel/releases/1.0.0/processquest.boot* as *rel/releases/1.0.0/start.boot*.

> **DON'T DRINK TOO MUCH KOOL-AID**
>
> Why didn't we just use `systools` to build our release? Well, `systools` has its share of issues. First, it will generate appup files that sometimes have weird versions in them and won't work perfectly. It will also assume a directory structure that is barely documented, but somewhat close to what Reltool uses. The biggest issue, though, is that it will use your default Erlang installation as the root directory, which might create all kinds of permission issues and whatnot when the time comes to unpack stuff.
>
> There's just no easy way to build the release with either tool, and it will require a lot of manual work. We thus make a chain of commands that uses both modules in a rather complex manner, because it ends up being a bit less work.

Now we can generate the relup file. To do this, start an Erlang shell and call the following:

```
erl -env ERL_LIBS apps/ -pa apps/processquest-1.0.0/ebin/ -pa apps/sockserv-1.0.0/ebin/
1> systools:make_relup("./rel/releases/1.1.0/processquest-1.1.0",
1>                     ["rel/releases/1.0.0/processquest-1.0.0"],
1>                     ["rel/releases/1.0.0/processquest-1.0.0"]).
ok
```

Because the `ERL_LIBS` environment variable will look for only the newest versions of applications, we also need to add the `-pa` *`Path_to_older_applications`* in there so that the `systools` relup generator will be able to find everything. Once this is done, move the relup file to *rel/releases/1.1.0/*. That directory will be looked into when updating the code in order to find the right stuff. One problem we'll have, though, is that the release handler module will depend on a bunch of files it assumes to be present, but won't necessarily be there.

Upgrading the Release

Sweet—we have a relup file. There's still stuff to do before being able to use it though. The next step is to generate a tar file for the whole new version of the release:

```
2> systools:make_tar("rel/releases/1.1.0/processquest-1.1.0").
ok
```

The file will be in *rel/releases/1.1.0/*. We now need to manually move it to *rel/releases*, and rename it to add the version number when doing so. More hard-coded junk! Here's our way out of this:

```
$ mv rel/releases/1.1.0/processquest-1.1.0.tar.gz rel/releases/
```

The next step is a step you want to do at *any time before you start the real production application*. This step needs to be done before you start the application, as it will allow you to roll back to the initial version after a relup. If you do not do this, you will be able to downgrade production applications only to releases newer than the first one, but not the first one!

Open a shell and run this:

```
1> release_handler:create_RELEASES(
1>     "rel",
1>     "rel/releases",
1>     "rel/releases/1.0.0/processquest-1.0.0.rel",
1>     [{kernel,"2.14.4", "rel/lib"}, {stdlib,"1.17.4","rel/lib"},
1>      {crypto,"2.0.3","rel/lib"},{regis,"1.0.0", "rel/lib"},
1>      {processquest,"1.0.0","rel/lib"},{sockserv,"1.0.0", "rel/lib"},
1>      {sasl,"2.1.9.4", "rel/lib"}]
1> ).
```

The general format of the function is as follows:

```
release_handler:create_RELEASES(RootDir, ReleasesDir, Relfile, [{AppName, Vsn, LibDir}])
```

This will create a file named *RELEASES* inside the *rel/releases* directory (or any other *ReleasesDir*), which will contain basic information on your releases when relup is looking for files and modules to reload.

We can now start running the old version of the code. If you start *rel/bin/erl*, it will start the 1.1.0 release by default. That's because we built the new release before starting the VM. For this demonstration, we'll need to start the release with this command:

```
$ ./rel/bin/erl -boot rel/releases/1.0.0/processquest
```

You should see everything starting up. Start a telnet client to connect to your socket server so you can see the live upgrade taking place.

Whenever you feel ready for an upgrade, go to the Erlang shell currently running Process Quest and call the following function:

```
1> release_handler:unpack_release("processquest-1.1.0").
{ok,"1.1.0"}
2> release_handler:which_releases().
[{"processquest","1.1.0",
  ["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",
   "regis-1.0.0","processquest-1.1.0","sockserv-1.0.1",
   "sasl-2.1.9.4"],
  unpacked},
 {"processquest","1.0.0",
  ["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",
   "regis-1.0.0","processquest-1.0.0","sockserv-1.0.0",
   "sasl-2.1.9.4"],
  permanent}]
```

The second prompt here tells you that the release is ready to be upgraded, but not installed or made permanent yet. To install it, enter the following:

```
3> release_handler:install_release("1.1.0").
{ok,"1.0.0",[]}
4> release_handler:which_releases().
[{"processquest","1.1.0",
  ["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",
   "regis-1.0.0","processquest-1.1.0","sockserv-1.0.1",
   "sasl-2.1.9.4"],
  current},
 {"processquest","1.0.0",
  ["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",
   "regis-1.0.0","processquest-1.0.0","sockserv-1.0.0",
   "sasl-2.1.9.4"],
  permanent}]
```

Now release 1.1.0 should be running, but it's still not there forever. Still, you could keep your application running that way. Call the following function to make things permanent:

```
5> release_handler:make_permanent("1.1.0").
ok.
```

Ah, damn—a bunch of our processes are dying now (error output removed from the preceding sample). But if you look at our telnet client, it did seem to upgrade fine. The issue is that all the `gen_servers` that were waiting for connections in `sockserv` could not listen to messages because

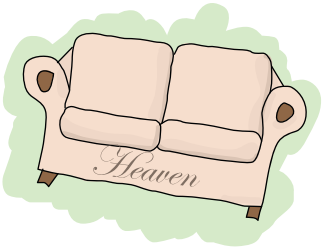
accepting a TCP connection is a blocking operation. Thus, the servers couldn't upgrade when new versions of the code were loaded and were killed by the VM. Here's how we can confirm this:

```
6> supervisor:which_children(sockserv_sup).
[{undefined,<0.51.0>,worker,[sockserv_serv]}]
7> [sockserv_sup:start_socket() || _ <- lists:seq(1,20)].
[{ok,<0.99.0>},
 {ok,<0.100.0>},
 ... <snip> ...
 {ok,<0.117.0>},
 {ok,<0.118.0>}]
8> supervisor:which_children(sockserv_sup).
[{undefined,<0.112.0>,worker,[sockserv_serv]},
 {undefined,<0.113.0>,worker,[sockserv_serv]},
 ... <snip> ...
 {undefined,<0.109.0>,worker,[sockserv_serv]},
 {undefined,<0.110.0>,worker,[sockserv_serv]},
 {undefined,<0.111.0>,worker,[sockserv_serv]}]
```

The first command shows that all children that were waiting for connections have already died. The processes left will be those with an active session going on. This shows the importance of keeping code responsive. Had our processes been able to receive messages and act on them, things would have been fine.

In the two last commands, we just start more workers to fix the problem. While this works, it requires manual action from the person running the upgrade. In any case, this is far from optimal.

A better way to solve the problem would be to change the way our application works in order to have a monitor process watching how many children `sockserv_sup` has. When the number of children falls under a given threshold, the monitor starts more of them.



Another strategy would be to change the code so accepting connections is done by blocking on intervals of a few seconds at a time, and keep retrying after pauses where messages can be received. This would give the `gen_servers` the time to upgrade themselves as required, assuming you would wait the right delay between the installation of a release and making it permanent. Implementing either or both of these solutions is left as an exercise to the reader (because I am somewhat lazy).

These kinds of crashes are why you want to test your code *before* doing these updates on a live system. If you want to really test your planned relups, you should be ready to test both for upgrades and downgrades, and restarting the node in case of failure, just to make sure.

In any case, we've solved the problem for now. Let's check how the upgrade procedure went:

```
9> release_handler:which_releases().
[{"processquest","1.1.0",
  ["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",
   "regis-1.0.0","processquest-1.1.0","sockserv-1.0.1",
   "sasl-2.1.9.4"],
  permanent},
 {"processquest","1.0.0",
  ["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",
   "regis-1.0.0","processquest-1.0.0","sockserv-1.0.0",
   "sasl-2.1.9.4"],
  old}]
```

That's worth a fist pump. You can try downgrading an installation by doing `release_handler:install(OldVersion).`. This should work fine, although it could risk killing more processes that never updated themselves.

> **DON'T DRINK TOO MUCH KOOL-AID**
>
> If rolling back always fails when trying to roll back to the first version of the release using the techniques shown in this chapter, you probably forgot to create the *RELEASES* file. You can tell this is the case if you see an empty list in `{YourRelease,Version,[],Status}` when calling `release_handler:which_releases()`. This is a list of where to find modules to load and reload, and it is first built when booting the VM and reading the *RELEASES* file, or when unpacking a new release.

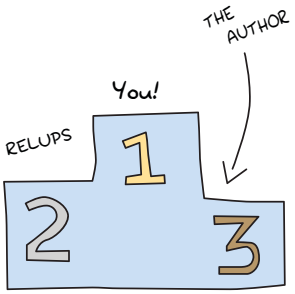
Relup Review

In summary, here's a list of all the actions that must be taken to have functional relups:

1. Write OTP applications for your first software iteration.
2. Compile them.
3. Build a release (1.0.0) using Reltool. It must have debug info and no *.ez* archive.
4. Make sure you create the *RELEASES* file at some point before starting your production application. You can do it with `release_handler:create_RELEASES(RootDir, ReleasesDir, Relfile, [{AppName, Vsn, LibDir}])`.
5. Run the release!
6. Find bugs in it.

7. Fix bugs in new versions of applications.
8. Write appup files for each of the applications.
9. Compile the new applications.
10. Build a new release (1.1.0 in our case). It must have debug info and no *.ez* archive.
11. Copy *rel/releases/NewVsn/RelName.rel* as *rel/releases/NewVsn/RelName-NewVsn.rel*.
12. Copy *rel/releases/NewVsn/RelName.boot* as *rel/releases/NewVsn/RelName-NewVsn.boot*.
13. Copy *rel/releases/NewVsn/RelName.boot* as *rel/releases/NewVsn/start.boot*.
14. Copy *rel/releases/OldVsn/RelName.rel* as *rel/releases/OldVsn/RelName-OldVsn.rel*.
15. Copy *rel/releases/OldVsn/RelName.boot* as *rel/releases/OldVsn/RelName-OldVsn.boot*.
16. Copy *rel/releases/OldVsn/RelName.boot* as *rel/releases/OldVsn/start.boot*.
17. Generate a relup file with `systools:make_relup("rel/releases/Vsn/RelName-Vsn", ["rel/releases/OldVsn/RelName-OldVsn"], ["rel/releases/DownVsn/RelName-DownVsn"])`.
18. Move the relup file to *rel/releases/Vsn*.
19. Generate a tar file of the new release with `systools:make_tar("rel/releases/Vsn/RelName-Vsn")`.
20. Move the tar file to *rel/releases/*.
21. Have some shell opened that still runs the first version of the release.
22. Call `release_handler:unpack_release("NameOfRel-Vsn")`.
23. Call `release_handler:install_release(Vsn)`.
24. Call `release_handler:make_permanent(Vsn)`.
25. Make sure things went fine. If not, roll back by installing an older version.

You might want to write a few scripts to automate this.



Again, relups are a very messy part of OTP—a part that is hard to grasp. You will likely find yourself finding plenty of new errors, which are all more impossible to understand than the previous ones. Some assumptions are made about how you're going to run things, and choosing different tools when creating releases will change how things should be done. You might even be tempted to write your own update code using the `sys` module's functions! Or maybe you'll want to use tools like rebar that automate some of the painful steps. In any case, this chapter and its examples have been written to the best knowledge of the author, a person who sometimes enjoys writing about himself in third person.

If it is possible to upgrade your application in ways that do not require relups, I recommend doing so. It is said that divisions of Ericsson that do use relups spend as much time testing them as they do testing their applications themselves. Relups are a tool to be used when working with products that can imperatively never be shut down. You will know when you will need them, mostly because you'll be ready to go through the hassle of using them (got to love that circular logic!). When the need arises, relups are entirely useful.

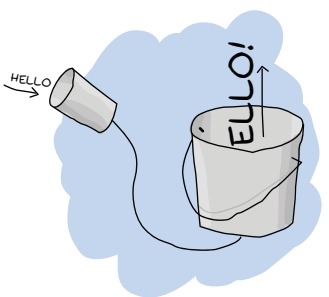
How about we move on to some of the friendlier features of Erlang now? Chapter 23 explores socket programming with Erlang.

23

BUCKETS OF SOCKETS

So far, we've had some fun dealing with Erlang itself, but we've barely communicated with the outside world. And even when we did, it was only by reading the occasional text file. Though relationships with yourself might be fun, it's time to get out of our lair and start talking to the rest of the world. One way to do this is by using sockets.

This chapter covers three components of using sockets: IO lists, UDP sockets, and TCP sockets.



IO Lists

As you've learned, for text, we can use either strings (lists of integers) or binaries (binary data structures that hold data).

Sending things over the wire such as “Hello World” can be done as a string (`"Hello World"`) or as a binary (`<<"Hello World">>`)—similar notation, similar results.

The difference lies in how you can assemble things.

A string is a bit like a linked list of integers. For each character, you need to store the character itself plus a link to the rest of the list. Moreover, if you want to add elements to a list—either in the middle or at the end—you must traverse the whole list up to the point you’re modifying, and then add your elements. This isn’t the case when you prepend, however:

```
A = [a]
B = [b|A] = [b,a]
C = [c|B] = [c,b,a]
```

In the case of prepending, whatever is held in `A`, `B`, or `C` never needs to be rewritten. The representation of `C` can be seen as `[c,b,a]`, `[c|B]`, or `[c,|[b|[a]]]` (among others). In the last case, you can see that the shape of `A` is the same at the end of the list as when it was declared, and it works similarly for `B`. Here’s how it looks with appending:

```
A = [a]
B = A ++ [b] = [a] ++ [b] = [a|[b]]
C = B ++ [c] = [a|[b]] ++ [c] = [a|[b|[c]]]
```

Do you see all that rewriting? When we create `B`, we need to rewrite `A`. When we write `C`, we must rewrite `B` (including the `[a|...]` part it contains). If we were to add `D` in a similar manner, we would need to rewrite `C`. Over long strings, this becomes way too inefficient, and it creates a lot of garbage left to be cleaned up by the Erlang VM.

With binaries, things are not quite as bad:

```
A = <<"a">>
B = <<A/binary, "b">> = <<"ab">>
C = <<B/binary, "c">> = <<"abc">>
```

In this case, binaries know their own length, and data can be joined in constant time. That’s good—much better than lists. They’re also more compact. For these reasons, we’ll try to stick to binaries when using text in the future.

There are a few downsides, however. Binaries were meant to handle things in certain ways, and there is still a cost to modifying binaries, splitting them, and so on. Moreover, sometimes we’ll work with code that uses strings, binaries, and individual characters interchangeably. Constantly converting between types would be a hassle.

In these cases, *IO lists* are our savior. IO lists are a weird type of data structure. They are lists of bytes (integers from 0 to 255), binaries, or other IO lists. This means that functions that accept IO lists can accept items such as `[$H, $e, [$l, <<"lo">>, " "], [[["W","o"], <<"rl">>]] | [<<"d">>]]`. When this happens, the Erlang VM will just flatten the list as it needs to in order to obtain the sequence of characters *Hello World*.

What are the functions that accept such IO lists? Most of the functions that have to do with outputting data will accept them, and any function from the `io` module, `file` module, TCP and UDP sockets will be able to handle them. Some library functions, such as several from the `unicode` module and all of the functions from the `re` (for regular expressions) module, will also handle them, as will many others.

Just to see, try the previous Hello World IO list in the shell with `io:format("~s~n", [IoList])`. It should work without a problem.

All in all, IO lists are a pretty clever way of building strings to avoid the problems of immutable data structures when it comes to dynamically building content to be output.

UDP and TCP: Bro-tocols

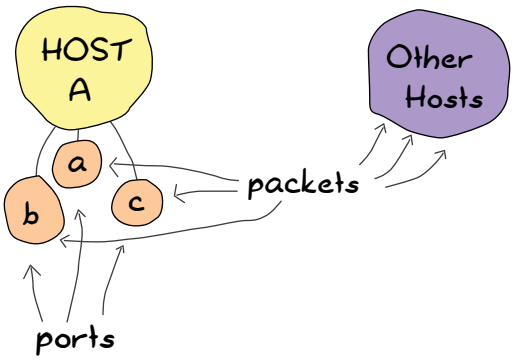
The first kind of socket that we can use in Erlang is based on the User Datagram Protocol (UDP). UDP is a protocol built on top of the IP layer that provides a few helpful abstractions, such as port numbers.

UDP is a *connectionless* protocol. The data that is received from the UDP port is broken into small untagged parts without a session (datagrams), and there is no guarantee that the fragments you received were sent in the same order as you got them. In fact, there is no guarantee that if someone sends a packet, you'll receive it at all. For these reasons, people tend to use UDP in the following situations:

- When the packets are small
- When packets can sometimes be lost with little consequences
- When there aren't too many complex exchanges taking place
- When low latency is absolutely necessary

This is in opposition to *connection-based* protocols like the Transmission Control Protocol (TCP), where the protocol takes care of handling lost packets, reordering them, maintaining isolated sessions between multiple senders and receivers, and so on. TCP allows reliable exchange of information, but can be slower and heavier to set up. UDP will be fast but less reliable. Choose carefully depending on what you need.

Using UDP in Erlang is relatively simple. We set up a socket over a given port, and that socket can both send and receive data:



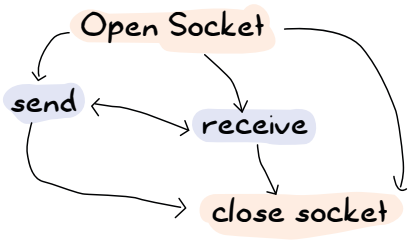
This is a bit like having a bunch of mailboxes for your house (each mailbox being a port) and receiving tiny slips of paper in each of them with small messages. They can have any content, from "I like how you look in these pants" to "The slip is coming from *inside* the house!" When some messages are too large for a slip of paper, many of them are dropped in the mailbox. It's your job to reassemble them in a way that makes sense, then drive up to some house, and drop slips as a reply. If the messages are purely informative ("Hey there, your door is unlocked") or very tiny ("What are you wearing? —Ron"), it should be fine, and you could use one mailbox for all of the queries. If they are complex, though, you might want to use one port per session, right? Ugh, no! Use TCP!

In the case of TCP, the protocol is said to be stateful, connection-based. Before being able to send messages, you must do a handshake. This means that someone is delivering messages to a mailbox (similar to what we have in the UDP analogy), and sending a message saying something like, "Hey dude, this is IP 94.25.12.37 calling. Wanna chat?" And you reply something like, "Sure. Tag your messages with number *N*, and then add an increasing number to them." From that point on, when you or IP 92.25.12.37 want to communicate with each other, it will be possible to order slips of paper, ask for missing ones, reply to them, and communicate in a meaningful manner. That way, you can use a single mailbox (or port) and keep all your communications working properly. That's the neat thing about TCP. It adds some overhead, but makes sure that everything is ordered and properly delivered.

If you're not a fan of these analogies, do not despair—we'll cut to the chase by seeing how to use TCP and UDP sockets with Erlang right now.

UDP Sockets

There are only a few basic operations with UDP: setting up a socket, sending messages, receiving messages, and closing a connection. The possibilities are a bit like this:



The first operation, no matter what, is to open a socket. This is done by calling `gen_udp:open/1-2`. The form `{ok, Socket} = gen_udp:open(PortNumber)` is the simplest.

The port number will be any integer between 1 and 65535.

From 0 to 1023, the ports are known as *well-known ports*. Most of the time, your operating system will make it impossible to listen to a well-known port unless you have administrative rights.

Ports from 1024 through 49151 are registered ports. They usually require no permissions and are free to use, although some of them are registered to well-known services (see *http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xml*).

The rest of the ports are known as *dynamic* or *private*. They're frequently used for *ephemeral* ports, which are ports randomly assigned for a single session by the caller connecting to a given service.

For our tests, we'll take port numbers that are somewhat safe, such as 8789, which are unlikely to be taken by other applications.

But before that, what about `gen_udp:open/2`? The second argument can be a list of options, specifying what type of data we want to receive (`list` or `binary`) and how we want to receive it: as messages (`{active, true}`) or as results of a function call (`{active, false}`). There are more options, such as whether the socket should be set with IPv4 (`inet4`) or IPv6 (`inet6`), whether the UDP socket can be used to broadcast information (`{broadcast, true | false}`), the size of buffers, and so on. For now, we'll stick with the simple stuff, and then you can look into the other socket options on your own.

Now let's open a UDP socket. First start a given Erlang shell:

```
1> {ok, Socket} = gen_udp:open(8789, [binary, {active,true}]).
{ok,#Port<0.676>}
2> gen_udp:open(8789, [binary, {active,true}]).
{error,eaddrinuse}
```

In the first command, we open the socket, order it to return binary data, and tell it we want it to be active. You can see a new data structure being returned: `#Port<0.676>`. This is the representation of the socket we have just opened. Sockets can be used a lot like pids. You can even set up links to them so that failure is propagated to the sockets in case of a crash!

The second function call tries to open the same socket over again, which is impossible. That's why `{error, eaddrinuse}` is returned. Fortunately, the first socket is still open.

Next, we'll start a second Erlang shell. In this one, we'll open a second UDP socket, with a different port number:

```
1> {ok, Socket} = gen_udp:open(8790).
{ok,#Port<0.587>}
2> gen_udp:send(Socket, {127,0,0,1}, 8789, "hey there!").
ok
```

Ah, a new function! In the second call, `gen_udp:send/4` is used to send messages (what a wonderfully descriptive name). The arguments are, in order: `gen_udp:send(OwnSocket, RemoteAddress, RemotePort, Message)`. The `RemoteAddress` can be either a string or an atom containing a domain name (`"example.org"`), a 4-tuple describing an IPv4 address, or an 8-tuple describing an IPv6 address. Next we specify the receiver's port number (the mailbox in which we are going to drop our slip of paper), and then the message, which can be a string, a binary, or an IO list.

Did the message ever get sent? Go back to your first shell and try to flush the data:

```
3> flush().
Shell got {udp,#Port<0.676>,{127,0,0,1},8790,<<"hey there!">>}
ok
```

Fantastic. The process that opened the socket will receive messages of the form `{udp, Socket, FromIp, FromPort, Message}`. Using these fields, we'll be able to know where a message is from, what socket it went through, and what the contents are.

We've covered opening sockets, sending data, and receiving data in an active mode. What about passive mode? For this, we need to close the socket from the first shell and open a new one:

```
4> gen_udp:close(Socket).
ok
5> f(Socket).
ok
6> {ok, Socket} = gen_udp:open(8789, [binary, {active,false}]).
{ok,#Port<0.683>}
```

Here, we close the socket, unbind the `Socket` variable, and then bind it as we open a socket again, in passive mode this time. Before sending a message back, try the following:

```
7> gen_udp:recv(Socket, 0).
```

And your shell should be stuck. The function here is `recv/2`. This is the function used to poll a passive socket for messages. The `0` here is the length of the message we want. The funny thing is that the length is completely ignored with `gen_udp`. (`gen_tcp` has a similar function, but in that case, it does have an impact.) Anyway, if we never send a message to the socket, `recv/2` will never return. Go back to the second shell, and send a new message:

```
3> gen_udp:send(Socket, {127,0,0,1}, 8789, "hey there!").
ok
```

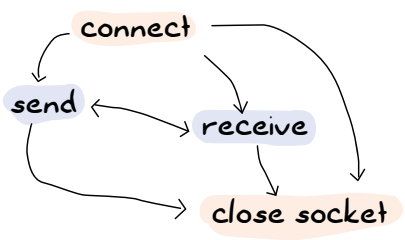
The first shell should have printed `{ok,{{127,0,0,1},8790,<<"hey there!">>}}` as the return value. What if you don't want to wait forever? Just add a timeout value:

```
8> gen_udp:recv(Socket, 0, 2000).
{error,timeout}
```

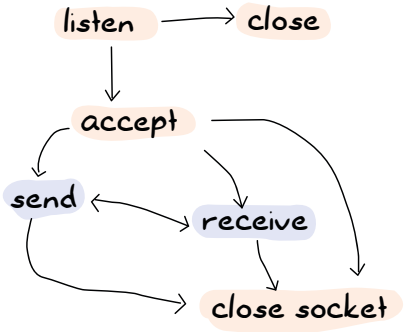
And that's most of it for UDP. No, really!

TCP Sockets

While TCP sockets share a large part of their interface with UDP sockets, there are some vital differences in how they work. The biggest one is that clients and servers are two entirely different things. A client will behave with the following operations:



While a server will follow this scheme:



Weird looking, huh? The client acts a bit like `gen_udp` does. You connect to a port, send and receive, and then stop. When serving, however, we have one new mode there: listening. That's because of how TCP works to set up sessions.

First of all, we open a new shell and start something called a *listen socket* with `gen_tcp:listen(Port, Options)`:

```
1> {ok, ListenSocket} = gen_tcp:listen(8091, [{active,true}, binary]).
{ok,#Port<0.661>}
```

The listen socket is simply in charge of waiting for connection requests. You can see that we used options similar to those we used with `gen_udp`. That's because most options are going to be similar for all IP sockets. The TCP ones do have a few more specific options, such as a connection backlog (`{backlog, N}`), keepalive sockets (`{keepalive, true | false}`), and packet framing (`{packet, N}`, where `N` is the length, in bytes, of each packet's header to be stripped and parsed for you), to name a few.

Once the listen socket is open, any process (and more than one) can take the listen socket and fall into an "accepting" state, locked up until some client asks to talk with it:

```
2> {ok, AcceptSocket} = gen_tcp:accept(ListenSocket, 2000).
** exception error: no match of right hand side value {error,timeout}
3> {ok, AcceptSocket} = gen_tcp:accept(ListenSocket).
** exception error: no match of right hand side value {error,closed}
```

Damn. We timed out and then crashed. The listen socket got closed when the shell process it was associated with disappeared. Let's start over again, this time without the 2 seconds (2000 milliseconds) timeout:

```
4> f().
ok
5> {ok, ListenSocket} = gen_tcp:listen(8091, [{active, true}, binary]).
{ok,#Port<0.728>}
6> {ok, AcceptSocket} = gen_tcp:accept(ListenSocket).
```

And then the process is locked. Great! Let's open a second shell:

```
1> {ok, Socket} = gen_tcp:connect({127,0,0,1}, 8091, [binary, {active,true}]).
{ok,#Port<0.596>}
```

This one takes the same options as usual, and you can add a `Timeout` argument in the last position if you don't want to wait forever. Your first shell should have returned with `{ok, SocketNumber}`. From this point on, the accept socket and the client socket can communicate on a one-on-one basis, similar to using `gen_udp`. Take the second shell and send messages to the first one:

```
3> gen_tcp:send(Socket, "Hey there first shell!").
ok
```

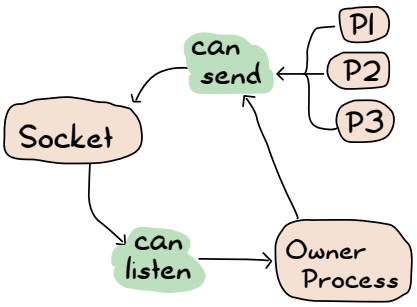
And from the first shell, enter this:

```
7> flush().
Shell got {tcp,#Port<0.729>,<<"Hey there first shell!">>}
ok
```

Both sockets can send messages in the same way, and can then be closed with `gen_tcp:close(Socket)`. Note that closing an accept socket will close that socket alone, and closing a listen socket will close all of the related accept sockets.

That's it for most of TCP sockets in Erlang! But is it really?

Ah yes, of course, there is more that can be done. If you've experimented with sockets a bit on your own, you might have noticed that there is some kind of ownership to sockets. By this, I mean that UDP sockets, TCP client sockets, and TCP accept sockets can all have messages sent through them from any process in existence, but messages received can be read only by the process that started the socket:



That's not very practical, is it? It means that we must always keep the owner process alive to relay messages, even if it has nothing to do with our needs. Wouldn't it be neat to be able to do something like the following?

1. Process A starts a socket
2. Process A sends a request

```
3.  Process A spawns process B
    with a socket
4a. Gives ownership of the      4b. Process B handles the response
    socket to Process B
5a. Process A sends a request   5b. Process B Keeps handling
                                    the response
6a. Process A spawns process C  6b. ...
    with a socket
    ...
```

Here, Process A would be in charge of running a bunch of requests, but each new process would take charge of waiting for the response, processing it, and whatnot. Because of this, it would be clever for Process A to delegate a new process to run the task. The tricky part here is giving away the ownership of the socket.

Here's the trick: Both `gen_tcp` and `gen_udp` contain a function called `controlling_process(Socket, Pid)`. This function must be called by the current socket owner. Then the process tells Erlang, "You know what? Just let this `Pid` guy take over my socket. I give up." From now on, the pid in the function is the one that can read and receive messages from the socket. That's it.

More Control with Inet

So now we have covered how to open sockets, send messages through them, change ownership, and so on. We also know how to listen to messages both in passive and active mode. Back in the UDP example, when we wanted to switch from active to passive mode, we restarted the socket, flushed variables, and went on. This is rather impractical, especially when we desire to make this switch while using TCP, because we would need to break an active session.

Fortunately, there's an Erlang module named `inet` that takes care of handling all operations that can be common to both `gen_tcp` and `gen_udp` sockets. For our problem at hand—changing between active and passive modes—there's a function named `inet:setopts(Socket, Options)`. The option list can contain any terms used at the setup of a socket.

WARNING *Be careful! There is a module named `inet` and a module named `inets`. `inet` is the module we want here. `inets` is an OTP application that contains a bunch of prewritten services and servers (including FTP, TFTP, HTTP, and so on). An easy way to differentiate them is that `inets` is about **s**ervices built on top of `inet`, or if you prefer, `inet` + **s**(ervices).*

Start a shell to be a TCP server:

```
1> {ok, Listen} = gen_tcp:listen(8088, [{active,false}]).
{ok,#Port<0.597>}
2> {ok, Accept} = gen_tcp:accept(Listen).
```

And in a second shell, enter the following:

```
1> {ok, Socket} = gen_tcp:connect({127,0,0,1}, 8088, []).
{ok,#Port<0.596>}
2> gen_tcp:send(Socket, "hey there").
ok
```

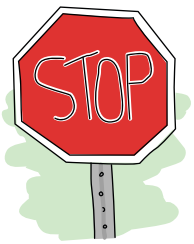
Then back at the first shell, the socket should have been accepted. We flush to see if we got anything:

```
3> flush().
ok
```

Of course not; we're in passive mode. Let's fix this:

```
4> inet:setopts(Accept, [{active, true}]).
ok
5> flush().
Shell got {tcp,#Port<0.598>,"hey there"}
ok
```

Yes! With full control over active and passive sockets, the power is ours. But how do we pick between active and passive modes?



In general, if you're expecting a message right away, passive mode will be much faster. Erlang won't need to toy with your process's mailbox to handle things, and you won't need to scan said mailbox, fetch messages, and so on. Using `recv` will be more efficient, especially if the size of the data to be received is unknown. However, `recv` changes your process from something event-driven to active polling. If you need to play middleman between a socket and some other Erlang code, this might make things a bit complex between blocking and handling incoming messages.

In that case, switching to active mode will be a good idea. If packets are sent as messages, you just need to wait in a `receive` (or a `gen_server`'s `handle_info` function) and play with them, just as with any other messages. The downside of this, apart from speed, has to do with rate limiting.

The idea is that if all data coming from the outside world is blindly accepted by Erlang and then converted to messages, it is somewhat easy for someone outside the VM to flood it and kill it. Passive mode has the advantage of restricting how and when messages can be put into the Erlang VM, and delegating the task of blocking, queuing, and discarding messages to the lower-level implementations.

So what if we need active mode for the semantics, but passive mode for the safety? We could try to quickly switch between passive and active mode with `inet:setopts/2`, but that would be rather risky for race conditions. Instead, there's a mode called *active once*, with the option `{active, once}`. Let's try it to see how it works.

Keep the shell with the server from earlier and enter this:

```
6> inet:setopts(Accept, [{active, once}]).
ok
```

Now go to the client shell and run two more `send/2` calls:

```
3> gen_tcp:send(Socket, "one").
ok
4> gen_tcp:send(Socket, "two").
ok
```

Then go back to the server shell and add the following:

```
7> flush().
Shell got {tcp,#Port<0.598>,"one"}
ok
8> flush().
ok
9> inet:setopts(Accept, [{active, once}]).
ok
10> flush().
Shell got {tcp,#Port<0.598>,"two"}
ok
```

See? Until we ask for `{active, once}` a second time, the packet containing `"two"` hasn't been converted to a message, which means the socket was back to passive mode. So the active once mode allows us to do that back-and-forth switch between active and passive modes in a safe way. This offers nice semantics plus safety.

There are other nice functions that are part of `inet`: stuff to read statistics, get current host information, inspect sockets, and so on. The documentation to the `inet` module contains more details about what can be done.

Well, that's most of what you need to know about sockets. Now let's put all this knowledge into practice.

NOTE *Out in the wilderness of the Internet, there are libraries available to handle a truckload of protocols: HTTP, ZeroMQ, raw Unix sockets, and more. The standard Erlang distribution, however, comes with two main options: TCP and UDP sockets. It also comes with some HTTP servers and parsing code, but that's not the most efficient approach around.*

Sockserv, Revisited

I won't be introducing that much new code in this example. Instead, we'll look back at the `sockserv` server from Process Quest, introduced in Chapter 22, which is a perfectly viable server. We'll look at how to deal with serving TCP connections within an OTP supervision tree, in a `gen_server`.

A naive implementation of a TCP server might look a bit like this:

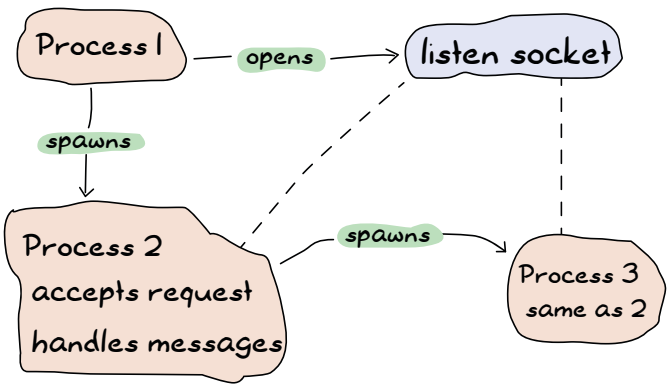
```
-module(naive_tcp).
-compile(export_all).

start_server(Port) ->
    Pid = spawn_link(fun() ->
        {ok, Listen} = gen_tcp:listen(Port, [binary, {active, false}]),
        spawn(fun() -> acceptor(Listen) end),
        timer:sleep(infinity)
    end),
    {ok, Pid}.

acceptor(ListenSocket) ->
    {ok, Socket} = gen_tcp:accept(ListenSocket),
    spawn(fun() -> acceptor(ListenSocket) end),
    handle(Socket).

%% Echoing back whatever was obtained.
handle(Socket) ->
    inet:setopts(Socket, [{active, once}]),
    receive
        {tcp, Socket, <<"quit", _/binary>>} ->
            gen_tcp:close(Socket);
        {tcp, Socket, Msg} ->
            gen_tcp:send(Socket, Msg),
            handle(Socket)
    end.
```

To understand how this works, a little graphical representation might be helpful:



The `start_server` function opens a listen socket, spawns an acceptor, and then just idles forever. The idling is necessary because the listen socket is bound to the process that opened it, so that one needs to remain alive as long as we want to handle connections. Each acceptor process waits for a connection to accept. Once one connection comes in, the acceptor process starts a new, similar process and shares the listen socket with it. Then it can move on and do some processing while the new guy is working. Each handler will repeat all messages it gets until one of them starts with `"quit"`—then the connection is closed.

NOTE *The pattern `<<"quit", _/binary>>` means that we first want to match on a binary string containing the characters `q`, `u`, `i`, and `t`, plus some binary data we don't care about (`_`).*

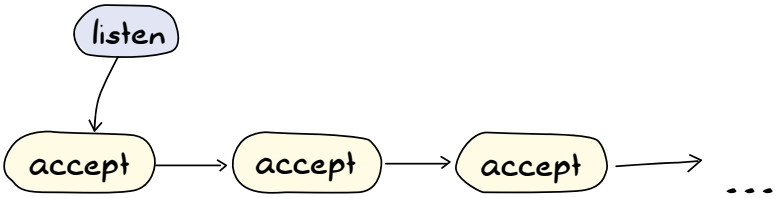
Start the server in an Erlang shell by calling `naive_tcp:start_server(8091)`. Then open a telnet client (remember that telnet clients are technically not for raw TCP, but act as good clients to test servers without needing to write one) to localhost, and you can see the following taking place:

```
$ telnet localhost 8091
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hey there
hey there
that's what I asked
that's what I asked
stop repeating >:(
stop repeating >:(
quit doing that!
Connection closed by foreign host.
```

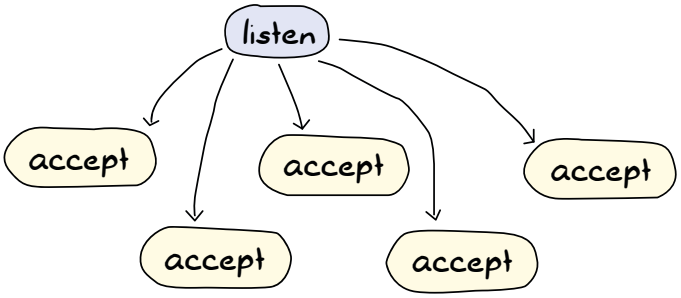
Hooray. Now it's time to start a new company called Poople Inc. and launch a few social networks with our server. However, as the name of the module mentions, this is a naive implementation. The code is simple, and it wasn't conceived with parallelism in mind. If all the requests come one by one, then the naive server works fine. But what happens if we have a queue of 15 people wanting to connect to the server at once?

Then only one query at a time can be replied to, and this involves each process first waiting for the connection, setting it up, and then spawning a new acceptor. The fifteenth request in the queue will have needed to wait for 14 other connections to have been set up to even get the chance of asking for a right to communicate with our server. If you're working with production servers, it might be closer to 500 to 1000 queries per second! That's impractical.

What we need to do is change the sequential workflow we have:



to something more parallel:



By having many acceptors already on standby, we'll be cutting down on a lot of delays to answer new queries.

Now, rather than going through another demo implementation, we'll study `sockserv-1.0.1` from Chapter 22. It will be nicer to explore something based on real OTP components and real-world practice. In fact, the general pattern of `sockserv` is the same one used in servers like `cowboy` (although `cowboy` is no doubt more reliable than `sockserv`) and `etorrent`'s torrent client and server.

To build Process Quest's `sockserv`, we'll go top-down. The scheme we'll need is a supervisor with many workers. If we look at the parallel drawing above, the supervisor should hold the listen socket and share it with all workers, which will be in charge of accepting things.

How do we write a supervisor that can share things across all workers? There is no way to do it with regular supervision; all children are entirely independent, no matter whether you use `one_for_one`, `one_for_all`, or `rest_for_one` supervision. A natural reflex might be to turn to some global state—a registered process that just holds the listen socket and hands it over to the handlers. You must fight this reflex and be clever. Use the force (and the ability to refer back to Chapter 17, which covers supervisors). You have 2 minutes to think of a solution (the 2-minute limit is based on the honor system; time it yourself).

The secret is in using a simple_one_for_one supervisor. Because simple_one_for_one supervisors share the child specification with all of their children, all we need to do is shove the listen socket in there for all the children to access it!

Here's the supervisor in all its glory:

```
%%% The supervisor in charge of all the socket acceptors.
-module(sockserv_sup).
-behavior(supervisor).

-export([start_link/0, start_socket/0]).
-export([init/1]).

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

init([]) ->
    {ok, Port} = application:get_env(port),
    %% Set the socket into {active_once} mode.
    %% See sockserv_serv comments for more details.
    {ok, ListenSocket} = gen_tcp:listen(Port, [{active,once}, {packet,line}]),
    spawn_link(fun empty_listeners/0),
    {ok, {{simple_one_for_one, 60, 3600},
          [{socket,
           {sockserv_serv, start_link, [ListenSocket]}, %Pass the socket!
           temporary, 1000, worker, [sockserv_serv]}
          ]}}.

start_socket() ->
    supervisor:start_child(?MODULE, []).

%% Start with 20 listeners so that many multiple connections can
%% be started at once, without serialization. In best circumstances,
%% a process would keep the count active at all times to insure nothing
%% bad happens over time when processes get killed too much.
empty_listeners() ->
    [start_socket() || _ <- lists:seq(1,20)],
    ok.
```

What is going on in there? The standard start_link/0 and init/1 functions are there. You can see sockserv getting the simple_one_for_one restart strategy, and the child specification having ListenSocket passed around. Every child started with start_socket/0 will have it as an argument by default. Magic!

Just having that won't be enough, though. We want the application to be able to serve queries as soon as possible. That's why we added that call to spawn_link(fun empty_listeners/0). The empty_listeners/0 function will start 20 handlers to be locked and waiting for incoming connections. We've put it inside a spawn_link/1 call for a simple reason: The supervisor process is in its init/1 phase and cannot answer any messages. If we were to call ourselves from within the init function, the process would deadlock and never finish running. An external process is needed just for this reason.

NOTE *In the preceding snippet, notice that we pass the option {packet, line} to gen_tcp. This option will make it so all received packets will be broken into separate lines and queued up based on that (the line ends will still be part of the received strings). This will help make sure we avoid some common errors with telnet clients in our case.*

So yeah, that was the whole tricky part. We can now focus on writing the workers themselves.

If you recall the Process Quest sessions from Chapter 22, interacting with the game followed these steps:

1. The user connects to the server.
2. The server asks for the character's name.
3. The user sends in a character name.
4. The server suggests stats.

5a. The user refuses; go back to point 4.

5b. The user accepts; go to point 6.

6. The game sends events to the player, until . . .
7. The user sends `quit` to the server or the socket is forced to close.

This means we will have two kinds of input to our server processes: input coming from the Process Quest application and input coming from the user. Data coming from the user will be doing so from a socket and so will be handled in our `gen_server`'s `handle_info/2` function. Data coming from Process Quest can be sent in a way we control, and so a cast handled by `handle_cast` will make sense there.

First, we must start the server module:

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

-record(state, {name, % player's name
                next, % next step, used when initializing
                socket}). % the current socket

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

Here we have a pretty standard `gen_server` callback module. The only special thing is the state containing the character's name, the socket, and a field called `next`. The `next` part is a bit of a catchall field to store temporary information related to the state of the server. A `gen_fsm` could also have been used here without too much trouble.

For the actual server startup, we use the following:

```
-define(TIME, 800).
-define(EXP, 50).
```

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

init(Socket) ->
    %% Properly seeding the process.
    <<A:32, B:32, C:32>> = crypto:rand_bytes(12),
    random:seed({A,B,C}),
    %% Because accepting a connection is a blocking function call,
    %% we can not do it in here. Forward to the server loop!
    gen_server:cast(self(), accept),
    {ok, #state{socket=Socket}}.

%% We never need you, handle_call!
handle_call(_E, _From, State) ->
    {noreply, State}.
```

The two macros defined here (`?TIME` and `?EXP`) are special parameters that make it possible to set the baseline delay between actions (800 milliseconds) and the amount of experience required to reach the second level (50, doubled after each level).

You'll notice that the `start_link/1` function takes a socket. That's the listen socket passed in from `sockserv_sup`.

The bit about the random seed is for making sure a process is properly seeded to later generate character statistics. Otherwise, some default value will be used across many processes, and we don't want that. The reason we're initializing in the `init/1` function rather than in whatever library that uses random numbers is that seeds are stored at a process level (damn it—mutable state!), and we wouldn't want to set a new seed on each library call.

The important bit here is that we're casting a message to ourselves. This is because `gen_tcp:accept/1-2` is a blocking operation, combined with the fact that all OTP `init` functions are synchronous. If we wait 30 seconds to accept a connection, the supervisor starting the process will also be locked 30 seconds. So we cast a message to ourselves, and then add the listen socket to the state's socket field.

DON'T DRINK TOO MUCH KOOL-AID

If you read code from other people, you will often see programmers calling `random:seed/1` with the result of `now()`. `now()` is a nice function because it returns monotonic time (always increasing; never twice the same). However, it's a bad seed value for the random algorithm used in Erlang. For this reason, it's better to use `crypto:rand_bytes(12)` to generate 12 crypto-safe random bytes (use `crypto:strong_rand_bytes(12)` if you want even more safety). By doing `<<A:32, B:32, C:32>>`, we're casting the 12 bytes to three integers to be passed in.

Enough fooling around. Now we need to accept that connection:

```
handle_cast(accept, S = #state{socket=ListenSocket}) ->
    {ok, AcceptSocket} = gen_tcp:accept(ListenSocket),
    %% Remember that thou art dust, and to dust thou shalt return.
    %% We want to always keep a given number of children in this app.
    sockserv_sup:start_socket(), % A new acceptor is born, praise the lord.
    send(AcceptSocket, "What's your character's name?", []),
    {noreply, S#state{socket=AcceptSocket, next=name}};
```

We accept the connection, start a replacement acceptor (so that we always have about 20 acceptors ready to handle new connections), and then store the accept socket as a replacement to `ListenSocket` and note that the next message we receive through a socket is about a name with the `next` field.

But before moving on, we send a question to the client through the `send` function, defined as follows:

```
send(Socket, Str, Args) ->
    ok = gen_tcp:send(Socket, io_lib:format(Str++"~n", Args)),
    ok = inet:setopts(Socket, [{active, once}]),
    ok.
```

Trickery! Because we expect to pretty much always need to reply after receiving a message, we do the active once routine within that function, and also add line breaks in there. This is just laziness locked in a function. It isn't necessarily optimal design, given it means that we need to always reply to any message received by the user if we don't want to ever lock them out (by being in `{active, false}` after dropping a message) and that the rate-limiting nature of `{active, once}` is now regulated by how fast we send data, rather than how fast we can digest it. As I said, it's laziness locked in a function.

We've completed steps 1 and 2, and now we need to wait for user input coming from the socket:

```
handle_info({tcp, _Socket, Str}, S = #state{next=name}) ->
    Name = line(Str),
    gen_server:cast(self(), roll_stats),
    {noreply, S#state{name=Name, next=stats}};
```

We have no idea what's going to be in the `Str` string, but that's all right, because the `next` field of the state lets us know whatever we receive is a name. Because we are expecting users to use telnet for the demo application, all bits of text we're going to receive will contain line ends. The `line/1` function, defined as follows, strips them away:

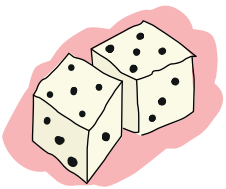
```
%% Let's get rid of the white space and ignore whatever's after.
%% Makes it simpler to deal with telnet.
line(Str) ->
    hd(string:tokens(Str, "\r\n ")).
```

Once we've received that name, we store it and then cast a message to ourselves (`roll_stats`) to generate stats for the player, the next step in line.

NOTE *If you look in the file, you'll see that instead of matching on entire messages, I've used a shorter `?SOCK(Var)` macro. The macro is defined as `-define(SOCK(Msg), {tcp, _Port, Msg})` and is just a quick way for someone as lazy as I am to match on strings with slightly less typing.*

The stats rolling comes back into a `handle_cast` clause:

```
handle_cast(roll_stats, S = #state{socket=Socket}) ->
    Roll = pq_stats:initial_roll(),
    send(Socket,
         "Stats for your character:~n"
         "  Charisma: ~B~n"
         "  Constitution: ~B~n"
         "  Dexterity: ~B~n"
         "  Intelligence: ~B~n"
         "  Strength: ~B~n"
         "  Wisdom: ~B~n~n"
         "Do you agree to these? y/n~n",
         [Points || {_Name, Points} <- lists:sort(Roll)]),
    {noreply, S#state{next={stats, Roll}}};
```



The `pq_stats` module contains functions to roll stats, and the whole clause is being used only to output the stats there. The `~B` format parameters mean we want an integer to be printed out. The `next` part of the state is a bit overloaded here. Because we ask the users whether they agree or not, we will need to wait for them to respond, and either drop the stats and generate new ones or pass them to the Process Quest character we'll no doubt start very soon.

Let's listen to the user input, this time in the `handle_info` function:

```
handle_info({tcp, Socket, Str}, S = #state{socket=Socket, next={stats, _}}) ->
    case line(Str) of
        "y" ->
            gen_server:cast(self(), stats_accepted);
        "n" ->
            gen_server:cast(self(), roll_stats);
        _ -> % Ask again because we didn't get what we wanted.
            send(Socket, "Answer with y (yes) or n (no)", [])
    end,
    {noreply, S};
```

It would have been tempting to start the character in this direct function clause, but I decided against it. `handle_info` is to handle user input, and `handle_cast` is for Process Quest things. Separation of concerns! If the user

denies the stats, we just call `roll_stats` again. This is nothing new. When the user accepts, we can start the Process Quest character and begin waiting for events from there:

```
%% The player has accepted the stats! Start the game!
handle_cast(stats_accepted, S = #state{name=Name, next={stats, Stats}}) ->
    processquest:start_player(Name, [{stats,Stats},{time,?TIME},
                                     {lvlexp, ?EXP}]),
    processquest:subscribe(Name, sockserv_pq_events, self()),
    {noreply, S#state{next=playing}};
```

These are regular calls defined for the game. You start a player and subscribe to the events with the `sockserv_pq_events` event handler. The next state is `playing`, which means that all messages received are more than likely to be from the game:

```
%% Events coming in from process quest.
%% We know this because all these events' tuples start with the
%% name of the player as part of the internal protocol defined for us.
handle_cast(Event, S = #state{name=N, socket=Sock}) when element(1, Event) =:= N ->
    [case E of
        {wait, Time} -> timer:sleep(Time);
        IoList -> send(Sock, IoList, [])
     end || E <- sockserv_trans:to_str(Event)], % Translate to a string.
    {noreply, S}.
```

Here, `sockserv_trans:to_str(Event)` converts some game event to lists of IO lists or `{wait, Time}` tuples that represent delays to wait between parts of events (we print "executing a . . ." messages a bit before showing what the item dropped by the enemy is).

In our list of steps to follow, we've covered all except one: quitting when users tell us they want to. Put the following clause as the top one in `handle_info`:

```
handle_info({tcp, _Socket, "quit"++_}, S) ->
    processquest:stop_player(S#state.name),
    gen_tcp:close(S#state.socket),
    {stop, normal, S};
```

Stop the character, close the socket, and terminate the process.

Other reasons to quit include the TCP socket being closed by the client:

```
handle_info({tcp_closed, _Socket, _}, S) ->
    {stop, normal, S};
handle_info({tcp_error, _Socket, _}, S) ->
    {stop, normal, S};
handle_info(E, S) ->
    io:format("unexpected: ~p~n", [E]),
    {noreply, S}.
```

You could also check for similar special cases when calling `gen_tcp:send/3` (it wouldn't return `ok`) or `inet:setopts/2`, although by virtue of being active most of the time, we'll get the message shown here anyway, although possibly later.

We also added an extra clause to handle unknown messages. If the user types in something we don't expect, we don't want to crash.

Only the `terminate/2` and `code_change/3` functions are left to write:

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

terminate(normal, _State) ->
    ok;
terminate(_Reason, _State) ->
    io:format("terminate reason: ~p~n", [_Reason]).
```

If you followed through this whole example, you can try compiling this file and substituting it for the corresponding BEAM file in the release we had and see if it runs correctly. It should, if you copied things right (and if I did, too).

Where to Go from Here?

Your next assignment, should you choose to accept it, is to add a few more commands of your choice to the client. Why not add things like "pause" that will queue up actions for a while and then output them all once you resume the server? Or if you're badass enough, try noting the levels and stats you have so far in the `sockserv_serv` module and adding commands to fetch them from the client side. I've always hated exercises left to the reader, but sometimes it's just too tempting to drop one here and there, so enjoy!

Reading the source of existing server implementations and programming some yourself are also good exercises. Rare are the languages where doing things like writing a basic web server is an exercise for amateurs, but Erlang is one of them. Practice a bit, and it will become second nature. Erlang communicating with the outside world is just one of the many steps we've taken toward writing useful software. Chapter 24 will give us some tools to make sure that useful software remains useful over time, with the intervention of unit tests.

24

EUNITED NATIONS COUNCIL

The software we've written has gotten progressively bigger and somewhat more complex. When that happens, it becomes rather tedious to start an Erlang shell, type things in, look at results, and make sure things work after code has been changed. As time passes, it becomes simpler for everyone to run tests that are prepared and ready in advance, rather than following lists of stuff to check by hand all the time. It's also possible that you're a fan of test-driven development and so will find tests useful.

When we created an RPN calculator in Chapter 8, we wrote a few tests manually. They were simply a set of pattern matches of the form `Result = Expression` that would crash if something went wrong and succeed

otherwise. That works for simple bits of code you write for yourself, but when you get to more serious tests, you will definitely want something better, like a framework.

For unit tests, we'll tend to stick with EUnit. For integration tests, EUnit and Common Test can both do the job. In fact, Common Test can do everything from unit tests up to system tests, and even testing of external software not written in Erlang. For now, we'll go with EUnit, given how simple it is and the good results it yields. We'll look into using Common Test in Chapter 28.

EUnit—What's an EUnit?

EUnit, in its simplest form, is just a way to automate running functions that end in `_test()` in a module by assuming they are unit tests. If you dig out that RPN calculator we wrote in Chapter 8, you'll find the following code:

```
rpn_test() ->
    5 = rpn("2 3 +"),
    87 = rpn("90 3 -"),
    -4 = rpn("10 4 3 + 2 * -"),
    -2.0 = rpn("10 4 3 + 2 * - 2 /"),
    ok = try
        rpn("90 34 12 33 55 66 + * - +")
    catch
        error:{badmatch,[_|_]} -> ok
    end,
    4037 = rpn("90 34 12 33 55 66 + * - + -"),
    8.0 = rpn("2 3 ^"),
    true = math:sqrt(2) == rpn("2 0.5 ^"),
    true = math:log(2.7) == rpn("2.7 ln"),
    true = math:log10(2.7) == rpn("2.7 log10"),
    50 = rpn("10 10 10 20 sum"),
    10.0 = rpn("10 10 10 20 sum 5 /"),
    1000.0 = rpn("10 10 20 0.5 prod"),
    ok.
```

This is the test function we wrote to make sure the calculator worked. Find the old module and try this:

```
1> c(calc).
{ok,calc}
2> eunit:test(calc).
  Test passed.
ok
```

Calling `eunit:test(Module)` was all we needed! Yay, we now know EUnit! Pop the champagne and let's head to a different chapter!

Obviously, a testing framework that does this little wouldn't be very useful, and in technical programmer jargon, it might be described as "not very good."

EUnit does more than automatically export and run functions ending in _test(). For one, you can move the tests to a different module so that your code and its tests are not mixed together. This means you can't test private functions anymore, but also that if you develop all your tests against the module's interface (the exported functions), you won't need to rewrite tests when you refactor your code.

Let's try separating tests and code with two simple modules:

```
-module(ops).
-export([add/2]).

add(A,B) -> A + B.
```

```
-module(ops_tests).
-include_lib("eunit/include/eunit.hrl").

add_test() ->
    4 = ops:add(2,2).
```

So we have ops and ops_tests, where the second includes tests related to the first. Here's something EUnit can do:

```
3> c(ops).
{ok,ops}
4> c(ops_tests).
{ok,ops_tests}
5> eunit:test(ops).
  Test passed.
ok
```

Calling eunit:test(Mod) automatically looks for *Mod*_tests and runs the tests within that module.

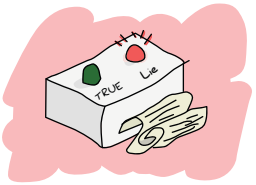
Let's change the test a bit (make it 3 = ops:add(2,2)) to see what failures look like:

```
6> c(ops_tests).
{ok,ops_tests}
7> eunit:test(ops).
ops_tests: add_test (module 'ops_tests')...*failed*
::error:{badmatch,4}
  in function ops_tests:add_test/0


=======================================================
  Failed: 1.  Skipped: 0.  Passed: 0.
error
```

We can see which test failed (ops_tests: add_test...) and why it failed (::error:{badmatch,4}). We also get a full report on how many tests passed or failed.

The output is pretty bad though—at least as bad as regular Erlang crashes. It doesn't have a clear explanation (`4 didn't match with what, exactly?`), for example. We're left helpless by a test framework that runs tests but doesn't tell you much about them.



For this reason, EUnit introduces a few macros to help us. Each of them will give us cleaner reporting (including line numbers) and clearer semantics. These macros are the difference between knowing that something goes wrong and knowing *why* something goes wrong:

`?assert(Expression)` and `?assertNot(Expression)`

These will test for Boolean values. If any value other than `true` makes it into `?assert`, an error will be shown. The same is true for `?assertNot`, but for negative values. These macros are somewhat equivalent to `true = X` and `false = Y`.

`?assertEqual(A, B)`

This does a strict comparison (equivalent to `=:=`) between two expressions, `A` and `B`. If they are different, a failure will occur. This is roughly equivalent to `true = X =:= Y`. The macro `?assertNotEqual` is available to do the opposite of `?assertEqual`.

`?assertMatch(Pattern, Expression)`

This allows you to match in a form similar to *`Pattern = Expression`*, without variables ever binding. This means that you could do something like `?assertMatch({X,X}, some_function())` and assert that you receive a tuple with two elements being identical. Moreover, you could later run `?assertMatch(X,Y)`, and `X` would not be bound.

Actually, rather than behaving like *`Pattern = Expression`*, this macro's semantics are closer to `(fun (Pattern) -> true; (_) -> erlang:error(nomatch) end)(Expression)`. Variables in the pattern's head *never* get bound across multiple assertions. The macro `?assertNotMatch` was added to EUnit in R14B04.

`?assertError(Pattern, Expression)`

This tells EUnit that *`Expression`* should result in an error. As an example, `?assertError(badarith, 1/0)` would be a successful test.

`?assertThrow(Pattern, Expression)`

This is the same as `?assertError`, but with `throw(Pattern)` instead of `erlang:error(Pattern)`.

`?assertExit(Pattern, Expression)`

This is the same as `?assertError`, but with `exit(Pattern)` (and not `exit/2`) instead of `erlang:error(Pattern)`.

?assertException(Class, Pattern, Expression)

This is a general form of the three previous macros. As an example, `?assertException(error, Pattern, Expression)` is the same as `?assertError(Pattern, Expression)`. Starting with R14B04, there is also the macro `?assertNotException/3` available for tests.

Using these macros, we could write better tests in our module, as follows:

```
-module(ops_tests).
-include_lib("eunit/include/eunit.hrl").

add_test() ->
    4 = ops:add(2,2).

new_add_test() ->
    ?assertEqual(4, ops:add(2,2)),
    ?assertEqual(3, ops:add(1,2)),
    ?assert(is_number(ops:add(1,2))),
    ?assertEqual(3, ops:add(1,1)),
    ?assertError(badarith, 1/0).
```

Let's try running them:

```
8> c(ops_tests).
./ops_tests.erl:12: Warning: this expression will fail with a 'badarith' exception
{ok,ops_tests}
9> eunit:test(ops).
ops_tests: new_add_test...*failed*
::error:{assertEqual_failed,[{module,ops_tests},
                             {line,11},
                             {expression,"ops : add ( 1 , 1 )"},
                             {expected,3},
                             {value,2}]}
  in function ops_tests:'-new_add_test/0-fun-3-'/1
  in call from ops_tests:new_add_test/0


=======================================================
  Failed: 1.  Skipped: 0.  Passed: 1.
error
```

See how much nicer the error reporting is? We know that the `?assertEqual` on line 11 of `ops_tests` failed. When we called `ops:add(1,1)`, we thought we would receive 3 as a value, but we instead got 2. Of course you must read these values as Erlang terms, but at least they're there.

What's annoying with this, however, is that even though we had five assertions and only one failed, the whole test was still considered a failure. It would be nicer to know that some assertion failed without behaving as if all the others after it failed, too. Our test is equivalent to taking an exam,

and as soon as you make a mistake, you fail and get thrown out of school. Then your dog dies, and you just have a horrible day.

Because of this common need for flexibility, EUnit supports something called *test generators*.

Test Generators

Test generators are pretty much shorthand for assertions wrapped in functions that can be run later, in clever manners. Instead of having functions ending with `_test()` with macros that are of the form `?assertSomething`, we use functions that end in `_test_()` and macros of the form `?_assertSomething`. Those are very small changes, but they make tests much more powerful.

The following two tests would be equivalent:

```
function_test() -> ?assert(A == B).
function_test_() -> ?_assert(A == B).
```

Here, `function_test_()` is called a *test generator function*, while `?_assert(A == B)` is called a *test generator*. It's called that because, secretly, the underlying implementation of `?_assert(A == B)` is `fun() -> ?assert(A,B) end`; that is to say, it's a function that generates a test.

The advantage of test generators, compared to regular assertions, is that they are `funs`. This means that they can be manipulated without being executed. We could, in fact, have *test sets* of the following form:

```
my_test_() ->
    [?_assert(A),
     [?_assert(B),
      ?_assert(C),
      [?_assert(D)]],
     [[?_assert(E)]]].
```

Test sets can be deeply nested lists of test generators. We could have functions that return tests! Let's add the following to `ops_tests`:

```
add_test_() ->
    [test_them_types(),
     test_them_values(),
     ?_assertError(badarith, 1/0)].

test_them_types() ->
    ?_assert(is_number(ops:add(1,2))).

test_them_values() ->
    [?_assertEqual(4, ops:add(2,2)),
     ?_assertEqual(3, ops:add(1,2)),
     ?_assertEqual(3, ops:add(1,1))].
```

Because only add_test_() ends in _test_(), the two functions test_them_types() and test_them_values() will not be seen as tests. In fact, they will be called by add_test_() to generate tests:

```
1> c(ops_tests).
./ops_tests.erl:12: Warning: this expression will fail with a 'badarith' exception
./ops_tests.erl:17: Warning: this expression will fail with a 'badarith' exception
{ok,ops_tests}
2> eunit:test(ops).
ops_tests:25: test_them_values...*failed*
[...]
ops_tests: new_add_test...*failed*
[...]

=======================================================
  Failed: 2.  Skipped: 0.  Passed: 5.
error
```

So we still get the expected failures, and now you see that we jumped from two tests to seven—the magic of test generators.

What if we want to test only some parts of the suite—maybe just add_test_/0? Well, EUnit has a few tricks up its sleeve.

```
3> eunit:test({generator, fun ops_tests:add_test_/0}).
ops_tests:25: test_them_values...*failed*
::error:{assertEqual_failed,[{module,ops_tests},
                             {line,25},
                             {expression,"ops : add ( 1 , 1 )"},
                             {expected,3},
                             {value,2}]}
  in function ops_tests:'-test_them_values/0-fun-4-'/1

=======================================================
  Failed: 1.  Skipped: 0.  Passed: 4.
error
```

Note that this works only with test generator functions. What we have here as {generator, *Fun*} is what EUnit parlance calls a *test representation*.

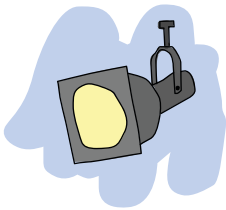
EUnit provides the following test representations:

- {module, *Mod*} runs all tests in *Mod*.
- {dir, *Path*} runs all the tests for the modules found in *Path*.
- {file, *Path*} runs all the tests found in a single compiled module.
- {generator, *Fun*} runs a single generator function as a test, as seen in the preceding example.
- {application, *AppName*} runs all the tests for all the modules mentioned in *AppName*'s *.app* file.

These different test representations can make it easy to run test suites for entire applications or even releases.

Fixtures

It would still be pretty hard to test entire applications just by using assertions and test generators. This is why *fixtures* were added. Fixtures, while not being a catchall solution to getting your tests up and running to the application level, allow you to build a certain scaffolding around tests.



The scaffolding in question is a general structure that allows you to define setup and teardown functions for each of the tests. These functions will allow you to build the state and environment required for each of the tests to be useful. Moreover, the scaffolding will let you specify how to run the tests (do you want to run them locally, in separate processes, or some other way?).

Several types of fixtures are available, with variations to them. The first type is simply called the setup fixture. A setup fixture takes one of the following forms:

```
{setup, Setup, Instantiator}
{setup, Setup, Cleanup, Instantiator}
{setup, Where, Setup, Instantiator}
{setup, Where, Setup, Cleanup, Instantiator}
```

Argh! It appears we need a bit of EUnit vocabulary in order to understand how fixtures work.

Setup
: A function that does not take any arguments. Each of the tests will be passed the value returned by the setup function, which will be called once per instantiator.

Cleanup
: A function that takes the result of a setup function as an argument and takes care of cleaning up whatever is needed. If an OTP `terminate` does the opposite of `init`, cleanup functions are the opposite of setup functions for EUnit. For each setup function call, the cleanup function will be called.

Instantiator
: A function that takes the result of a setup function and returns a test set (remember that test sets may be deeply nested lists of `?_Macro` assertions). You can also return a list of instantiators, and the setup and cleanup function will be called for each one.

Where
: A function that specifies how to run the tests, as in `local`, `spawn`, `{spawn, node()}`.

All right, so what does this look like in practice? Well, let's imagine some test to make sure that a fictive process registry correctly handles trying to register the same process twice, with different names:

```
double_register_test_() ->
    {setup,
     fun start/0,               % setup function
     fun stop/1,                % cleanup function
     fun two_names_one_pid/1}.  % instantiator

start() ->
    {ok, Pid} = registry:start_link(),
    Pid.

stop(Pid) ->
    registry:stop(Pid).

two_names_one_pid(Pid) ->
    ok = registry:register(Pid, quite_a_unique_name, self()),
    Res = registry:register(Pid, my_other_name_is_more_creative, self()),
    [?_assertEqual({error, already_named}, Res)].
```

This fixture first starts the `registry` server within the `start/0` function. Then the instantiator `two_names_one_pid(ResultFromSetup)` is called. In that test, the only thing we do is try to register the current process twice.

That's where the instantiator does its work. The result of the second registration is stored in the variable `Res`. The function will then return a test set containing a single test (`?_assertEqual({error, already_named}, Res)`). That test set will be run by EUnit. Then the cleanup function `stop/1` will be called. Using the pid returned by the setup function, `stop/1` will be able to shut down the registry that we started beforehand.

What's even better is that this whole fixture itself can be put inside a test set:

```
some_test_() ->
    [{setup, fun start/0, fun stop/1, fun some_instantiator1/1},
     {setup, fun start/0, fun stop/1, fun some_instantiator2/1},
     ...
     {setup, fun start/0, fun stop/1, fun some_instantiatorN/1}].
```

And this will work!

What's annoying here is the need to always repeat that setup and teardown functions, especially when they're always the same. That's where the second type of fixture, the foreach fixture, enters the stage.

The foreach fixture is quite similar to the setup fixture, with the difference that it takes lists of instantiators.

```
{foreach, Where, Setup, Cleanup, [Instantiator]}
{foreach, Setup, Cleanup, [Instantiator]}
```

```
{foreach, Where, Setup, [Instantiator]}
{foreach, Setup, [Instantiator]}
```

Here's the `some_test_/0` function written with a foreach fixture:

```
some_test2_() ->
    {foreach
     fun start/0,
     fun stop/1,
     [fun some_instantiator1/1,
      fun some_instantiator2/1,
      ...
      fun some_instantiatorN/1]}.
```

That's better. The foreach fixture will then take each of the instantiators and run the setup and teardown function for them.

Now we have covered how to use a fixture for one instantiator and a fixture for many instantiators (each getting their setup and teardown function calls). What if we want to have one setup function call and one teardown function call for many instantiators?

In other words, what if we have many instantiators, but want to set some state only once? There's no easy way for this, but here's a little trick that might do it:

```
some_tricky_test_() ->
    {setup,
     fun start/0,
     fun stop/1,
     fun (SetupData) ->
        [some_instantiator1(SetupData),
         some_instantiator2(SetupData),
         ...
         some_instantiatorN(SetupData)]
     end}.
```

By using the fact that test sets can be deeply nested lists, we wrap a bunch of instantiators with an anonymous function behaving like an instantiator for them.

More Test Control

Tests can also have some finer-grained control as to how they should be running when you use fixtures. Four options are available:

`{spawn, TestSet}`
: This runs tests in a separate process than the main test process. The test process will wait for all of the spawned tests to finish.

{timeout, *Seconds*, TestSet}
: The tests will run for *Seconds* number of seconds. If they take longer than *Seconds* to finish, they will be terminated without further ado.

{inorder, TestSet}
: This tells EUnit to run the tests within the test set strictly in the order they are returned.

{inparallel, Tests}
: Where possible, the tests will be run in parallel.

As an example, the `some_tricky_test_/0` test generator could be rewritten as follows:

```
some_tricky_test2_() ->
    {setup,
     fun start/0,
     fun stop/1,
     fun(SetupData) ->
        {inparallel,
         [some_instantiator1(SetupData),
          some_instantiator2(SetupData),
          ...
          some_instantiatorN(SetupData)]}
     end}.
```

Test Documentation

That's really most of it for fixtures, but there's one more nice trick to show you. You can add descriptions of tests in a neat way. Check this out:

```
double_register_test_() ->
    {"Verifies that the registry doesn't allow a single process to "
     "be registered under two names. We assume that each pid has the "
     "exclusive right to only one name",
     {setup,
      fun start/0,
      fun stop/1,
      fun two_names_one_pid/1}}.
```

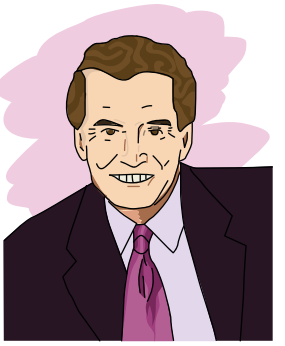
Nice, huh? You can wrap a fixture by doing `{Comment, Fixture}` in order to get readable tests. Let's put this in practice.

Testing Regis

Because seeing fake tests isn't the most entertaining thing to do, and because pretending to test software that doesn't exist is even worse, we'll instead study the tests I have written for the *regis-1.0.0* process registry, the one used by Process Quest.

The development of `regis` was done in a test-driven manner. Hopefully you don't hate test-driven development (TDD), but even if you do, it shouldn't be too bad because we'll look at the test suite after the fact. By doing this, we cut through the few trial-and-error sequences and backpedaling that I might have experienced when writing it the first time, and I'll look like I'm really competent, thanks to the magic of text editing.

The `regis` application is made of three processes: a supervisor, a main server, and an application callback module. Knowing that the supervisor will check only the server, and that the application callback module will do nothing except behave as an interface for the two other modules, we can safely write a test suite focusing on the server itself, without any external dependencies.



Being a good TDD fan, I began by writing a list of all the features I wanted to cover:

- Implement an interface similar to the Erlang default process registry.
- The server will have a registered name so that it can be contacted without tracking its pid.
- A process can be registered through our service and can then be contacted by its name.
- A list of all registered processes can be obtained.
- A name that is not registered by any process should return the atom `undefined` (much like the regular Erlang registry) in order to crash calls using it.
- A process cannot have two names.
- Two processes cannot share the same name.
- A process that was registered can be registered again if it was unregistered between calls.
- Unregistering a process never causes a process crash.
- A registered process's exit will unregister its name.

That's a respectable list. Doing the elements one by one and adding cases as I went, I transformed each of the specifications into a test.

The final file obtained was *regis_server_tests*. I wrote things using a basic structure a bit like this:

```
-module(regis_server_tests).
-include_lib("eunit/include/eunit.hrl").

%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% TESTS DESCRIPTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%
%%% SETUP FUNCTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%
%%% ACTUAL TESTS %%%
%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%
%%% HELPER FUNCTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%
```

Yes, this does look weird when the module is empty, but as you fill it up, it makes more and more sense.

After adding a first test—that it should be possible to start a server and access it by name—the file looked like this:

```
-module(regis_server_tests).
-include_lib("eunit/include/eunit.hrl").

%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% TESTS DESCRIPTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%
start_stop_test_() ->
    {"The server can be started, stopped and has a registered name",
     {setup,
      fun start/0,
      fun stop/1,
      fun is_registered/1}}.

%%%%%%%%%%%%%%%%%%%%%%%
%%% SETUP FUNCTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%
start() ->
    {ok, Pid} = regis_server:start_link(),
    Pid.

stop(_) ->
    regis_server:stop().
```

```
%%%%%%%%%%%%%%%%%%%
%%% ACTUAL TESTS %%%
%%%%%%%%%%%%%%%%%%%
is_registered(Pid) ->
    [?_assert(erlang:is_process_alive(Pid)),
     ?_assertEqual(Pid, whereis(regis_server))].

%%%%%%%%%%%%%%%%%%%%%%%
%%% HELPER FUNCTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%
```

See the organization now? It's already so much better. The top part of the file contains only fixtures and top-level descriptions of features. The second part contains setup and cleanup functions that we might need. The last part contains the instantiators returning test sets.

In this case, the instantiator is `is_registered(Pid)`, which will make sure the server can be started and stopped. We'll actually revisit it in a short while.

In the final file for the tests, if you've downloaded it with the rest of the code in this book, the first two sections would look more like this:

```
-module(regis_server_tests).
-include_lib("eunit/include/eunit.hrl").

-define(setup(F), {setup, fun start/0, fun stop/1, F}).

%%%%%%%%%%%%%%%%%%%%%%%%
%%% TESTS DESCRIPTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%%

start_stop_test_() ->
    {"The server can be started, stopped and has a registered name",
     ?setup(fun is_registered/1)}.

register_test_() ->
    [{"A process can be registered and contacted",
      ?setup(fun register_contact/1)},
     {"A list of registered processes can be obtained",
      ?setup(fun registered_list/1)},
     {"An undefined name should return 'undefined' to crash calls",
      ?setup(fun noregister/1)},
     {"A process cannot have two names",
      ?setup(fun two_names_one_pid/1)},
     {"Two processes cannot share the same name",
      ?setup(fun two_pids_one_name/1)}].

unregister_test_() ->
    [{"A process that was registered can be registered again iff it was "
      "unregistered between both calls",
      ?setup(fun re_un_register/1)},
     {"Unregistering never crashes",
      ?setup(fun unregister_nocrash/1)},
     {"A crash unregisters a process",
      ?setup(fun crash_unregisters/1)}].
```

```
%%%%%%%%%%%%%%%%%%%%%%
%%% SETUP FUNCTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%
start() ->
    {ok, Pid} = regis_server:start_link(),
    Pid.

stop(_) ->
    regis_server:stop().

%%%%%%%%%%%%%%%%%%%%%%
%%% HELPER FUNCTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%
%% nothing here yet
```

Nice, isn't it? Note that as I was writing the suite, I ended up seeing that I never needed any other setup and teardown functions than `start/0` and `stop/1`. For this reason, I added the `?setup(Instantiator)` macro, which makes things look a bit better than if all the fixtures were fully expanded.

It's now pretty obvious that I turned each point of the feature list into a bunch of tests. You'll note that I divided all tests depending on whether they had to do with starting and stopping the server (`start_stop_test_/0`), registering processes (`register_test_/0`), and unregistering processes (`unregister_test_/0`).

By reading the test generators' definitions, we can know what the module is supposed to be doing. The tests become documentation (although they should not replace proper documentation).

We'll study the tests a bit and see why things were done in a certain way. Here's the first test in the list, `start_stop_test_/0`, with the simple requirement that the server can be registered:

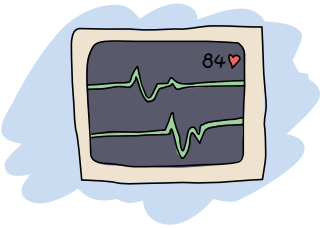
```
start_stop_test_() ->
    {"The server can be started, stopped and has a registered name",
     ?setup(fun is_registered/1)}.
```

The implementation of the test itself is put in the `is_registered/1` function, which hasn't changed:

```
%%%%%%%%%%%%%%%%%%%%
%%% ACTUAL TESTS %%%
%%%%%%%%%%%%%%%%%%%%
is_registered(Pid) ->
    [?_assert(erlang:is_process_alive(Pid)),
     ?_assertEqual(Pid, whereis(regis_server))].
```

This is the same instantiator as earlier. There's nothing really special about the test, although the function `erlang:is_process_alive(Pid)` might be new to you. As its name says, the function checks whether a process is currently running. I've put that test in there for the simple reason that it might be possible that the server crashes as soon as we start it, or that it's never

started in the first place. This would give confusing test results if we assumed the registration failed due to some code error, when the server wasn't started in the first place. We don't want that.



The second test is related to being able to register a process:

```
{"A process can be registered and contacted",
 ?setup(fun register_contact/1)}
```

Here's what the test looks like:

```
register_contact(_) ->
    Pid = spawn_link(fun() -> callback(regcontact) end),
    timer:sleep(15),
    Ref = make_ref(),
    WherePid = regis_server:whereis(regcontact),
    regis_server:whereis(regcontact) ! {self(), Ref, hi},
    Rec = receive
        {Ref, hi} -> true
        after 2000 -> false
    end,
    [?_assertEqual(Pid, WherePid),
     ?_assert(Rec)].
```

Granted, this isn't the most elegant test around. The timers are the biggest eyesore, and they could be avoided by using some sort of synchronous process initialization (either a behavior or functions such as `proc_lib:start_link` and its related synchronization functions, which are described in the `proc_lib` module's documentation). The test spawns a process that will do nothing but register itself and reply to some message we send it. This is all done in the `callback/1` helper function, which is defined as follows:

```
%%%%%%%%%%%%%%%%%%%%%%%
%%% HELPER FUNCTIONS %%%
%%%%%%%%%%%%%%%%%%%%%%%
callback(Name) ->
    ok = regis_server:register(Name, self()),
    receive
        {From, Ref, Msg} -> From ! {Ref, Msg}
    end.
```

So the function has the module register itself, receives a message, and sends a response back. Once the process is started, the `register_contact/1` instantiator waits 15 milliseconds (just a tiny delay to make sure the other process registers itself), and then tries to use the `whereis` function from `regis_server` to retrieve a pid and send a message to the process. If the `regis` server is functioning correctly, a message will be returned, and the pids will match in the tests at the bottom of the function.

DON'T DRINK TOO MUCH KOOL-AID

By reading the second test, you have seen the little timer work we've needed to do. Because of the concurrent and time-sensitive nature of Erlang programs, tests will frequently be filled with tiny timers like this that have the sole role of trying to synchronize bits of code.

The problem then becomes to try to define what should be considered a good timer, with a delay that is long enough. With a system running many tests, or even a computer under a heavy load, will the timers still be waiting for long enough?

Erlang programmers who write tests sometimes must be clever in order to minimize how much synchronization they need to get things to work. There is no easy solution.

The next tests are introduced as follows:

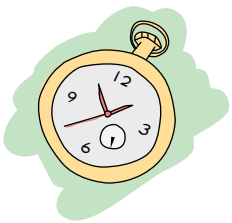
```
{"A list of registered processes can be obtained",
 ?setup(fun registered_list/1)}
```

So when a bunch of processes have been registered, it should be possible to get a list of all the names. This is a functionality similar to Erlang's `registered()` function call:

```
registered_list(_) ->
    L1 = regis_server:get_names(),
    Pids = [spawn(fun() -> callback(N) end) || N <- lists:seq(1,15)],
    timer:sleep(200),
    L2 = regis_server:get_names(),
    [exit(Pid, kill) || Pid <- Pids],
    [?_assertEqual([], L1),
     ?_assertEqual(lists:sort(lists:seq(1,15)), lists:sort(L2))].
```

First, we make sure that the first list of registered processes is empty (`?_assertEqual(L1, [])`) so that we have something that works even when no process has ever tried to register itself. Then 15 processes are created, all of which will try to register themselves with a number (1 through 15). We make the test sleep a bit to make sure all processes have the time to register themselves, and then call `regis_server:get_names()`. The names should include all integers between 1 and 15, inclusively. Then a slight cleanup is done by eliminating all the registered processes—we don't want to be leaking them, after all.

You'll notice the tendency of the tests to store state in variables (`L1` and `L2`) before using them in test sets. The reason for this is that the test set that is returned is executed well after the test initiator (the whole active bit of code) has been running. If you were to try to put function calls that depend on



other processes and time-sensitive events in the ?_assert* macros, you would get everything out of sync, and things would generally be awful for you and the people using your software.

The next test is simple:

```
{"An undefined name should return 'undefined' to crash calls",
 ?setup(fun noregister/1)}
...
noregister(_) ->
    [?_assertError(badarg, regis_server:whereis(make_ref()) ! hi),
     ?_assertEqual(undefined, regis_server:whereis(make_ref()))].
```

As you can see, this tests for two things: we return undefined and the specification's assumption that using undefined does indeed crash attempted calls. Here, there is no need to use temporary variables to store the state; both tests can be executed at any time during the life of the regis server, given we never change its state.

Let's keep going:

```
{"A process cannot have two names",
 ?setup(fun two_names_one_pid/1)},
...
two_names_one_pid(_) ->
    ok = regis_server:register(make_ref(), self()),
    Res = regis_server:register(make_ref(), self()),
    [?_assertEqual({error, already_named}, Res)].
```

This is pretty much the same test we used in a demo earlier in the chapter. In this one, we're just looking to see whether we get the right output and that the test process can't register itself twice with different names.

The next test is the opposite of two_names_one_pid:

```
{"Two processes cannot share the same name",
 ?setup(fun two_pids_one_name/1)}].
...
two_pids_one_name(_) ->
    Pid = spawn(fun() -> callback(myname) end),
    timer:sleep(15),
    Res = regis_server:register(myname, self()),
    exit(Pid, kill),
    [?_assertEqual({error, name_taken}, Res)].
```

Here, because we need two processes and the results of only one of them, the trick is to spawn one process (the one whose results we do not need), and then do the critical part ourselves.

You can see that timers are used to make sure that the other process tries registering a name first (within the callback/1 function), and that the test process itself waits to take its turn, expecting an error tuple ({error, name_taken}) as a result.

USING UNIQUE VALUES

You might have noticed that the preceding tests tend to use `make_ref()` a lot. When possible, it is useful to use functions that generate unique values, as `make_ref()` does. If at some point in the future someone wants to run tests in parallel or to run them under a single `regis` server that never stops, then it will be possible to do so without needing to modify the tests.

If we were to use hardcoded names like `a`, `b`, and `c` in all the tests, sooner or later, name conflicts could happen if we were to try to run many test suites at once. Not all tests in the `regis_server_tests` suite follow this advice, mostly for demonstration purposes.

This covers all the features for the tests related to the registration of processes. Only those related to unregistering processes remain:

```
unregister_test_() ->
[{"A process that was registered can be registered again iff it was "
  "unregistered between both calls",
  ?setup(fun re_un_register/1)},
 {"Unregistering never crashes",
  ?setup(fun unregister_nocrash/1)},
 {"A crash unregisters a process",
  ?setup(fun crash_unregisters/1)}].
```

Let's see how they are to be implemented. The first one is kind of simple:

```
re_un_register(_) ->
    Ref = make_ref(),
    L = [regis_server:register(Ref, self()),
         regis_server:register(make_ref(), self()),
         regis_server:unregister(Ref),
         regis_server:register(make_ref(), self())],
    [?_assertEqual([ok, {error, already_named}, ok, ok], L)].
```

This way of serializing all the calls in a list is a nifty trick I like to do when I need to test the results of all the events. By putting them in a list, I can then compare the sequence of actions to the expected `[ok, {error, already_named}, ok, ok]` to see how things went. Note that there is nothing specifying that Erlang should evaluate the list in order, but this trick has pretty much always worked for me.

The following test—the one about never crashing—goes like this:

```
unregister_nocrash(_) ->
    ?_assertEqual(ok, regis_server:unregister(make_ref())).
```

Whoa, slow down here, buddy! That's it? Yes it is. The `re_un_register` function already handles testing the "unregistration" of processes. For `unregister_nocrash`, we really only want to know if it will work to try to remove a process that's not there.

Then comes the last test, and one of the most important ones for any test registry you'll ever have: A named process that crashes will have the name unregistered. This has serious implications, because if you don't remove names, you'll end up having an ever-growing registry server with an ever-shrinking name selection.

```
crash_unregisters(_) ->
    Ref = make_ref(),
    Pid = spawn(fun() -> callback(Ref) end),
    timer:sleep(150),
    Pid = regis_server:whereis(Ref),
    exit(Pid, kill),
    timer:sleep(95),
    regis_server:register(Ref, self()),
    S = regis_server:whereis(Ref),
    Self = self(),
    ?_assertEqual(Self, S).
```

This one reads sequentially:

1. Register a process.
2. Make sure the process is registered.
3. Kill that process.
4. Steal the process's identity (the true spy way).
5. Check whether we do hold the name ourselves.

In all honesty, the test could have been written in a simpler manner:

```
crash_unregisters(_) ->
    Ref = make_ref(),
    Pid = spawn(fun() -> callback(Ref) end),
    timer:sleep(150),
    Pid = regis_server:whereis(Ref),
    exit(Pid, kill),
    ?_assertEqual(undefined, regis_server:whereis(Ref)).
```

That whole part about stealing the identity of the dead process was nothing but a petty thief's fantasy.

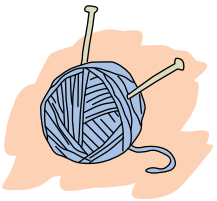
That's it! If you've done things right, you should be able to compile the code and run the test suite:

```
$ erl -make
Recompile: src/regis_sup
... <snip> ...
$ erl -pa ebin/
```

```
1> eunit:test(regis_server).
  All 13 tests passed.
ok
2> eunit:test(regis_server, [verbose]).
======================== EUnit ========================
module 'regis_server'
  module 'regis_server_tests'
    The server can be started, stopped and has a registered name
      regis_server_tests:49: is_registered...ok
      regis_server_tests:50: is_registered...ok
      [done in 0.006 s]
... <snip> ...
  [done in 0.520 s]
=======================================================
  All 13 tests passed.
ok
```

Oh yeah, see how adding the `verbose` option will add test descriptions and runtime information to the reports? That's neat.

He Who Knits EUnits



We've now covered how to use most of EUnit's features to run test suites. More important, you've seen a few techniques related to writing tests for concurrent processes, using patterns that make sense in the real world.

You should know one last trick. When you feel like testing processes such as `gen_server` and `gen_fsm`, you might want to inspect their internal state. Here's a nice way to do this, courtesy of the `sys` module:

```
3> regis_server:start_link().
{ok,<0.160.0>}
4> regis_server:register(shell, self()).
ok
5> sys:get_status(whereis(regis_server)).
{status,<0.160.0>,
        {module,gen_server},
        [[{'$ancestors',[<0.31.0>]},
          {'$initial_call',{regis_server,init,1}}],
         running,<0.31.0>,[],
         [{header,"Status for generic server regis_server"},
          {data,[{"Status",running},
                 {"Parent",<0.31.0>},
                 {"Logged events",[]}]},
          {data,[{"State",
                  {state,{1,{<0.31.0>,{shell,#Ref<0.0.0.333>},nil,nil}},
                         {1,{shell,{<0.31.0>,#Ref<0.0.0.333>},nil,nil}}}}]}]]}
```

Neat, huh? Everything that has to do with the server's innards is given to you, and you can now inspect everything you need, all the time!

If you feel like getting more comfortable with testing servers and whatnot, I recommend reading the tests written for Process Quest's player module at *http://learnyousomeerlang.com/static/erlang/processquest/apps/processquest-1.1.0/test/pq_player_tests.erl*. They test the `gen_server` using a different technique, where all individual calls to `handle_call`, `handle_cast`, and `handle_info` are tried independently. It was still developed in a test-driven manner, but the needs of that approach forced things to be done differently.

You'll see the true value of tests in Chapter 25, when we rewrite the process registry to use ETS, an in-memory database available for all Erlang processes.