Elixir: Mix and OTP

Plataformatec

July 2014

1 Introduction to Mix

In this guide, we will learn how to build a complete Elixir application, with its own supervision tree, configuration, tests and more.

The application works as a distributed key-value store. We are going to organize key-value pairs into buckets and distribute those buckets across multiple nodes. We will also build a simple client that allows us to connect to any of those nodes and send requests such as:

```
CREATE shopping
OK

PUT shopping milk 1
OK

PUT shopping eggs 3
OK

GET shopping milk
1
OK

DELETE shopping eggs
OK
```

In order to build our key-value application, we are going to use three main tools:

- OTP is a set of libraries that ships with Erlang. Erlang developers use OTP to build robust, fault-tolerant applications. In this chapter we will explore how many aspects from OTP integrate with Elixir, including supervision trees, event managers and more;
- Mix is a build tool that ships with Elixir that provides tasks for creating, compiling, testing your application, managing its dependencies and much more;

- ExUnit is a test-unit based framework that ships with Elixir;

In this chapter, we will create our first project using Mix and explore different features in OTP, Mix and ExUnit as we go.

> Note: this guide requires Elixir v0.14.1 or later. You can check your Elixir version with `elixir -v` and install a more recent version if required by following the steps described in the first chapter of the Getting Started guide.

1.1 Our first project

When you install Elixir, besides getting the `elixir`, `elixirc` and `iex` executables, you also get an executable Elixir script named `mix`.

Let's create our first project by invoking `mix new` from the command line. We'll pass the project name as argument (`kv`, in this case), and tell mix that our main module should be the all-uppercase `KV`, instead of the default, which would have been `Kv`:

```
$ mix new kv --module KV
```

Mix will create a directory named `kv` with few files in it:

```
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/kv.ex
* creating test
* creating test/test_helper.exs
* creating test/kv_test.exs
```

Let's take a brief look at those generated files.

> Note: Mix is an Elixir executable. This means that in order to run `mix`, you need to have elixir's executable in your PATH. If not, you can run it by passing the script as argument to elixir:
>
> ```
> $ bin/elixir bin/mix new kv --module KV
> ```
>
> Note that you can also execute any script in your PATH from Elixir via the -S option:
>
> ```
> $ bin/elixir -S mix new kv --module KV
> ```
>
> When using -S, elixir finds the script wherever it is in your PATH and executes it.

1.2 Project compilation

A file named `mix.exs` was generated inside our new project and its main responsibility is to configure our project. Let's take a look at it (comments removed):

```
fmodule KV.Mixfile do
use Mix.Project

def project do
  [app: :kv,
   version: "0.0.1",
   deps: deps]
end

def application do
  [applications: []]
end

defp deps do
  []
end
d
```

Our `mix.exs` defines two public functions: `project`, which returns project configuration like the project name and version, and `application`, which is used to generate an application file.

There is also a private function named `deps`, which is invoked from the `project` function, that defines our project dependencies. Defining `deps` as a separate function is not required, but it helps keep the project configuration tidy.

Mix also generates a file at `lib/kv.ex` with a simple module definition:

```
fmodule KV do
d
```

This structure is enough to compile our project:

```
$ mix compile
```

Will generate:

```
Compiled lib/kv.ex
Generated kv.app
```

Notice the `lib/kv.ex` file was compiled and `kv.app` file was generated. This `.app` file is generated with the information from the `application/0` function in the `mix.exs` file. We will further explore `mix.exs` configuration features in future chapters.

Once the project is compiled, you can start an `iex` session inside the project by running:

```
$ iex -S mix
```

1.3 Running tests

Mix also generated the appropriate structure for running our project tests. Mix projects usually follow the convention of having a `<filename>_test.exs` file in the `test` directory for each file in the `lib` directory. For this reason, we can already find a `test/kv_test.exs` corresponding to our `lib/kv.ex` file. It doesn't do much at this point:

```
1 fmodule KVTest do
2 use ExUnit.Case
3
4 test "the truth" do
5   assert 1 + 1 == 2
6 end
7 d
```

It is important to note a couple things:

1. the test file is an Elixir script file (`.exs`). This is convenient because we don't need to compile test files before running them;
2. we define a test module named `KVTest`, use `ExUnit.Case` to inject the testing API and define a simple test using the `test/2` macro;

Mix also generated a file named `test/test_helper.exs` which is responsible for setting up the test framework:

```
1 Unit.start
```

This file will be automatically required by Mix every time before we run our tests. We can run tests with `mix test`:

```
Compiled lib/kv.ex
Generated kv.app
.
```

```
Finished in 0.04 seconds (0.04s on load, 0.00s on tests)
1 tests, 0 failures

Randomized with seed 540224
```

Notice that by running `mix test`, Mix has compiled the source files and generated the application file once again. This happens because Mix supports multiple environments, which we will explore in the next section.

Furthermore, you can see that ExUnit prints a dot for each successful test and automatically randomize tests too. Let's make the test fail on purpose and see what happens.

Change the assertion in `test/kv_test.exs` to the following:

```
1 sert 1 + 1 == 3
```

Now run `mix test` again (notice this time there was no compilation):

```
1) test the truth (KVTest)
   test/kv_test.exs:4
   Assertion with == failed
   code: 1 + 1 == 3
   lhs:  2
   rhs:  3
   stacktrace:
     test/kv_test.exs:5

Finished in 0.05 seconds (0.05s on load, 0.00s on tests)
1 tests, 1 failures
```

For each failure, ExUnit prints a detailed report, containing the test name with the test case, the code that failed and the values for the left-hand side (lhs) and right-hand side (rhs) of the `==` operator.

In the second line of the failure, right below the test name, there is the location the test was defined. If you copy the test location in the second line, containing the file and line, and paste it in-front of `mix test`, Mix will load and run just that particular test:

```
$ mix test test/kv_test.exs:4
```

This shortcut will be extremely useful as we build our project, allowing us to quickly iterate by running just a specific test.

Finally, the stacktrace relates to the failure itself, giving information about the test and often the place the failure was generated from within the source files.

1.4 Environments

Mix supports the concept of "environments". They allow a developer to customize compilation and other options for specific scenarios. By default, Mix understands three environments:

- `:dev` - the one in which mix tasks (like `compile`) run by default
- `:test` - used by `mix test`
- `:prod` - the one you will use to put your project in production

> Note: If you add dependencies to your project, they will not inherit your project's environment, but instead run with their `:prod` environment settings!

By default, these environments behave the same and all configuration we have seen so far will affect all three environments. Customization per environment can be done by accessing the `Mix.env` function in your `mix.exs` file, which returns the current environment as an atom:

```
f project do
[deps_path: deps_path(Mix.env)]
d

fp deps_path(:prod), do: "prod_deps"
fp deps_path(_), do: "deps"
```

Mix will default to the `:dev` environment, except for the `test` task that will default to the `:test` environment. The environment can be changed via the `MIX_ENV` environment variable:

```
$ MIX_ENV=prod mix compile
```

1.5 Exploring

There is much more to Mix, and we will continue to explore it as we build our project. Keep in mind that you can always invoke the help task to list all available tasks:

```
$ mix help
```

You can get further information about a particular by invoking `mix help TASK`.

Let's write some code.

2 Agent

In this chapter, we will create a module named `KV.Bucket`. This module will be responsible for storing our key-value entries in a way that allows reading and modification by different processes.

If you have skipped the Getting Started guide or if you have read it long ago, be sure to re-read the chapter about Processes. We will use it as starting point.

2.1 The trouble with state

Elixir is an immutable language where nothing is shared by default. If we want to create buckets, store and access them from multiple places, we have two main options in Elixir:

- Processes
- ETS (Erlang Term Storage)

We have talked about processes, while ETS is something new that we will explore later in this guide. When it comes to processes though, we rarely hand-roll our own process, instead we use the abstractions available in Elixir and OTP:

- Agent - Simple wrappers around state
- GenServer - "Generic servers" (processes) that encapsulate state, provide sync and async calls, support code reloading, and more
- GenEvent - "Generic event" managers that allow publishing events to multiple handlers
- Task - Asynchronous units of computation that allow spawning a process and easily retrieving its result at a later time

We will explore all of these abstractions in this guide. Keep in mind that they are all implemented on top of processes using the basic features provided by the VM, like `send`, `receive`, `spawn` and `link`.

2.2 Agents

Agents are simple wrappers around state. If all you want from a process is to keep state, agents are a great fit. Let's start an `iex` session inside the project with:

```
$ iex -S mix
```

And play a bit with agents:

```
1 x> {:ok, agent} = Agent.start_link fn -> [] end
2 ok, #PID<0.57.0>}
```

```
x> Agent.update(agent, fn list -> ["eggs"|list] end)
k
x> Agent.get(agent, fn list -> list end)
eggs"]
x> Agent.stop(agent)
k
```

We started an agent with an initial state of an empty list. Next, we issue a command to update the state, adding our new item to the head of the list. Finally, we retrieved the whole list. Once we are done with the agent, we can call `Agent.stop/1` to terminate the agent process.

Let's implement our `KV.Bucket` using agents. But before starting the implementation, let's first write some tests. Create a file at `test/kv/bucket_test.exs` (remember the `.exs` extension) with the following:

```
fmodule KV.BucketTest do
use ExUnit.Case, async: true

test "stores values by key" do
  {:ok, bucket} = KV.Bucket.start_link
  assert KV.Bucket.get(bucket, "milk") == nil

  KV.Bucket.put(bucket, "milk", 3)
  assert KV.Bucket.get(bucket, "milk") == 3
end
d
```

Our first test is straightforward. We start a new `KV.Bucket` and perform some `get/2` and `put/3` operations on it, asserting the result. We don't need to explicitly stop the agent because it is linked to the test process and the agent is shut down automatically once the test finishes.

Also note that we passed the `async: true` option to `ExUnit.Case`. This option makes this test case run in parallel with other test cases that set up the `:async` option. This is extremely useful to speed up our test suite by using multiple cores in our machine. Note though the `:async` option must only be set if the test case does not rely or change any global value. For example, if the test requires writing to the filesystem, registering processes, accessing a database, you must not make it async to avoid race conditions in between tests.

Regardless of being async or not, our new test should obviously fail, as none of the functionality is implemented.

In order to fix the failing test, let's create a file at `lib/kv/bucket.ex` with the contents below. Feel free to give a try at implementing the `KV.Bucket` module yourself using agents before peeking the implementation below.

```
fmodule KV.Bucket do
@doc """
Starts a new bucket.
"""
def start_link do
  Agent.start_link(fn -> HashDict.new end)
end

@doc """
Gets a value from the `bucket` by `key`.
"""
def get(bucket, key) do
  Agent.get(bucket, &HashDict.get(&1, key))
end

@doc """
Puts the `value` for the given `key` in the `bucket`.
"""
def put(bucket, key, value) do
  Agent.update(bucket, &HashDict.put(&1, key, value))
end
d
```

With the `KV.Bucket` module defined, our test should pass! Note that we are using a HashDict to store our state instead of a `Map`, because in the current version of Elixir maps are less efficient when holding a large number of keys.

2.3 ExUnit callbacks

Before moving on and adding more features to `KV.Bucket`, let's talk about ExUnit callbacks. As you may expect, all `KV.Bucket` tests will require a bucket to be started during setup and stopped after the test. Luckily, ExUnit supports callbacks that allow us to skip such repetitive tasks.

Let's rewrite the test case to use callbacks:

```
fmodule KV.BucketTest do
use ExUnit.Case, async: true

setup do
  {:ok, bucket} = KV.Bucket.start_link
  {:ok, bucket: bucket}
end

test "stores values by key", %{bucket: bucket} do
  assert KV.Bucket.get(bucket, "milk") == nil
```

```
11
12    KV.Bucket.put(bucket, "milk", 3)
13    assert KV.Bucket.get(bucket, "milk") == 3
14  end
15  d
```

We have first defined a setup callback with the help of the `setup/1` macro. The `setup/1` callback runs before every test, in the same process as the test itself.

Note that we need a mechanism to pass the `bucket` pid from the callback to the test. We do so by using the *test context*. When we return `{:ok, bucket: bucket}` from the callback, ExUnit will merge the second element of the tuple (a dictionary) into the test context. The test context is a map which we can then match in the test definition, providing access to these values inside the block:

```
1  st "stores values by key", %{bucket: bucket} do
2  # ‘bucket‘ is now the bucket from the setup block
3  d
```

You can read more about ExUnit cases in the `ExUnit.Case` module documentation and more about callbacks in `ExUnit.Callbacks` docs.

2.4 Other Agent actions

Besides getting a value and updating the agent state, agents allow us to get a value and update the agent state in one function call via `Agent.get_and_update/2`. Let's implement a `KV.Bucket.delete/2` function that deletes a key from the bucket, returning its current value:

```
1  oc """
2  letes ‘key‘ from ‘bucket‘.
3
4  turns the current value of ‘key‘, if ‘key‘ exists.
5  "
6  f delete(bucket, key) do
7  Agent.get_and_update(bucket, &HashDict.pop(&1, key))
8  d
```

Now it is your turn to write a test for the functionality above! Also, be sure to explore the documentation for Agents to learn more about them.

2.5 Client/Server in Agents

Before we move on to the next chapter, let's discuss the client/server dichotomy in agents. Let's expand the `delete/2` function we have just implemented:

```
f delete(bucket, key) do
Agent.get_and_update(bucket, fn dict->
  HashDict.pop(dict, key)
end)
d
```

Everything that is inside the function we passed to the agent happens in the agent process. In this case, since the agent process is the one receiving and responding to our messages, we say the agent process is the server. Everything outside the function is happening in the client.

This distinction is important. If there are expensive actions to be done, you must consider if it will be better to perform these actions on the client or on the server. For example:

```
f delete(bucket, key) do
:timer.sleep(1000) # sleeps the client
Agent.get_and_update(bucket, fn dict ->
  :timer.sleep(1000) # sleeps the server
  HashDict.pop(dict, key)
end)
d
```

When a long action is performed on the server, all other requests to that particular server will wait until the action is done, which may cause some clients to timeout.

In the next chapter we will explore GenServers, where the segregation between clients and servers is made even more apparent.

3 GenServer

In the previous chapter we used agents to represent our buckets. In the first chapter, we specified we would like to name each bucket so we can do the following:

```
EATE shopping


T shopping milk 1
```

```
T shopping milk
```

Since agents are processes, each bucket has a process identifier (pid) but it doesn't have a name. We have learned about the name registry in the Process chapter and you could be inclined to solve this problem by using such registry. For example, we could create a bucket as:

```
x> Agent.start_link(fn -> [] end, name: :shopping)
ok, #PID<0.43.0>}
x> KV.Bucket.put(:shopping, "milk", 1)
k
x> KV.Bucket.get(:shopping, "milk")
```

However, this a terrible idea! Local names in Elixir must be atoms, which means we would need to convert the bucket name (often received from an external client) to atoms, and **we should never convert user input to atoms**. This is because atoms are not garbage collected. Once an atom is created, it is never reclaimed. Generating atoms from user input would mean the user can inject enough different names to exhaust our system memory! In practice it is more likely you will reach the Erlang VM limit for the maximum number of atoms before you run out of memory, which will bring your system down regardless.

Instead of abusing the name registry facility, we will instead create our own *registry process* that holds a dictionary that associates the bucket name to the bucket process.

The registry needs to guarantee the dictionary is always up to date. For example, if one of the bucket processes crashes due to a bug, the registry must clean up the dictionary in order to avoid serving stale entries. In Elixir, we describe this by saying that the registry needs to *monitor* each bucket.

We will use a GenServer to create a registry process that can monitor the bucket process. GenServers are the go-to abstraction for building generic servers in both Elixir and OTP.

3.1 Our first GenServer

A GenServer is implemented in two parts: the client API and the server callbacks, all in a single module. Create a new file at `lib/kv/registry.ex` with the following contents:

```
fmodule KV.Registry do
use GenServer

## Client API

@doc """
Starts the registry.
"""
def start_link(opts \\ []) do
  GenServer.start_link(__MODULE__, :ok, opts)
end

@doc """
Looks up the bucket pid for `name` stored in `server`.

Returns `{:ok, pid}` if the bucket exists, `:error` otherwise.
"""
def lookup(server, name) do
  GenServer.call(server, {:lookup, name})
end

@doc """
Ensures there is a bucket associated to the given `name` in `server`.
"""
def create(server, name) do
  GenServer.cast(server, {:create, name})
end

## Server Callbacks

def init(:ok) do
  {:ok, HashDict.new}
end

def handle_call({:lookup, name}, _from, names) do
  {:reply, HashDict.fetch(names, name), names}
end

def handle_cast({:create, name}, names) do
  if HashDict.get(names, name) do
    {:noreply, names}
  else
    {:ok, bucket} = KV.Bucket.start_link()
    {:noreply, HashDict.put(names, name, bucket)}
  end
```

```
46 end
47 d
```

The first function is `start_link/1`, which starts a new GenServer passing three arguments:

1. The module where the server callbacks are implemented, in this case `__MODULE__`, meaning the current module
2. The initialization arguments, in this case the atom `:ok`
3. A list of options which can, for example, hold the name of the server

There are two types of requests you can send to a GenServer: calls and casts. Calls are synchronous and the server **must** send a response back to such requests. Casts are asynchronous and the server won't send a response back.

The next two functions, `lookup/2` and `create/2` are responsible for sending these requests to the server. The requests are represented by the first argument to `handle_call/3` or `handle_cast/2`. In this case, we have used {`:lookup, name`} and {`:create, name`} respectively. Requests are often specified as tuples, like this, in order to provide more than one "argument" in that first argument slot. It's common to specify the action being requested as the first element of a tuple, and arguments for that action in the remaining elements.

On the server side, we can implement a variety of callbacks to guarantee the server initialization, termination and handling of requests. Those callbacks are optional and for now we have only implemented the ones we care about.

The first is the `init/1` callback, that receives the argument given `GenServer.start_link/3` and returns {`:ok, state`}, where state is a new `HashDict`. We can already notice how the `GenServer` API makes the client/server segregation more apparent. `start_link/3` happens in the client, while `init/1` is the respective callback that runs on the server.

For `call` requests, we must implement a `handle_call/3` callback that receives the `request`, the process from which we received the request (`_from`), and the current server state (`names`). The `handle_call/3` callback returns a tuple in the format {`:reply, reply, new_state`}, where `reply` is what will be sent to the client and the `new_state` is the new server state.

For `cast` requests, we must implement a `handle_cast/2` callback that receives the `request` and the current server state (`names`). The `handle_cast/2` callback returns a tuple in the format {`:noreply, new_state`}.

There are other tuple formats both `handle_call/3` and `handle_cast/2` callbacks may return. There are also other callbacks like `terminate/2` and `code_change/3` that we could implement. You are welcome to explore the full GenServer documentation to learn more about those.

For now, let's write some tests to guarantee our GenServer works as expected.

3.2 Testing a GenSever

Testing a GenServer is not much different from testing an agent. We will spawn the server on a setup callback and use it throughout our tests. Create a file at `test/kv/registry_test.exs` with the following:

```
fmodule KV.RegistryTest do
use ExUnit.Case, async: true

setup do
  {:ok, registry} = KV.Registry.start_link
  {:ok, registry: registry}
end

test "spawns buckets", %{registry: registry} do
  assert KV.Registry.lookup(registry, "shopping") == :error

  KV.Registry.create(registry, "shopping")
  assert {:ok, bucket} = KV.Registry.lookup(registry, "shopping")

  KV.Bucket.put(bucket, "milk", 1)
  assert KV.Bucket.get(bucket, "milk") == 1
end
d
```

Our test should pass right out of the box!

To shutdown the registry, we are simply sending a `:shutdown` signal to its process when our test finishes. While this solution is ok for tests, if there is a need to stop a `GenServer` as part of the application logic, it is best to define a `stop/1` function that sends a `call` message causing the server to stop:

```
 Client API

oc """
ops the registry.
"
f stop(server) do
GenServer.call(server, :stop)
d

 Server Callbacks

f handle_call(:stop, _from, state) do
{:stop, :normal, :ok, state}
d
```

In the example above, the new `handle_call/3` clause is returning the atom `:stop`, along side the reason the server is being stopped (`:normal`), the reply `:ok` and the server state.

3.3 The need for monitoring

Our registry is almost complete. The only remaining issue is that the registry may become stale if a bucket stops or crashes. Let's add a test to `KV.RegistryTest` that exposes this bug:

```
st "removes buckets on exit", %{registry: registry} do
KV.Registry.create(registry, "shopping")
{:ok, bucket} = KV.Registry.lookup(registry, "shopping")
Agent.stop(bucket)
assert KV.Registry.lookup(registry, "shopping") == :error
d
```

The test above will fail on the last assertion as the bucket name remains in the registry even after we stop the bucket process.

In order to fix this bug, we need the registry to monitor every bucket it spawns. Once we set up a monitor, the registry will receive a notification every time a bucket exits, allowing us to clean the dictionary up.

Let's first play with monitors by starting a new console with `iex -S mix`:

```
x> {:ok, pid} = KV.Bucket.start_link
ok, #PID<0.66.0>}
x> Process.monitor(pid)
eference<0.0.0.551>
x> Agent.stop(pid)
k
x> flush()
DOWN, #Reference<0.0.0.551>, :process, #PID<0.66.0>, :normal}
```

Note `Process.monitor(pid)` returns a unique reference that allows us to match upcoming messages to that monitoring reference. After we stop the agent, we can `flush()` all messages and notice a `:DOWN` message arrived, with the exact reference returned by monitor, notifying the bucket process exited with reason `:normal`.

Let's reimplement the server callbacks to fix the bug and make the test pass. First, we will modify the GenServer state to two dictionaries: one that contains `name -> pid` and another that holds `ref -> name`. Then we need to monitor the buckets on `handle_cast/2` as well as implement a `handle_info/2` callback

to handle the monitoring messages. The full server callbacks implementation is shown below:

```
 Server callbacks

f init(:ok) do
names = HashDict.new
refs  = HashDict.new
{:ok, {names, refs}}
d

f handle_call({:lookup, name}, _from, {names, _} = state) do
{:reply, HashDict.fetch(names, name), state}
d

f handle_call(:stop, _from, state) do
{:stop, :normal, :ok, state}
d

f handle_cast({:create, name}, {names, refs}) do
if HashDict.get(names, name) do
  {:noreply, {names, refs}}
else
  {:ok, pid} = KV.Bucket.start_link()
  ref = Process.monitor(pid)
  refs = HashDict.put(refs, ref, name)
  names = HashDict.put(names, name, pid)
  {:noreply, {names, refs}}
end
d

f handle_info({:DOWN, ref, :process, _pid, _reason}, {names, refs}) do
{name, refs} = HashDict.pop(refs, ref)
names = HashDict.delete(names, name)
{:noreply, {names, refs}}
d

f handle_info(_msg, state) do
{:noreply, state}
d
```

Observe that we were able to considerably change the server implementation without changing any of the client API. That's one of the benefits of explicitly segregating the server and the client.

Finally, different from the other callbacks, we have defined a "catch-all" clause for `handle_info/2` that discards any unknown message. To understand why, let's move on to the next section.

3.4 call, cast or info?

So far we have used three callbacks: `handle_call/3`, `handle_cast/2` and `handle_info/2`. Deciding when to use each is straightforward:

1. `handle_call/3` must be used for synchronous requests. This should be the default choice as waiting for the server reply is a useful backpressure mechanism.
2. `handle_cast/2` must be used for asynchronous requests, when you don't care about a reply. A cast does not even guarantee the server has received the message and, for this reason, must be used sparingly. For example, the `create/2` function we have defined in this chapter should have used `call/2`. We have used `cast/2` for didatic purposes.
3. `handle_info/2` must be used for all other messages a server may receive that are not sent via `GenServer.call/2` or `GenServer.cast/2`, including regular messages sent with `send/2`. The monitoring `:DOWN` messages are a perfect example of this.

Since any message, including the ones sent via `send/2`, go to `handle_info/2`, there is a chance unexpected messages will arrive to the server. Therefore, if we don't define the `catch-all` clause, those messages could lead our supervisor to crash, because no clause would match.

We don't need to worry about this for `handle_call/3` and `handle_cast/2` because these requests are only done via the `GenServer` API, so an unknown message is quite likely to be due to a developer mistake.

3.5 Monitors or links?

We have previously learned about links in the Process chapter. Now, with the registry complete, you may be wondering: when should we use monitors and when should we use links?

Links are bi-directional. If you link two process and one of them crashes, the other side will crash too (unless it is trapping exits). A monitor is uni-directional: only the monitoring process will receive notifications about the the monitored one. Simply put, use links when you want linked crashes, and monitors when you just want to be informed of crashes, exits, and so on.

Returning to our `handle_cast/2` implementation, you can see the registry is both linking and monitoring the buckets:

```
1 ok, pid} = KV.Bucket.start_link()
2 f = Process.monitor(pid)
```

This is a bad idea, as we don't want the registry to crash when a bucket crashes! We will explore solutions to this problem when we talk about supervisors. In a nutshell, we typically avoid creating new processes directly. Instead, we delegate this responsibility to supervisors. As we'll see, supervisors work with links, and that explains why link-based APIs (`spawn_link`, `start_link`, etc) are so prevalent in Elixir and OTP.

Before jumping into supervisors, let's first explore event managers and event handlers with GenEvent.

4 GenEvent

In this chapter, we will explore GenEvent, another behaviour provided by Elixir and OTP that allows us to spawn an event manager that is able to publish events to many handlers.

There are two events we are going to emit: one for every time a bucket is added to the registry and another when it is removed from it.

4.1 Event managers

Let's start a new `iex -S mix` session and explore the GenEvent API a bit:

```
x> {:ok, manager} = GenEvent.start_link
ok, #PID<0.83.0>}
x> GenEvent.sync_notify(manager, :hello)
k
x> GenEvent.notify(manager, :world)
k
```

`GenEvent.start_link/0` starts a new event manager. That is literally all that is required to start a manager. After the manager is created, we can call `GenEvent.notify/2` and `GenEvent.sync_notify/2` to send notifications.

However, since there are no event handlers tied to the manager, not much happens on every notification.

Let's create our first handler, still on IEx, that sends all events to a given process:

```
x> defmodule Forwarder do
.>   use GenEvent
.>   def handle_event(event, parent) do
.>     send parent, event
.>     {:ok, parent}
.>   end
.> end
```

```
x> GenEvent.add_handler(manager, Forwarder, self())
k
x> GenEvent.sync_notify(manager, {:hello, :world})
k
x> flush
hello, :world}
k
```

We created our handler and added it to the manager by calling `GenEvent.add_handler/3` passing:

1. The manager we previously started and linked
2. The event handler module (named `Forwarder`) we just defined
3. The event handler state: in this case, the current process pid

After adding the handler, we can see that by calling `sync_notify/2`, the `Forwarder` handler successfully forwards events to our inbox.

There are a couple things that are important to highlight at this point:

1. The event handler runs in the same process as the event manager
2. `sync_notify/2` runs event handlers synchronously to the request
3. `notify/2` runs event handlers asynchronously

Therefore, `sync_notify/2` and `notify/2` are similar to `call/2` and `cast/2` in GenServer and using `sync_notify/2` is generally recommended. It works as a backpressure mechanism in the calling process, to reduce the likelihood of messages being sent more quickly than they can be dispatched to handlers.

Be sure to check other funcionality provided by GenEvent in its module documentation. For now we have enough knowledge to add an event manager to our application.

4.2 Registry events

In order to emit events, we need to change the registry to work with an event manager. While we could automatically start the event manager when the registry is started, for example in the `init/1` callback, it is preferrable to pass the event manager pid/name to `start_link`, decoupling the start of the event manager from the registry.

Let's first change our tests to showcase the behaviour we want the registry to exhibit. Open up `test/kv/registry_test.exs` and change the existing `setup` callback to the one below, then add the new test:

```
fmodule Forwarder do
use GenEvent

```

```
def handle_event(event, parent) do
  send parent, event
  {:ok, parent}
end
d

tup do
{:ok, manager} = GenEvent.start_link
{:ok, registry} = KV.Registry.start_link(manager)

GenEvent.add_handler(manager, Forwarder, self(), link: true)
{:ok, registry: registry}
d

st "sends events on create and crash", %{registry: registry} do
KV.Registry.create(registry, "shopping")
{:ok, bucket} = KV.Registry.lookup(registry, "shopping")
assert_receive {:create, "shopping", ^bucket}

Agent.stop(bucket)
assert_receive {:exit, "shopping", ^bucket}
d
```

In order to test the functionality we want to add, we first define a `Forwarder` event handler similar to the one we typed in IEx previously. On `setup`, we start the event manager, pass it as an argument to the registry and add our `Forwarder` handler to the manager so events can be sent to the test process.

In the test, we create and stop a bucket process and use `assert_receive` to assert we will receive both `:create` and `:exit` messages. `assert_receive` has a default timeout of 500ms which should be more than enough for our tests. Also note that `assert_receive` expects a pattern, rather than a value, that's why we have used `^bucket` to match on the bucket pid.

Finally, notice when calling `GenEvent.add_handler/4`, we passed `link: true` as an option. This links the event manager with the current process and, if the current process dies, the event handler is automatically removed. This makes sense because, in the `Forwarder` case, it makes no sense to continue forwarding messages if the recipient of those messages (`self()`/the test process) is no longer alive.

Let's now change the registry to make the tests pass. Open up `lib/kv/registry.ex` and paste the new registry implementation below (comments inlined):

```
fmodule KV.Registry do
use GenServer

## Client API
```

```
5
6  @doc """
7  Starts the registry.
8  """
9  def start_link(event_manager, opts \\ []) do
10   # 1. start_link now expects the event manager as argument
11   GenServer.start_link(__MODULE__, event_manager, opts)
12 end
13
14 @doc """
15 Looks up the bucket pid for `name` stored in `server`.
16
17 Returns `{:ok, pid}` in case a bucket exists, `:error` otherwise.
18 """
19 def lookup(server, name) do
20   GenServer.call(server, {:lookup, name})
21 end
22
23 @doc """
24 Ensures there is a bucket associated with the given `name` in `server`.
25 """
26 def create(server, name) do
27   GenServer.cast(server, {:create, name})
28 end
29
30 ## Server callbacks
31
32 def init(events) do
33   # 2. The init callback now receives the event manager.
34   #    We have also changed the manager state from a tuple
35   #    to a map, allowing us to add new fields in the future
36   #    without needing to rewrite all callbacks.
37   names = HashDict.new
38   refs  = HashDict.new
39   {:ok, %{names: names, refs: refs, events: events}}
40 end
41
42 def handle_call({:lookup, name}, _from, state) do
43   {:reply, HashDict.fetch(state.names, name), state}
44 end
45
46 def handle_cast({:create, name}, state) do
47   if HashDict.get(state.names, name) do
48     {:noreply, state}
49   else
50     {:ok, pid} = KV.Bucket.start_link()
```

```
    ref = Process.monitor(pid)
    refs = HashDict.put(state.refs, ref, name)
    names = HashDict.put(state.names, name, pid)
    # 3. Push a notification to the event manager on create
    GenEvent.sync_notify(state.events, {:create, name, pid})
    {:noreply, %{state | names: names, refs: refs}}
  end
end

def handle_info({:DOWN, ref, :process, pid, _reason}, state) do
  {name, refs} = HashDict.pop(state.refs, ref)
  names = HashDict.delete(state.names, name)
  # 4. Push a notification to the event manager on exit
  GenEvent.sync_notify(state.events, {:exit, name, pid})
  {:noreply, %{state | names: names, refs: refs}}
end

def handle_info(_msg, state) do
  {:noreply, state}
end
d
```

The changes are straightforward. We now pass the event manager we received as an argument to `start_link` on to `GenServer` initialization. We also change both cast and info callbacks to call `GenEvent.sync_notify/2`. Lastly, we have taken the opportunity to change the server state to a map, making it easier to improve the registry in the future.

Run the test suite, and all tests should be green again.

4.3 Event streams

One last functionality worth exploring from `GenEvent` is the ability to consume its events as a stream:

```
x> {:ok, manager} = GenEvent.start_link
ok, #PID<0.83.0>}
x> spawn_link fn ->
.>   for x <- GenEvent.stream(manager), do: IO.inspect(x)
.> end
k
x> GenEvent.notify(manager, {:hello, :world})
hello, :world}
k
```

In the example above, we have created a `GenEvent.stream(manager)` that returns a stream (an enumerable) of events that are consumed as they come. Since consuming those events is a blocking action, we spawn a new process that will consume the events and print them to the terminal, and that is exactly the behaviour we see. Every time we call `sync_notify/2` or `notify/2`, the event is printed to the terminal followed by `:ok` (which is just IEx printing the result returned by notify functions).

Often event streams provide enough functionality for consuming events that we don't need to register our own handlers. However, when custom functionality is required, or during testing, defining our own event handler callbacks is the best way to go.

At this point, we have an event manager, a registry and potentially many buckets running at the same time. It is about time to start worrying what would happen if any of those processes crash.

5 Supervisor and Application

So far our application requires an event manager and a registry. It may potentially use dozens, if not hundreds, of buckets. While we may think our implementation so far is quite good, no software is bug free, and failures are definitely going to happen.

When things fail, our first reaction may be: "let's rescue those errors". But, as we have learned in the Getting Started guide, in Elixir we don't have the defensive programming habit of rescuing exceptions, as commonly seen in other languages. Instead, we say "fail fast" or "let it crash". If there is a bug that leads our registry to crash, we have nothing to worry about because we are going to setup a supervisor that will start a fresh copy of the registry.

In this chapter, we are going to learn about supervisors and also about applications. We are going to create not one, but two supervisors, and use them to supervise our processes.

5.1 Our first Supervisor

Creating a supervisor is not much different from creating a GenServer. We are going to define a module named `KV.Supervisor`, which will use the Supervisor behaviour, inside the `lib/kv/supervisor.ex` file:

```
1 fmodule KV.Supervisor do
2 use Supervisor
3
4 def start_link do
5   Supervisor.start_link(__MODULE__, :ok)
6 end
7
```

```
@manager_name KV.EventManager
@registry_name KV.Registry

def init(:ok) do
  children = [
    worker(GenEvent, [[name: @manager_name]]),
    worker(KV.Registry, [@manager_name, [name: @registry_name]])
  ]

  supervise(children, strategy: :one_for_one)
end
d
```

Our supervisor has two children: the event manager and the registry. It's common to give names to processes under supervision so that other processes can access them by name without needing to know their pid. This is useful because a supervised process might crash, in which case its pid will change when the supervisor restarts it. We declare the names of our supervisor's children by using the module attributes `@manager_name` and `@registry_name`, then reference those attributes in the worker definitions. While it's not required that we declare the names of our child processes in module attributes, it's helpful, because doing so helps make them stand out to the reader of our code.

For example, the `KV.Registry` worker receives two arguments, the first is the name of the event manager and the second is a keyword list of options. In this case, we set the name option to `[name: KV.Registry]` (using our previously-defined module attribute, `@registry_name`), guaranteeing we can access the registry by the name `KV.Registry` throughout the application. It is very common to name the children of a supervisor after the module that defines them, as this association becomes very handy when debugging a live system.

The order children are declared in the supervisor also matters. Since the registry depends on the event manager, we must start the latter before the former. That's why the `GenEvent` worker must come before the `KV.Registry` worker in the children list.

Finally, we call `supervise/2`, passing the list of children and the strategy of `:one_for_one`.

The supervision strategy dictates what happens when one of the children crashes. `:one_for_one` means that if a child dies only one is restarted to replace it. This strategy makes sense for now. If the event manager crashes, there is no reason to restart the registry and vice-versa. However, those dynamics may change once we add more children to supervisor. The `Supervisor` behaviour supports many different strategies and we will discuss three of them in this chapter.

If we start a console inside our project using `iex -S mix`, we can manually start the supervisor:

```
x> KV.Supervisor.start_link
ok, #PID<0.66.0>}
x> KV.Registry.create(KV.Registry, "shopping")
k
x> KV.Registry.lookup(KV.Registry, "shopping")
ok, #PID<0.70.0>}
```

When we started the supervisor tree, both the event manager and registry worker were automatically started, allowing us to create buckets without the need to manually start these processes.

In practice though, we rarely start the application supervisor manually. Instead it is started as part of the application callback.

5.2 Understanding applications

We have been working inside an application this entire time. Every time we changed a file and ran `mix compile`, we could see `Generated kv.app` message in the compilation output.

We can find the generated `.app` file at `_build/dev/lib/kv/ebin/kv.app`. Let's have a look at its contents:

```
pplication,kv,
            [{registered,[]},
             {description,"kv"},
             {applications,[kernel,stdlib,elixir]},
             {vsn,"0.0.1"},
             {modules,['Elixir.KV','Elixir.KV.Bucket',
                       'Elixir.KV.Registry','Elixir.KV.Supervisor']}]}.
```

This file contains Erlang terms (written using Erlang syntax). Even though we are not familiar with Erlang, it is easy to guess this file holds our application definition. It contains our application `version`, all the modules defined by it, as well as a list of applications we depend on, like Erlang's `kernel` and `elixir` itself.

It would be pretty boring to update this file manually every time we add a new module to our application. That's why mix generates and maintains it automatically for us.

We can also configure the generated `.app` file by customizing the values returned by the `application/0` inside our `mix.exs` project file. We will get to that in upcoming chapters.

5.2.1 .1 Starting applications

When we define an `.app` file, which is the application definition, we are able to start and stop the application as a whole. We haven't worried about this so far

for two reasons:

1. Mix automatically starts our current application for us
2. Even if Mix didn't start our application for us, our application does not yet need to do anything when it starts

In any case, let's see how Mix starts the application for us. Let's start a project console with `iex -S mix` and try:

```
x> Application.start(:kv)
error, {:already_started, :kv}}
```

Oops, it's already started.

We can pass an option to mix to ask it to not start our application. Let's give it a try by running `iex -S mix run --no-start`:

```
x> Application.start(:kv)
k
x> Application.stop(:kv)

NFO REPORT==== 6-Jun-2014::18:24:26 ===
  application: kv
  exited: stopped
  type: temporary
k
```

Nothing really exciting happens but it shows how we can control our application.

> When you run `iex -S mix`, it is equivalent to running `iex -S mix run`. So whenever you need to pass more options to mix when starting iex, it's just a matter of typing `mix run` and then passing any options the `run` command accepts. You can find more information about `run` by running `mix help run` in your shell.

5.2.2 .2 The application callback

Since we spent all this time talking about how applications are started and stopped, there must be a way to do something useful when the application starts. And indeed, there is!

We can specify an application callback function. This is a function that will be invoked when the application starts. The function must return a result of `{:ok, pid}`, where `pid` is the process identifier of a supervisor process.

We can configure the application callback in two steps. First, open up the `mix.exs` file and change `def application` to the following:

```
f application do
[applications: [],
 mod: {KV, []}]
d
```

The `:mod` option specifies the "application callback module", followed by the arguments to be passed on application start. The application callback module can be any module that implements the Application behaviour.

Now that we have specified `KV` as the module callback, we need to change the `KV` module, defined in `lib/kv.ex`:

```
fmodule KV do
use Application

def start(_type, _args) do
  KV.Supervisor.start_link
end
d
```

When we `use Application`, we only need to define a `start/2` function. If we wanted to specify custom behaviour on application stop, we could define a `stop/1` function, as well. In this case, the one automatically defined by `use Application` is fine.

Let's start our project console once again with `iex -S mix`. We will see a process named `KV.Registry` is already running:

```
x> KV.Registry.create(KV.Registry, "shopping")
k
x> KV.Registry.lookup(KV.Registry, "shopping")
ok, #PID<0.88.0>}
```

Excellent!

5.2.3 .3 Projects or applications?

Mix makes a distinction between projects and applications. Based on the current contents of our `mix.exs` file, we would say we have a Mix project that defines the `:kv` application. As we will see in later chapters, there are projects that don't define any application.

When we say "project," you should think about Mix. Mix is the tool that manages your project. It knows how to compile your project, test your project and more. It also knows how to compile and start the application relevant to your project.

When we talk about applications, we talk about OTP. Applications are the entities that are started and stopped as a whole by the runtime. You can learn more about applications in the docs for the Application module, as well as by running `mix help compile.app` to learn more about the supported options in `def application`.

5.3 Simple one for one supervisors

We have now successfully defined our supervisor which is automatically started (and stopped) as part of our application lifecycle.

Remember however that our `KV.Registry` is both linking and monitoring bucket processes in the `handle_cast/2` callback:

```
ok, pid} = KV.Bucket.start_link()
f = Process.monitor(pid)
```

Links are bi-directional, which implies that a crash in a bucket will crash the registry. Although we now have the supervisor, which guarantees the registry will be back up and running, crashing the registry still means we lose all data associating bucket names to their respective processes.

In other words, we want the registry to keep on running even if a bucket crashes. Let's write a test:

```
st "removes bucket on crash", %{registry: registry} do
KV.Registry.create(registry, "shopping")
{:ok, bucket} = KV.Registry.lookup(registry, "shopping")

# Kill the bucket and wait for the notification
Process.exit(bucket, :shutdown)
assert_receive {:exit, "shopping", ^bucket}
assert KV.Registry.lookup(registry, "shopping") == :error
d
```

The test is similar to "removes bucket on exit" except that we are being a bit more harsh. Instead of using `Agent.stop/1`, we are sending an exit signal to shutdown the bucket. Since the bucket is linked to the registry, which is then linked to the test process, killing the bucket causes the registry to crash which then causes the test process to crash too:

```
 test removes bucket on crash (KV.RegistryTest)
 test/kv/registry_test.exs:52
 ** (EXIT from #PID<0.94.0>) shutdown
```

One possible solution to this issue would be to provide a `KV.Bucket.start/0`, that invokes `Agent.start/1`, and use it from the registry, removing the link between registry and buckets. However, this would be a bad idea, because buckets would not be linked to any process after this change. This means that if someone stops the `kv` application, all buckets would remain alive as they are unreachable.

We are going to solve this issue by defining a new supervisor that will spawn and supervise all buckets. There is one supervisor strategy, called `:simple_one_for_one`, that is the perfect fit for such situations: it allows us to specify a worker template and supervise many children based on this template.

Let's define our `KV.Bucket.Supervisor` as follows:

```
1  fmodule KV.Bucket.Supervisor do
2  use Supervisor
3
4  def start_link(opts \\ []) do
5    Supervisor.start_link(__MODULE__, :ok, opts)
6  end
7
8  def start_bucket(supervisor) do
9    Supervisor.start_child(supervisor, [])
10 end
11
12 def init(:ok) do
13   children = [
14     worker(KV.Bucket, [], type: :temporary)
15   ]
16
17   supervise(children, strategy: :simple_one_for_one)
18 end
19 d
```

There are two changes in this supervisor compared to the first one.

First, we define a `start_bucket/1` function that will receive a supervisor and start a bucket process as a child of that supervisor. `start_bucket/1` is the function we are going to invoke instead of calling `KV.Bucket.start_link` directly in the registry.

Second, in the `init/1` callback, we are marking the worker as `:temporary`. This means that if the bucket dies, it won't be restarted! That's because we only want to use the supervisor as a mechanism to group the buckets. The creation of buckets should always pass through the registry.

Run `iex -S mix` so we can give our new supervisor a try:

```
1 x> {:ok, sup} = KV.Bucket.Supervisor.start_link
2 ok, #PID<0.70.0>}
```

```
x> {:ok, bucket} = KV.Bucket.Supervisor.start_bucket(sup)
ok, #PID<0.72.0>}
x> KV.Bucket.put(bucket, "eggs", 3)
k
x> KV.Bucket.get(bucket, "eggs")

```

Let's change the registry to work with the buckets supervisor. We are going to follow the same strategy we did with the events manager, where we will explicitly pass the buckets supervisor pid to `KV.Registry.start_link/3`. Let's start by changing the setup callback in `test/kv/registry_test.exs` to do so:

```
tup do
{:ok, sup} = KV.Bucket.Supervisor.start_link
{:ok, manager} = GenEvent.start_link
{:ok, registry} = KV.Registry.start_link(manager, sup)

GenEvent.add_handler(manager, Forwarder, self(), link: true)
{:ok, registry: registry}
d
```

Now let's change the appropriate functions in `KV.Registry` to take the new supervisor into account:

```
 Client API

oc """
arts the registry.
"
f start_link(event_manager, buckets, opts \\ []) do
# 1. Pass the buckets supevisor as argument
GenServer.start_link(__MODULE__, {event_manager, buckets}, opts)
d

 Server callbacks

f init({events, buckets}) do
names = HashDict.new
refs  = HashDict.new
# 2. Store the buckets supevisor in the state
{:ok, %{names: names, refs: refs, events: events, buckets: buckets}}
d

f handle_cast({:create, name}, state) do
```

```
if HashDict.get(state.names, name) do
  {:noreply, state}
else
  # 3. Use the buckets supervisor instead of starting buckets directly
  {:ok, pid} = KV.Bucket.Supervisor.start_bucket(state.buckets)
  ref = Process.monitor(pid)
  refs = HashDict.put(state.refs, ref, name)
  names = HashDict.put(state.names, name, pid)
  GenEvent.sync_notify(state.events, {:create, name, pid})
  {:noreply, %{state | names: names, refs: refs}}
end
d
```

Those changes should be enough to make our tests pass! To complete our task, we just need to update our supervisor to also take the buckets supervisor as child.

5.4 Supervision trees

In order to use the buckets supervisor in our application, we need to add it as a child of `KV.Supervisor`. Notice we are beginning to have supervisors that supervise other supervisors, forming so-called "supervision trees."

Open up `lib/kv/supervisor.ex`, add an additional module attribute for the buckets supervisor name, and change `init/1` to match the following:

```
anager_name KV.EventManager
egistry_name KV.Registry
ucket_sup_name KV.Bucket.Supervisor

f init(:ok) do
children = [
  worker(GenEvent, [[name: @manager_name]]),
  supervisor(KV.Bucket.Supervisor, [[name: @bucket_sup_name]]),
  worker(KV.Registry, [@manager_name, @bucket_sup_name, [name: @registry_name]])
]

supervise(children, strategy: :one_for_one)
d
```

This time we have added a supervisor as child and given it the name of `KV.Bucket.Supervisor` (again, the same name as the module). We have also updated the `KV.Registry` worker to receive the bucket supervisor name as argument.

Also remember that the order in which children are declared is important. Since the registry depends on the buckets supervisor, the buckets supervisor must be listed before it in the children list.

Since we have added more children to the supervisor, it is important to evaluate if the `:one_for_one` strategy is still correct. One flaw that shows up right away is the relationship between registry and buckets supervisor. If the registry dies, the buckets supervisor must die too, because once the registry dies all information linking the bucket name to the bucket process is lost. If the buckets supervisor is kept alive, it would be impossible to reach those buckets.

We could consider moving to another strategy like `:one_for_all`. The `:one_for_all` strategy kills and restarts all children whenever one of the children die. This change is not ideal either, because a crash in the registry should not crash the event manager. In fact, doing so would be harmful, as crashing the event manager would cause all installed event handlers to be removed.

One possible solution to this problem is to create another supervisor that will supervise the registry and buckets supervisor with `:one_for_all` strategy, and have the root supervisor supervise both the event manager and the new supervisor with `:one_for_one` strategy. The proposed tree would have the following format:

```
root supervisor [one_for_one]
* event manager
* supervisor [one_for_all]
  * buckets supervisor [simple_one_for_one]
    * buckets
  * registry
```

You can take a shot at building this new supervision tree, but we will stop here. This is because in the next chapter we will make changes to the registry that will allow the registry data to be persisted, making the `:one_for_one` strategy a perfect fit.

Remember, there are other strategies and other options that could be given to `worker/2`, `supervisor/2` and `supervise/2` functions, so don't forget to check out the Supervisor module documentation.

6 ETS

Every time we need to look up a bucket, we need to send a message to the registry. In some applications, this means the registry may become a bottleneck!

In this chapter we will learn about ETS (Erlang Term Storage), and how to use it as a cache mechanism. Later we will expand its usage to persist data from the supervisor to its children, allowing data to persist even on crashes.

> Warning! Don't use ETS as a cache prematurely! Log and analyze your application performance and identify which parts are bottlenecks, so you know *whether* you should cache, and *what* you should cache. This chapter is merely an example of how ETS can be used, once you've determined the need.

6.1 ETS as a cache

ETS allows us to store any Erlang/Elixir term in an in-memory table. Working with ETS tables is done via erlang's `:ets` module:

```
x> table = :ets.new(:buckets_registry, [:set, :protected])
07
x> :ets.insert(table, {"foo", self})
ue
x> :ets.lookup(table, "foo")
"foo", #PID<0.41.0>}]
```

When creating an ETS table, two arguments are required: the table name and a set of options. From the available options, we passed the table type and its access rules. We have chosen the `:set` type, which means that keys cannot be duplicated. We've also set the table's access to `:protected`, which means that only the process that created the table can write to it, but all processes can read it from it. Those are actually the default values, so we will skip them from now on.

ETS tables can also be named, allowing us to access them by a given name:

```
x> :ets.new(:buckets_registry, [:named_table])
uckets_registry
x> :ets.insert(:buckets_registry, {"foo", self})
ue
x> :ets.lookup(:buckets_registry, "foo")
"foo", #PID<0.41.0>}]
```

Let's change the `KV.Registry` to use ETS tables. We will use the same technique as we did for the event manager and buckets supervisor, and pass the ETS table name explicitly on `start_link`. Remember that, as with server names, any local process that knows an ETS table name will be able to access that table.

Open up `lib/kv/registry.ex`, and let's change its implementation. We've added comments to the source code to highlight the changes we've made:

```
fmodule KV.Registry do
use GenServer

## Client API

@doc """
Starts the registry.
"""
def start_link(table, event_manager, buckets, opts \\ []) do
  # 1. We now expect the table as argument and pass it to the server
  GenServer.start_link(__MODULE__, {table, event_manager, buckets}, opts)
end

@doc """
Looks up the bucket pid for `name` stored in `table`.

Returns `{:ok, pid}` if a bucket exists, `:error` otherwise.
"""
def lookup(table, name) do
  # 2. lookup now expects a table and looks directly into ETS.
  #    No request is sent to the server.
  case :ets.lookup(table, name) do
    [{^name, bucket}] -> {:ok, bucket}
    [] -> :error
  end
end

@doc """
Ensures there is a bucket associated with the given `name` in `server`.
"""
def create(server, name) do
  GenServer.cast(server, {:create, name})
end

## Server callbacks

def init({table, events, buckets}) do
  # 3. We have replaced the names HashDict by the ETS table
  ets  = :ets.new(table, [:named_table, read_concurrency: true])
  refs = HashDict.new
  {:ok, %{names: ets, refs: refs, events: events, buckets: buckets}}
end

# 4. The previous handle_call callback for lookup was removed

def handle_cast({:create, name}, state) do
```

```
  # 5. Read and write to the ETS table instead of the HashDict
  case lookup(state.names, name) do
    {:ok, _pid} ->
      {:noreply, state}
    :error ->
      {:ok, pid} = KV.Bucket.Supervisor.start_bucket(state.buckets)
      ref = Process.monitor(pid)
      refs = HashDict.put(state.refs, ref, name)
      :ets.insert(state.names, {name, pid})
      GenEvent.sync_notify(state.events, {:create, name, pid})
      {:noreply, %{state | refs: refs}}
  end
end

def handle_info({:DOWN, ref, :process, pid, _reason}, state) do
  # 6. Delete from the ETS table instead of the HashDict
  {name, refs} = HashDict.pop(state.refs, ref)
  :ets.delete(state.names, name)
  GenEvent.sync_notify(state.events, {:exit, name, pid})
  {:noreply, %{state | refs: refs}}
end

def handle_info(_msg, state) do
  {:noreply, state}
end
d
```

Notice that before our changes `KV.Registry.lookup/2` sent requests to the server, but now it reads directly from the ETS table, which is shared across all processes. That's the main idea behind the cache mechanism we are implementing.

In order for the cache mechanism to work, the created ETS table needs to have access `:protected` (the default), so all clients can read from it, while only the `KV.Registry` process writes to it. We have also set `read_concurrency: true` when starting the table, optimizing the table for the common scenario of concurrent read operations.

The changes we have performed above have definitely broken our tests. For starters, there is a new argument we need to pass to `KV.Registry.start_link/3`. Let's start amending our tests in `test/kv/registry_test.exs` by rewriting the `setup` callback:

```
tup do
{:ok, sup} = KV.Bucket.Supervisor.start_link
{:ok, manager} = GenEvent.start_link
{:ok, registry} = KV.Registry.start_link(:registry_table, manager, sup)
```

```
5
6 GenEvent.add_handler(manager, Forwarder, self(), link: true)
7 {:ok, registry: registry, ets: :registry_table}
8 d
```

Notice we are passing the table name of `:registry_table` to `KV.Registry.start_link/3` as well as returning `ets: :registry_table` as part of the test context.

After changing the callback above, we will still have failures in our test suite. All in the format of:

```
1  test spawns buckets (KV.RegistryTest)
2  test/kv/registry_test.exs:38
3  ** (ArgumentError) argument error
4  stacktrace:
5    (stdlib) :ets.lookup(#PID<0.99.0>, "shopping")
6    (kv) lib/kv/registry.ex:22: KV.Registry.lookup/2
7    test/kv/registry_test.exs:39
```

This is happening because we are passing the registry pid to `KV.Registry.lookup/2` while now it expects the ETS table. We can fix this by changing all occurrences of:

```
1 .Registry.lookup(registry, ...)
```

to:

```
1 .Registry.lookup(ets, ...)
```

Where `ets` will be retrieved in the same way we retrieve the registry:

```
1 st "spawns buckets", %{registry: registry, ets: ets} do
```

Let's change our tests to pass `ets` to `lookup/2`. Once we finish these changes, some tests will continue to fail. You may even notice tests pass and fail inconsistently between runs. For example, the "spawns buckets" test:

```
1 st "spawns buckets", %{registry: registry, ets: ets} do
2 assert KV.Registry.lookup(ets, "shopping") == :error
3
```

```
4 KV.Registry.create(registry, "shopping")
5 assert {:ok, bucket} = KV.Registry.lookup(ets, "shopping")
6
7 KV.Bucket.put(bucket, "milk", 1)
8 assert KV.Bucket.get(bucket, "milk") == 1
9 d
```

may be failing on this line:

```
1 sert {:ok, bucket} = KV.Registry.lookup(ets, "shopping")
```

However how can this line fail if we just created the bucket in the previous line?

The reason those failures are happening is because, for didactic purposes, we have made two mistakes:

1. We are prematurely optimizing (by adding this cache layer)
2. We are using `cast/2` (while we should be using `call/2`)

6.2 Race conditions?

Developing in Elixir does not make your code free of race conditions. However, Elixir's simple abstractions where nothing is shared by default make it easier to spot a race condition's root cause.

What is happening in our test is that there is a delay in between an operation and the time we can observe this change in the ETS table. Here is what we were expecting to happen:

1. We invoke `KV.Registry.create(registry, "shopping")`
2. The registry creates the bucket and updates the cache table
3. We access the information from the table with `KV.Registry.lookup(ets, "shopping")`
4. The command above returns `{:ok, bucket}`

However, since `KV.Registry.create/2` is a cast operation, the command will return before we actually write to the table! In other words, this is happening:

1. We invoke `KV.Registry.create(registry, "shopping")`
2. We access the information from the table with `KV.Registry.lookup(ets, "shopping")`
3. The command above returns `:error`
4. The registry creates the bucket and updates the cache table

To fix the failure we just need to make `KV.Registry.create/2` synchronous by using `call/2` rather than `cast/2`. This will guarantee that the client will only continue after changes have been made to the table. Let's change the function and its callback as follows:

```
f create(server, name) do
GenServer.call(server, {:create, name})
d

f handle_call({:create, name}, _from, state) do
case lookup(state.names, name) do
  {:ok, pid} ->
    {:reply, pid, state} # Reply with pid
  :error ->
    {:ok, pid} = KV.Bucket.Supervisor.start_bucket(state.buckets)
    ref = Process.monitor(pid)
    refs = HashDict.put(state.refs, ref, name)
    :ets.insert(state.names, {name, pid})
    GenEvent.sync_notify(state.events, {:create, name, pid})
    {:reply, pid, %{state | refs: refs}} # Reply with pid
end
d
```

We simply changed the callback from `handle_cast/2` to `handle_call/3` and changed it to reply with the pid of the created bucket.

Let's run the tests once again. This time though, we will pass the `--trace` option:

```
$ mix test --trace
```

The `--trace` option is useful when your tests are deadlocking or there are race conditions, as it runs all tests synchronously (`async: true` has no effect) and shows detailed information about each test. This time we should be down to one failure (that may be intermittent):

```
 test removes buckets on exit (KV.RegistryTest)
 test/kv/registry_test.exs:48
 Assertion with == failed
 code: KV.Registry.lookup(ets, "shopping") == :error
 lhs:  {:ok, #PID<0.103.0>}
 rhs:  :error
 stacktrace:
   test/kv/registry_test.exs:52
```

According to the failure message, we are expecting that the bucket no longer exists on the table, but it still does! This problem is the opposite of the one we have just solved: while previously there was a delay between the command to create a bucket and updating the table, now there is a delay between the bucket process dying and its entry being removed from the table.

Unfortunately this time we cannot simply change `handle_info/2` to a synchronous operation. We can, however, fix our tests by using event manager notifications. Let's take another look at our `handle_info/2` implementation:

```
f handle_info({:DOWN, ref, :process, pid, _reason}, state) do
# 5. Delete from the ETS table instead of the HashDict
{name, refs} = HashDict.pop(state.refs, ref)
:ets.delete(state.names, name)
GenEvent.sync_notify(state.event, {:exit, name, pid})
{:noreply, %{state | refs: refs}}
d
```

Notice that we are deleting from the ETS table **before** we send the notification. This is by design! This means that when we receive the `{:exit, name, pid}` notification, the table will already be up to date. Let's update the remaining failing test as follows:

```
st "removes buckets on exit", %{registry: registry, ets: ets} do
KV.Registry.create(registry, "shopping")
{:ok, bucket} = KV.Registry.lookup(ets, "shopping")
Agent.stop(bucket)
assert_receive {:exit, "shopping", ^bucket} # Wait for event
assert KV.Registry.lookup(ets, "shopping") == :error
d
```

We have simply amended the test to guarantee we first receive the `{:exit, name, pid}` message before invoking `KV.Registry.lookup/2`.

It is important to observe that we were able to keep our suite passing without a need to use `:timer.sleep/1` or other tricks. Most of the time, we can rely on events, monitoring and messages to assert the system is in an expected state before performing assertions.

For your convenience, here is the fully passing test case:

```
fmodule KV.RegistryTest do
use ExUnit.Case, async: true

defmodule Forwarder do
  use GenEvent
```

```
6
7   def handle_event(event, parent) do
8     send parent, event
9     {:ok, parent}
10  end
11 end
12
13 setup do
14   {:ok, sup} = KV.Bucket.Supervisor.start_link
15   {:ok, manager} = GenEvent.start_link
16   {:ok, registry} = KV.Registry.start_link(:registry_table, manager, sup)
17
18   GenEvent.add_handler(manager, Forwarder, self(), link: true)
19   {:ok, registry: registry, ets: :registry_table}
20 end
21
22 test "sends events on create and crash", %{registry: registry, ets: ets} do
23   KV.Registry.create(registry, "shopping")
24   {:ok, bucket} = KV.Registry.lookup(ets, "shopping")
25   assert_receive {:create, "shopping", ^bucket}
26
27   Agent.stop(bucket)
28   assert_receive {:exit, "shopping", ^bucket}
29 end
30
31 test "spawns buckets", %{registry: registry, ets: ets} do
32   assert KV.Registry.lookup(ets, "shopping") == :error
33
34   KV.Registry.create(registry, "shopping")
35   assert {:ok, bucket} = KV.Registry.lookup(ets, "shopping")
36
37   KV.Bucket.put(bucket, "milk", 1)
38   assert KV.Bucket.get(bucket, "milk") == 1
39 end
40
41 test "removes buckets on exit", %{registry: registry, ets: ets} do
42   KV.Registry.create(registry, "shopping")
43   {:ok, bucket} = KV.Registry.lookup(ets, "shopping")
44   Agent.stop(bucket)
45   assert_receive {:exit, "shopping", ^bucket} # Wait for event
46   assert KV.Registry.lookup(ets, "shopping") == :error
47 end
48
49 test "removes bucket on crash", %{registry: registry, ets: ets} do
50   KV.Registry.create(registry, "shopping")
51   {:ok, bucket} = KV.Registry.lookup(ets, "shopping")
```

```
52
53    # Kill the bucket and wait for the notification
54    Process.exit(bucket, :shutdown)
55    assert_receive {:exit, "shopping", ^bucket}
56    assert KV.Registry.lookup(ets, "shopping") == :error
57  end
58  d
```

With tests passing, we just need to update the supervisor `init/1` callback at `lib/kv/supervisor.ex` to pass the ETS table name as an argument to the registry worker:

```
1  anager_name KV.EventManager
2  egistry_name KV.Registry
3  ts_registry_name KV.Registry
4  ucket_sup_name KV.Bucket.Supervisor
5
6  f init(:ok) do
7  children = [
8    worker(GenEvent, [[name: @manager_name]]),
9    supervisor(KV.Bucket.Supervisor, [[name: @bucket_sup_name]]),
10   worker(KV.Registry, [@ets_registry_name, @manager_name,
11                        @bucket_sup_name, [name: @registry_name]])
12 ]
13
14 supervise(children, strategy: :one_for_one)
15 d
```

Note that we are using `KV.Registry` as name for the ETS table as well, which makes it convenient to debug, as it points to the module using it. ETS names and process names are stored in different registries, so there is no chance of conflicts.

6.3 ETS as persistent storage

So far we have created an ETS table during the registry initialization but we haven't bothered to close the table on registry termination. That's because the ETS table is "linked" (in a figure of speech) to the process that creates it. If that process dies, the table is automatically closed.

This is extremely convenient as a default behaviour, and we can use it even more to our advantage. Remember that there is a dependency between the registry and the buckets supervisor. If the registry dies, we want the buckets supervisor to die too, because once the registry dies all information linking the bucket name to the bucket process is lost. However, what if we could keep the

registry data even if the registry process crashes? If we are able to do so, we remove the dependency between the registry and the buckets supervisor, making the `:one_for_one` strategy the perfect strategy for our supevisor.

A couple of changes will be required in order to make this happen. First, we'll need to start the ETS table inside the supervisor. Second, we'll need to change the table's access type from `:protected` to `:public`, because the owner is the supervisor, but the process doing the writes is still the manager.

Let's get started by first changing `KV.Supervisor`'s `init/1` callback:

```
f init(:ok) do
ets = :ets.new(@ets_registry_name,
               [:set, :public, :named_table, {:read_concurrency, true}])

children = [
  worker(GenEvent, [[name: @manager_name]]),
  supervisor(KV.Bucket.Supervisor, [[name: @bucket_sup_name]]),
  worker(KV.Registry, [ets, @manager_name,
                       @bucket_sup_name, [name: @registry_name]])
]

supervise(children, strategy: :one_for_one)
d
```

Next, we change `KV.Registry`'s `init/1` callback, as it no longer needs to create a table. It should instead just use the one given as an argument:

```
f init({table, sup, event}) do
refs = HashDict.new
{:ok, %{names: table, refs: refs, sup: sup, event: event}}
d
```

Finally, we just need to change the `setup` callback in `test/kv/registry_test.exs` to explicitly create the ETS table. We will use this opportunity to also split the `setup` functionality into a private function that will be handy soon:

```
tup do
ets = :ets.new(:registry_table, [:set, :public])
registry = start_registry(ets)
{:ok, registry: registry, ets: ets}
d

fp start_registry(ets) do
{:ok, sup} = KV.Bucket.Supervisor.start_link
```

```
{:ok, manager} = GenEvent.start_link
{:ok, registry} = KV.Registry.start_link(ets, manager, sup)

GenEvent.add_handler(manager, Forwarder, self(), link: true)
registry
d
```

After those changes, our test suite should continue to be green!

There is just one last scenario to consider: once we receive the ETS table, there may be existing bucket pids on the table. After all, that's the whole purpose of this change! However, the newly started registry is not monitoring those buckets, as they were created as part of previous, now defunct, registry. This means that the table may go stale, because we won't remove those buckets if they die.

Let's add a test to `test/kv/registry_test.exs` that shows this bug:

```
st "monitors existing entries", %{registry: registry, ets: ets} do
bucket = KV.Registry.create(registry, "shopping")

# Kill the registry. We unlink first, otherwise it will kill the test
Process.unlink(registry)
Process.exit(registry, :shutdown)

# Start a new registry with the existing table and access the bucket
start_registry(ets)
assert KV.Registry.lookup(ets, "shopping") == {:ok, bucket}

# Once the bucket dies, we should receive notifications
Process.exit(bucket, :shutdown)
assert_receive {:exit, "shopping", ^bucket}
assert KV.Registry.lookup(ets, "shopping") == :error
d
```

Run the new test and it will fail with:

```
 test monitors existing entries (KV.RegistryTest)
 test/kv/registry_test.exs:72
 No message matching {:exit, "shopping", ^bucket}
 stacktrace:
   test/kv/registry_test.exs:85
```

That's what we expected. If the bucket is not being monitored, the registry is not notified when it dies and therefore no event is sent. We can fix this by

changing `KV.Registry`'s `init/1` callback one last time to setup monitors for all existing entries in the table:

```
f init({table, events, buckets}) do
refs = :ets.foldl(fn {name, pid}, acc ->
  HashDict.put(acc, Process.monitor(pid), name)
end, HashDict.new, table)

{:ok, %{names: table, refs: refs, events: events, buckets: buckets}}
d
```

We use `:ets.foldl/3` to go through all entries in the table, similar to `Enum.reduce/3`, invoking the given function for each element in the table with the given accumulator. In the function callback, we monitor each pid in the table and update the refs dictionary accordingly. If any of the entries is already dead, we will still receive the `:DOWN` message, causing them to be purged later.

In this chapter we were able to make our application more robust by using an ETS table that is owned by the supervisor and passed to the registry. We have also explored how to use ETS as a cache and discussed some of the race conditions we may run into as data becomes shared between the server and all clients.

7 Dependencies and umbrella projects

In this chapter, we will briefly discuss how to manage dependencies in Mix.

Our `kv` application is complete, so it's time to implement the server that will handle the requests we defined in the first chapter:

```
EATE shopping


T shopping milk 1


T shopping eggs 3


T shopping milk



LETE shopping eggs

```

However, instead of adding more code to the `kv` application, we are going to build the TCP server as another application that is a client of the `kv` application. Since the whole runtime and Elixir ecosystem are geared towards applications, it makes sense to break our projects into smaller applications that work together rather than building a big, monolithic app.

Before creating our new application, we must discuss how Mix handles dependencies. In practice, there are two kinds of dependencies we usually work with: internal and external dependencies. Mix supports mechanisms to work with both of them.

7.1 External dependencies

External dependencies are the ones not tied to your business domain. For example, if you need a HTTP API for your distributed KV application, you can use the Plug project as an external dependency.

Installing external dependencies is simple. Most commonly, we use the Hex Package Manager, by listing the dependency inside the deps function in our `mix.exs` file:

```
f deps do
[{:plug, "~> 0.5.0"}]
d
```

This dependency refers to the latest version of plug in the 0.5.x version series that has been pushed to Hex. This is indicated by the `~>` preceding the version number. For more information on specifying version requirements, see the documentation for the Version module.

Typically, stable releases are pushed to Hex. If you want to depend on an external dependency still in development, Mix is able to manage git dependencies, too:

```
f deps do
[{:plug, git: "git://github.com/elixir-lang/plug.git"}]
d
```

You will notice that when you add a dependency to your project, Mix generates a `mix.lock` file that guarantees *repeatable builds*. The lock file must be checked in to your version control system, to guarantee that everyone who uses the project will use the same dependency versions as you.

Mix provides many tasks for working with dependencies, which can be seen in `mix help`:

```
$ mix help
mix deps              # List dependencies and their status
```

```
mix deps.clean        # Remove the given dependencies' files
mix deps.compile      # Compile dependencies
mix deps.get          # Get all out of date dependencies
mix deps.unlock       # Unlock the given dependencies
mix deps.update       # Update the given dependencies
```

The most common tasks are `mix deps.get` and `mix deps.update`. Once fetched, dependecies are automatically compiled for you. You can read more about deps by typing `mix help deps`, and in the documentation for the Mix.Tasks.Deps module.

7.2 Internal dependencies

Internal dependencies are the ones that are specific to your project. They usually don't make sense outside the scope of your project/company/organization. Most of the time, you want to keep them private, whether due to technical, economic or business reasons.

If you have an internal dependency, Mix supports two method of working with them: git repositories or umbrella projects.

For example, if you push the `kv` project to a git repository, you just need to list it in your deps code in order to use it:

```
1 f deps do
2 [{:kv, git: "git://github.com/YOUR_ACCOUNT/kv.git"}]
3 d
```

It doesn't matter if the git repository is public or private, Mix will be able to fetch it for you as long as you have the proper credentials.

However, using git dependencies for internal dependencies is somewhat discouraged in Elixir. Remember that the runtime and the Elixir ecosystem already provides the concept of applications, and as such we expect you to frequently break your code into applications that can be organized logically, even within a single project.

However, if you push every application as a separate project to a git repository, your projects can become very hard to maintain, because now you will have to spend a lot of time managing those git repositories rather than writing your code.

For this reason, Mix supports "umbrella projects." Umbrella projects allow you to create one project that hosts many applications and push all of them to a single git repository. That is exactly the style we are going to explore in the next sections.

What we are going to do is create a new mix project. We are going to creatively name it `kv_umbrella`, and this new project will have both the existing `kv` application and the new `kv_server` application inside. The directory structure will look like this:

```
+ kv_umbrella
  + apps
    + kv
    + kv_server
```

The interesting thing about this approach is that Mix has many conveniences for working with such projects, such as the ability to compile and test all applications inside `apps` with a single command. However, even though they are all listed together inside `apps`, they are still decoupled from each other, so you can build, test and deploy each application in isolation if you want to.

So let's get started!

7.3 Umbrella projects

Let's start a new project using `mix new`. This new project will be named `kv_umbrella` and we need to pass the `--umbrella` option when creating it. Do not create this new project inside the existing `kv` project!

```
$ mix new kv_umbrella --umbrella
* creating .gitignore
* creating README.md
* creating mix.exs
* creating apps
* creating config
* creating config/config.exs
```

From the printed information, we can see far fewer files are generated. The generated `mix.exs` file is different too. Let's take a look (comments have been removed):

```
1  fmodule KvUmbrella.Mixfile do
2  use Mix.Project
3
4  def project do
5    [apps_path: "apps",
6     deps: deps]
7  end
8
9  defp deps do
10   []
11 end
12 d
```

What makes this project different from the previous one is simply the `apps_path: "apps"` entry in the project definition. This means this project will act as an

umbrella. Such projects do not have source files nor tests, although they can have dependencies which are only available for themselves. We'll create new application projects inside the apps directory. We call these applications "umbrella children".

Let's move inside the apps directory and start building `kv_server`. This time, we are going to pass the `--sup` flag, which will tell Mix to generate a supervision tree automatically for us, instead of building one manually as we did in previous chapters:

```
$ cd kv_umbrella/apps
$ mix new kv_server --module KVServer --sup
```

The generated files are similar to the ones we first generated for `kv`, with a few differences. Let's open up `mix.exs`:

```
1  fmodule KVServer.Mixfile do
2  use Mix.Project
3
4  def project do
5    [app: :kv_server,
6     version: "0.0.1",
7     deps_path: "../../deps",
8     lockfile: "../../mix.lock",
9     elixir: "~> 0.14.1-dev",
10    deps: deps]
11 end
12
13 def application do
14   [applications: [],
15    mod: {KVServer, []}]
16 end
17
18 defp deps do
19   []
20 end
21 d
```

First of all, since we generated this project inside `kv_umbrella/apps`, Mix automatically detected the umbrella structure and added two lines to the project definition:

```
1  ps_path: "../../deps",
2  ckfile: "../../mix.lock",
```

Those options mean all dependencies will be checked out to `kv_umbrella/deps`, and they will share the same lock file. Those two lines are saying that if two applications in the umbrella share the same dependency, they won't be fetched twice. They'll be fetched once, and Mix will ensure that both apps are always running against the same version of their shared dependency.

The second change is in the `application` function inside `mix.exs`:

```
f application do
[applications: [],
 mod: {KVServer, []}]
d
```

Because we passed the `--sup` flag, Mix automatically added `mod: {KVServer, []}`, specifying that `KVServer` is our application callback module. `KVServer` will start our application supervision tree.

In fact, let's open up `lib/kv_server.ex`:

```
fmodule KVServer do
use Application

def start(_type, _args) do
  import Supervisor.Spec, warn: false

  children = [
    # worker(KVServer.Worker, [arg1, arg2, arg3])
  ]

  opts = [strategy: :one_for_one, name: KVServer.Supervisor]
  Supervisor.start_link(children, opts)
end
d
```

Notice that it defines the application callback function, `start/2`, and instead of defining a supervisor named `KVServer.Supervisor` that uses the `Supervisor` module, it conveniently defined the supervisor inline! You can read more about such supervisors by reading the Supervisor module documentation.

We can already try out our first umbrella child. We could run tests inside the `apps/kv_server` directory, but that wouldn't be much fun. Instead, go to the root of the umbrella project and run `mix test`:

```
$ mix test
```

And it works!

Since we want `kv_server` to eventually use the functionality we defined in `kv`, we need to add `kv` as a dependency to our application.

7.4 In umbrella dependencies

Mix supports an easy mechanism to make one umbrella child depend on another. Open up `apps/kv_server/mix.exs` and change the `deps/0` function to the following:

```
fp deps do
[{:kv, in_umbrella: true}]
d
```

The line above makes `:kv` available as a dependency inside `:kv_server`. We can invoke the modules defined in `:kv` but it does not automatically start the `:kv` application. For that, we also need to list `:kv` as an application inside `application/0`:

```
f application do
[applications: [:kv],
 mod: {KVServer, []}]
d
```

Now Mix will guarantee the `:kv` application is started before `:kv_server` is started.

Finally, copy the `kv` application we have built so far to the `apps` directory in our new umbrella project. The final directory structure should match the structure we mentioned earlier:

```
+ kv_umbrella
  + apps
    + kv
    + kv_server
```

We now just need to modify `apps/kv/mix.exs` to contain the umbrella entries we have seen in `apps/kv_server/mix.exs`. Open up `apps/kv/mix.exs` and add to the `project` function:

```
ps_path: "../../deps",
ckfile: "../../mix.lock",
```

Now you can run tests for both projects from the umbrella root with `mix test`. Sweet!

Remember that umbrella projects are a convenience to help you organize and manage your applications. Applications inside the `apps` directory are still decoupled from each other. Each application has its independent configuration, and dependencies in between them must be explicitly listed. This allows them to be developed together, but compiled, tested and deployed independently if desired.

7.5 Summing up

In this chapter we have learned more about Mix dependencies and umbrella projects. We have decided to build an umbrella project because we consider `kv` and `kv_server` to be internal dependencies that matter only in the context of this project.

In the future, you are going to write applications and you will notice they can be easily extracted into a concise unit that can be used by different projects. In such cases, using Git or Hex dependencies is the way to go.

Here are a couple questions you can ask yourself when working with dependencies. Start with: does this application makes sense outside this project?

- If no, use an umbrella project with umbrella children.
- If yes, can this project be shared outside your company / organization?
- If no, use a private git repository.
- If yes, push your code to a git repository and do frequent releases using Hex.

With our umbrella project up and running, it is time to start writing our server.

8 Task and gen_tcp

In this chapter, we are going to learn how to use Erlang's `:gen_tcp` module to serve requests. In future chapters we will expand our server so it can actually serve the commands. This will also provide a great opportunity to explore Elixir's tasks.

8.1 Echo server

We will start our TCP server by first implementing an echo server, that sends as response the same text it received in the request. We will slowly improve our server until it is supervised and ready to handle multiple connections.

A TCP server, in broad strokes, does the following steps:

1. Listens to a port until the port is available and it gets hold of the socket
2. It then waits for a client connection on that port and accepts it
3. It reads the client request and writes a response back

Let's implement those steps. Move to the `apps/kv_server` application and open up `lib/kv_server.ex` and add the following funtions:

```
f accept(port) do
# The options below mean:
#
```

```
# 1. ‘:binary‘ - receives data as binaries (instead of lists)
# 2. ‘packet: :line‘ - receives data by line by line
# 3. ‘active: false‘ - block on ‘:gen_tcp.recv/2‘ until data is available
#
{:ok, socket} = :gen_tcp.listen(port,
                  [:binary, packet: :line, active: false])
IO.puts "Accepting connections on port #{port}"
loop_acceptor(socket)
d

fp loop_acceptor(socket) do
{:ok, client} = :gen_tcp.accept(socket)
serve(client)
loop_acceptor(socket)
d

fp serve(socket) do
socket
|> read_line()
|> write_line(socket)

serve(socket)
d

fp read_line(socket) do
{:ok, data} = :gen_tcp.recv(socket, 0)
data
d

fp write_line(line, socket) do
:gen_tcp.send(socket, line)
d
```

We are going to start our server by calling `KVServer.accept(4040)` where 4040 is the port. The first step in `accept/1` is to listen the port until the socket becomes available and then call `loop_acceptor/1` which is a loop accepting client connections. Once a connection is accepted, we call `serve/1`.

`serve/1` is another loop that reads a line the socket and then those lines back to the socket. Note the `serve/1` function uses the pipeline operator |> to express this flow of operation. The pipeline operator evaluates the left side and pass its result as first argument to the function on the right side. The example above:

```
cket |> read_line() |> write_line(socket)
```

is equivalent to:

```
ite_line(read_line(socket), socket)
```

> When using the |> operator, it is important to add parentheses to the function calls due to how operator precedence work. In particular, this code:
>
> ```
> 1..10 |> Enum.filter &(&1 <= 5) |> Enum.map &(&1 * 2)
> ```
>
> Actually translates to:
>
> ```
> 1..10 |> Enum.filter(&(&1 <= 5) |> Enum.map(&(&1 * 2)))
> ```
>
> Which is not what we want, since the function given to `Enum.filter/2` is the one passed as first argument to `Enum.map/2`. The solution is to use explicit parentheses:
>
> ```
> 1..10 |> Enum.filter(&(&1 <= 5)) |> Enum.map(&(&1 * 2))
> ```

The `read_line/1` implementation receives data from the socket using `:gen_tcp.recv/2` and `write_line/2` writes to the socket using `:gen_tcp.send/2`.

This is pretty much all we need to implement our echo server. Let's give it a try!

Start an iex session inside the `kv_server` application with `iex -S mix` and inside IEx run:

```
x> KVServer.accept(4040)
```

The server is now running, you will even notice the console is blocked. Let's use a `telnet` client to access our server. There are clients available to most operating systems and their command line instructions are quite similar:

```
$ telnet 127.0.0.1 4040
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello
hello
is it me
is it me
you are looking for?
you are looking for?
```

Type "hello", press enter, and you will get "hello" back. Excellent!

My particular telnet client can be exited by typing `ctrl + ]` and then typing quit and pressing enter but your client may require different steps.

Once you kill the telnet client, you will likely get an error back in the IEx session:

```
** (MatchError) no match of right hand side value: {:error, :closed}
    (kv_server) lib/kv_server.ex:41: KVServer.read_line/1
    (kv_server) lib/kv_server.ex:33: KVServer.serve/1
    (kv_server) lib/kv_server.ex:27: KVServer.loop_acceptor/1
```

That's because we were expecting data from `:gen_tcp.recv/2` but the client closed the connection. We need to handle such cases better in upcoming versions.

For now there is a more important bug we need to fix: what happens if our TCP acceptor crashes? Since there is no supervision, the server dies and we won't be able to serve more requests as it won't be restarted. That's why we must move our server inside a supervision tree.

8.2 Tasks

We have learned about agents, generic server, event managers and they were all meant to work with multiple messages or work around state. But what do we use when we only need to execute some task and that is it?

The Task module provides the exact functionality for this use case. For example, there is a function named `start_link/3` that receives the module, function and arguments, allowing us to run a given function as part of a supervision tree.

Let's give it a try. Open up `lib/kv_server.ex` and let's change the supervisor directly in the `start/2` function to the following:

```
1  f start(_type, _args) do
2  import Supervisor.Spec
3
4  children = [
5    worker(Task, [KVServer, :accept, [4040]])
6  ]
7
8  opts = [strategy: :one_for_one, name: KVServer.Supervisor]
9  Supervisor.start_link(children, opts)
10 d
```

We are basically saying we want to run `KVServer.accept(4040)` as a worker. We are hardcoding the port for now but we will get back to it later.

Now the server is part of the supervision tree and it should start by simply running the application. Type `mix run --no-halt` in the terminal and then once again use the `telnet` client to guarantee everything works:

```
$ telnet 127.0.0.1 4040
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
say you
say you
say me
say me
```

Yes, it works! If you kill the client, causing the whole server to crash, you will see another one will start right away. However, does it *scale*?

Try to connect two telnet clients at the same time. When you do so, you will see the second client won't echo:

```
$ telnet 127.0.0.1 4040
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello
hello?
HELLOOOOOO?
```

It doesn't seem to work at all. That's because we are serving requests in the same process that are accepting connections. So when one client is connected, we cannot accept anyone else.

8.3 Task supervisor

In order to make our server handle multiple connections at the same time, we need to have one process working as an acceptor that spawns other processes to serve requests. One solution would be to change:

```
1 fp loop_acceptor(socket) do
2 {:ok, client} = :gen_tcp.accept(socket)
3 serve(client)
4 loop_acceptor(socket)
5 d
```

to use `Task.start_link/1` (similar to `Task.spawn_link/3` but it receives an anonymous function instead of module, function and arguments):

```
fp loop_acceptor(socket) do
{:ok, client} = :gen_tcp.accept(socket)
Task.start_link(fn -> serve(client) end)
loop_acceptor(socket)
d
```

But we have made this mistake in the past, do you remember?

This is similar to the mistake we have made when we called `KV.Bucket.start_link/0` from the registry. That meant a failure in any bucket would bring the whole registry down.

The code above would have the exact same flaw: if we link the `serve(client)` task to the acceptor, a crash when serving a request would bring the whole acceptor, and consequently, all other connections down.

We fixed the issue for the registry by using a simple one for one supervisor. We are going to use the same solution here, except that this pattern is so common with tasks, that tasks already come with a simple one for one supervisor with temporary workers that we can just use in our supervision tree!

Let's change `start/2` once again to add a supervisor to our tree:

```
f start(_type, _args) do
import Supervisor.Spec

children = [
  supervisor(Task.Supervisor, [[name: KVServer.TaskSupervisor]]),
  worker(Task, [KVServer, :accept, [4040]])
]

opts = [strategy: :one_for_one, name: KVServer.Supervisor]
Supervisor.start_link(children, opts)
d
```

We simply start a `Task.Supervisor` process with name `KVServer.TaskSupervisor`. Remember that, since the acceptor task depends on this supervisor, the supervisor must be started first.

Now we just need to change `loop_acceptor/2` to use `Task.Supervisor` to serve each request:

```
fp loop_acceptor(socket) do
{:ok, client} = :gen_tcp.accept(socket)
Task.Supervisor.start_child(KVServer.TaskSupervisor, fn -> serve(client) end)
loop_acceptor(socket)
d
```

Start a new server with `mix run --no-halt` and we can now open up many concurrent telnet clients. You will also notice that quitting a client does not bring the acceptor down. Excellent!

Here is the full echo server implementation, in a single module:

```
fmodule KVServer do
use Application

@doc false
def start(_type, _args) do
  import Supervisor.Spec

  children = [
    supervisor(Task.Supervisor, [[name: KVServer.TaskSupervisor]]),
    worker(Task, [KVServer, :accept, [4040]])
  ]

  opts = [strategy: :one_for_one, name: KVServer.Supervisor]
  Supervisor.start_link(children, opts)
end

@doc """
Starts accepting connections on the given `port`.
"""
def accept(port) do
  {:ok, socket} = :gen_tcp.listen(port,
                    [:binary, packet: :line, active: false])
  IO.puts "Accepting connections on port #{port}"
  loop_acceptor(socket)
end

defp loop_acceptor(socket) do
  {:ok, client} = :gen_tcp.accept(socket)
  Task.Supervisor.start_child(KVServer.TaskSupervisor, fn -> serve(client) end)
  loop_acceptor(socket)
end

defp serve(socket) do
  socket
  |> read_line()
  |> write_line(socket)

  serve(socket)
end

```

```
defp read_line(socket) do
  {:ok, data} = :gen_tcp.recv(socket, 0)
  data
end

defp write_line(line, socket) do
  :gen_tcp.send(socket, line)
end
d
```

Since we have changed the supervisor specification, we need to ask: is our supervision strategy is still correct?

In this case, the answer is yes: if the acceptor crashes, there is no need to crash the existing connections. On the other hand, if the task supervisor crashes, there is no need to crash the acceptor too. This is a contrast to the registry, where we initially had to crash the supervisor every time the registry crashed, until we used ETS to persist state. However tasks have no state and nothing will go stale if one of those process die.

In the next chapter we will start parsing the client requests and sending responses, finishing our server.

9 Docs, tests and pipelines

In this chapter, we will implement the code that parses the commands we have first described in the first chapter:

```
EATE shopping


T shopping milk 1


T shopping eggs 3


T shopping milk


LETE shopping eggs

```

After the parsing is done, we will update our server to dispatch the parsed commands to the `:kv` application we have built in previous chapters.

9.1 Doctests

On the language homepage, we mention how Elixir makes documentation a first-class citizen in the language. We have explored this chapter many times throughout this guide, be it via `mix help` or by typing `h Enum` or another module in the terminal.

In this section, we will implement the parse functionality using doctests, which allows us to write tests directly from our documentation, which helps provide documentation with accurate code samples.

Let's create our command parser at `lib/kv_server/command.ex` and start with the doctest:

```
1 fmodule KVServer.Command do
2 @doc ~S"""
3 Parses the given `line` into a command.
4
5 ## Examples
6
7     iex> KVServer.Command.parse "CREATE shopping\r\n"
8     {:ok, {:create, "shopping"}}
9
10 """
11 def parse(line) do
12   :not_implemented
13 end
14 d
```

Doctests are specified by four spaces in the documentation followed by the `iex>` prompt. If a command spawns multiple lines, you can use `...>`, as in IEx. The expected result should start at the next line after `iex>` or `...>` line(s) and is terminated either by a newline or a new `iex>` prefix.

Also note that we started the documentation string using `@doc ~S"""`. We have used `~S` so the `\r\n` characters inside the doctest are preserved as is.

In order to run our doctests, create a file at `test/kv_server/command_test.exs` and simply call `doctest KVServer.Command` in the test case:

```
1 fmodule KVServer.CommandTest do
2 use ExUnit.Case, async: true
3 doctest KVServer.Command
4 d
```

Run the test suite and the doctest should fail:

```
1) test doc at KVServer.Command.parse/1 (1) (KVServer.CommandTest)
```

```
test/kv_server/command_test.exs:3
Doctest failed
code: KVServer.Command.parse "CREATE shopping\r\n" === {:ok, {:create, "shopping"}}
lhs:  :not_implemented
stacktrace:
  lib/kv_server/command.ex:11: KVServer.Command (module)
```

Excellent!

Now it is just a matter of making the doctest pass. Let's implement the `parse/1` function:

```
f parse(line) do
case String.split(line) do
  ["CREATE", bucket] -> {:ok, {:create, bucket}}
end
d
```

Our implementation simply splits the line on every whitespace and then matches the command against a list. Using `String.split/2` means our commands are actually white-space insensitive. Let's add some new doctests to test this behaviour and the other commands:

```
oc ~S"""
rses the given ‘line‘ into a command.

 Examples

  iex> KVServer.Command.parse "CREATE shopping\r\n"
  {:ok, {:create, "shopping"}}

  iex> KVServer.Command.parse "CREATE  shopping  \r\n"
  {:ok, {:create, "shopping"}}

  iex> KVServer.Command.parse "PUT shopping milk 1\r\n"
  {:ok, {:put, "shopping", "milk", "1"}}

  iex> KVServer.Command.parse "GET shopping milk\r\n"
  {:ok, {:get, "shopping", "milk"}}

  iex> KVServer.Command.parse "DELETE shopping eggs\r\n"
  {:ok, {:delete, "shopping", "eggs"}}

known commands or commands with the wrong number of
guments return an error:
```

```
23
24    iex> KVServer.Command.parse "UNKNOWN shopping eggs\r\n"
25    {:error, :unknown_command}
26
27    iex> KVServer.Command.parse "GET shopping\r\n"
28    {:error, :unknown_command}
29
30  "
```

With doctests is hand, it is your turn to make tests pass! Once you ready, you can compare with our solution below:

```
1 f parse(line) do
2 case String.split(line) do
3   ["CREATE", bucket] -> {:ok, {:create, bucket}}
4   ["GET", bucket, key] -> {:ok, {:get, bucket, key}}
5   ["PUT", bucket, key, value] -> {:ok, {:put, bucket, key, value}}
6   ["DELETE", bucket, key] -> {:ok, {:delete, bucket, key}}
7   _ -> {:error, :unknown_command}
8 end
9 d
```

Notice how we were able to elegantly parse the commands without adding a bunch of `if/else` clauses that checks the command name and length!

Finally, you may have observed that each doctest was considered to be a different test in our test case, as our test suite now reports a total of 7 tests. That is because ExUnit considers this two different tests:

```
1 x> KVServer.Command.parse "UNKNOWN shopping eggs\r\n"
2 error, :unknown_command}
3
4 x> KVServer.Command.parse "GET shopping\r\n"
5 error, :unknown_command}
```

But without new lines, like the one below, ExUnit compiles it into a single test:

```
1 x> KVServer.Command.parse "UNKNOWN shopping eggs\r\n"
2 error, :unknown_command}
3 x> KVServer.Command.parse "GET shopping\r\n"
4 error, :unknown_command}
```

You can read more about doctests in the `ExUnit.DocTest` docs.

9.2 Pipelines

With our command parser in hand, we can finally start implementing the logic that runs the commands. Let's add a stub definition for this function for now:

```
fmodule KVServer.Command do
@doc """
Runs the given command.
"""
def run(command) do
  {:ok, "OK\r\n"}
end
d
```

Before we implement this function, let's change our server to start using both `parse/1` and `run/1` functions. Remember our `read_line/1` function was also crashing when the client closed the socket, so let's take the opportunity to fix it too. Open up `lib/kv_server.ex` and replace the existing server definition:

```
fp serve(socket) do
socket
|> read_line()
|> write_line(socket)

serve(socket)
d

fp read_line(socket) do
{:ok, data} = :gen_tcp.recv(socket, 0)
data
d

fp write_line(line, socket) do
:gen_tcp.send(socket, line)
d
```

by the following:

```
fp serve(socket) do
msg =
  case read_line(socket) do
    {:ok, data} ->
      case KVServer.Command.parse(data) do
```

```
        {:ok, command} ->
          KVServer.Command.run(command)
        {:error, _} = err ->
          err
      end
    {:error, _} = err ->
      err
  end

write_line(socket, msg)
serve(socket)
d

fp read_line(socket) do
:gen_tcp.recv(socket, 0)
d

fp write_line(socket, msg) do
:gen_tcp.send(socket, format_msg(msg))
d

fp format_msg({:ok, text}), do: text
fp format_msg({:error, :unknown_command}), do: "UNKNOWN COMMAND\r\n"
fp format_msg({:error, _}), do: "ERROR\r\n"
```

If we start our server, we can now send commands to it. For now we can get two different responses, "OK" when the command is known and "UNKNOWN COMMAND" otherwise:

```
$ telnet 127.0.0.1 4040
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
CREATE shopping
OK
HELLO
UNKNOWN COMMAND
```

This means our implementation is going in the correct direction, but it doesn't look very elegant, does it?

The previous implementation used pipes which made the logic straightforward to understand:

```
ad_line(socket) |> KVServer.Command.parse |> KVServer.Command.run()
```

Since we may have failures along the way, we need our pipeline logic to match error outputs and abort if they occur. Wouldn't it be great if instead we could say: "pipe those functions while the response is ok" or "pipe those functions while the response matches the {:ok, _} tuple"?

Actually there is a project called elixir-pipes that provides exactly this functionality! Let's give it a try. Open up your `apps/kv_server/mix.exs` file and change both `application/0` and `deps/0` functions to the following:

```
f application do
[applications: [:pipe, :kv],
 mod: {KVServer, []}]
d

fp deps do
[{:kv, in_umbrella: true},
 {:pipe, github: "batate/elixir-pipes"}]
d
```

Run `mix deps.get` to get the dependency and rewrite `serve/1` function to use the `pipe_matching/3` functionality from the elixir-pipes project:

```
fp serve(socket) do
import Pipe

msg =
  pipe_matching x, {:ok, x},
    read_line(socket)
    |> KVServer.Command.parse()
    |> KVServer.Command.run()

write_line(socket, msg)
serve(socket)
d
```

With `pipe_matching/3` we can ask Elixir to pipe the value `x` from each step if it matches {:ok, x}. We do so by basically converting each expression given to `case/2` as a step in the pipeline. As soon as any of the steps return something that does not match {:ok, x}, the pipeline aborts, and returns the non-matching value.

Excellent! Feel free to read the elixir-pipes project documentation to learn about other options for expressing pipelines. Let's continue moving forward with our server implementation.

9.3 Running commands

The last step is to implement `KVServer.Command.run/1` to run the parsed commands against the `:kv` application. Its implementation is shown below:

```
oc """
ns the given command.
"
f run(command)

f run({:create, bucket}) do
KV.Registry.create(KV.Registry, bucket)
{:ok, "OK\r\n"}
d

f run({:get, bucket, key}) do
lookup bucket, fn pid ->
  value = KV.Bucket.get(pid, key)
  {:ok, "#{value}\r\nOK\r\n"}
end
d

f run({:put, bucket, key, value}) do
lookup bucket, fn pid ->
  KV.Bucket.put(pid, key, value)
  {:ok, "OK\r\n"}
end
d

f run({:delete, bucket, key}) do
lookup bucket, fn pid ->
  KV.Bucket.delete(pid, key)
  {:ok, "OK\r\n"}
end
d

fp lookup(bucket, callback) do
case KV.Registry.lookup(KV.Registry, bucket) do
  {:ok, pid} -> callback.(pid)
  :error -> {:error, :not_found}
end
d
```

The implementation is quite-straight forward: we just dispatch to the `KV.Registry` server registered during the `:kv` application startup.

Note we have also defined a private function named `lookup/2` to help with the common functionality of looking up a bucket and returning its `pid` if it exists, {`:error, :not_found`} otherwise.

By the way, since we are now returning {`:error, :not_found`}, we should amend the `format_msg/1` function in `KV.Server` to nicely show not found messages too:

```
fp format_msg({:ok, text}), do: text
fp format_msg({:error, :unknown_command}), do: "UNKNOWN COMMAND\r\n"
fp format_msg({:error, :not_found}), do: "NOT FOUND\r\n"
fp format_msg({:error, _}), do: "ERROR\r\n"
```

And our server functionality is almost complete, we just need to add tests. This time we have left tests for last because there are some important considerations to be done.

`KVServer.Command.run/1`'s implementation is sending commands directly to the server named `KV.Registry`, which is registered by the `:kv` application. This means this server is global and if we have two tests sending messages to it at the same time, our tests will conflict with each other (and likely fail). We need to decide in between having unit tests, that are isolated and can run asynchronously, or write integration tests, that work on top of the global state, but exercises our application full stack as it is meant to be exercised in production.

So far we have been chosing the unit test approach. For example, in order to make `KVServer.Command.run/1` testable as a unit we would need to change its implementation to not send commands directly to the `KV.Registry` process but instead pass a server as argument. This means we would need to change `run`'s signature to `def run(command, pid)` and the implementation for the `:create` command would look like:

```
f run({:create, bucket}, pid) do
KV.Registry.create(pid, bucket)
{:ok, "OK\r\n"}
d
```

Then in `KVServer.Command`'s test case, we need to start an instance of the `KV.Registry`, similar to how we have done in `apps/kv/test/kv/registry_test.exs` and pass it as argument to `run/2`.

This has been the approach we have done so far in our tests and it brings some benefits:

1. Our implementation is not coupled to any particular server name
2. We can keep our tests running asynchronously as there is no shared state

However it comes with the downside that our APIs become increasingly larger in order to receive all external parameters.

The alternative is to continue relying on the global server names and run tests against the global data, ensuring we clean up the data in between the tests. In this case, since the test would exercise the whole stack, from the TCP server, to the command parser and running, to the registry and finally reaching the bucket, it becomes an integration test.

The downside of integration tests though is that they can be much slower than unit tests and as such they must be used more sparingly. For example, we should not use integration tests to test an edge case in our command parsing implementation.

Since we have used unit tests so far, this time we will take the other road and write an integration test. The integration test will have a TCP client that sends commands to our server and we will assert we are getting the desired responses.

Let's implement our integration test in `test/kv_server_test.exs` as shown below:

```
1  fmodule KVServerTest do
2  use ExUnit.Case
3
4  setup do
5    :application.stop(:kv)
6    :ok = :application.start(:kv)
7  end
8
9  setup do
10   opts = [:binary, packet: :line, active: false]
11   {:ok, socket} = :gen_tcp.connect('localhost', 4040, opts)
12   {:ok, socket: socket}
13 end
14
15 test "server interaction", %{socket: socket} do
16   assert send_and_recv(socket, "UNKNOWN shopping\r\n") ==
17          "UNKNOWN COMMAND\r\n"
18
19   assert send_and_recv(socket, "GET shopping eggs\r\n") ==
20          "NOT FOUND\r\n"
21
22   assert send_and_recv(socket, "CREATE shopping\r\n") ==
23          "OK\r\n"
24
25   assert send_and_recv(socket, "PUT shopping eggs 3\r\n") ==
26          "OK\r\n"
27
28   # GET returns two lines
```

```
  assert send_and_recv(socket, "GET shopping eggs\r\n") == "3\r\n"
  assert send_and_recv(socket, "") == "OK\r\n"

  assert send_and_recv(socket, "DELETE shopping eggs\r\n") ==
         "OK\r\n"

  # GET returns two lines
  assert send_and_recv(socket, "GET shopping eggs\r\n") == "\r\n"
  assert send_and_recv(socket, "") == "OK\r\n"
end

defp send_and_recv(socket, command) do
  :ok = :gen_tcp.send(socket, command)
  {:ok, data} = :gen_tcp.recv(socket, 0, 1000)
  data
end
d
```

Our integration test checks the whole server interaction, including unknown commands and not found errors. It is worthy reminding that, as in ETS tables and linked processes, there is no need to close the socket because once the test process exits, the socket is automatically closed.

This time, since our test relies on global data, we have not given `async: true` to `use ExUnit.Case`. Furthermore, in order to guarantee our test is always in a clean slate, we stop and start the `:kv` application before each test. In fact, stopping the `:kv` application even prints a warning on the terminal:

```
NFO REPORT==== 14-Jun-2014::13:40:21 ===
  application: kv
  exited: stopped
  type: temporary
```

If desired, we can avoid such data being printed by turning the error_logger off and on in the test setup:

```
tup do
:error_logger.tty(false)
:application.stop(:kv)
:ok = :application.start(:kv)
:error_logger.tty(true)
:ok
d
```

With this simple integration test, we start to see why integration tests may be slow. Not only this particular test cannot run asynchronously, it also requires the expensive setup of stopping and starting the `:kv` application.

At the end of the day, it is up to you and your team to figure the best testing strategy for your applications. You need to balance the code quality, confidence and time of your test suite. For example, we may start with testing the server only with integration tests, but if the server continuously grows on upcoming releases, or it becomes a part of the application with frequent bugs, it is important to consider breaking it apart and writing more intensive unit tests that don't have the weight of an integration test.

I personally err on the side of unit tests and have integration tests only as smoke tests to guarantee the basic skeleton of the system works.

In the next chapter we will finally make our system distributed by adding a bucket routing mechanism and learn about application configurations.

10 Distributed tasks and configuration

In this last chapter, we will go back to the `:kv` application and add a routing layer that allows us to distribute requests in between nodes based on the bucket name.

The routing layer will receive a routing table of the following format:

```
[{?a..?m, :"foo@computer-name"},
 {?n..?z, :"bar@computer-name"}]
```

The router will check the first byte of the bucket name against the table and dispatch to the appropriate node based on that. For example, a bucket starting with the letter `?a` will be dispatched to node `foo@computer-name`.

In case the matching entry points to the current node itself, we are done. Otherwise the next node will receive the bucket name, look at its own routing table (which may be different from the one in the first node) and act accordingly. In case no entry matches, an error will be raised.

10.1 Our first distributed code

Elixir ships with facilities to connect nodes and exchange information in between them. In fact, we use the same concepts of processes, message passing and receiving messages when working on a distributed environment because we say Elixir processes are *location transparent*. When sending a message, it doesn't matter if the recipient process is on the same node or in another node, the VM will be able to deliver the message in both cases.

In order to run distributed code, we need to start the VM with a name. The name can be short (when in the same network) or long (requires the full computer address). Let's start a new IEx session:

```
$ iex --sname foo
```

You can see now the prompt is slightly different and shows the node name followed by the computer name:

```
Interactive Elixir - press Ctrl+C to exit (type h() ENTER for help)
iex(foo@jv)1>
```

My computer is named `jv`, that's why we see `foo@jv` in the example above, but you should get a different result. We will use `jv@computer-name` in the following examples and you should update them accordingly when trying out the code.

Let's define a module named `HelloWorld` in this shell:

```
x> defmodule Hello do
.>   def world, do: IO.puts "hello world"
.> end
```

If you have another computer in the same network with both Erlang and Elixir installed, you can start another shell on it. If you don't, you can simply start another IEx session in another terminal and give it the short name of `bar`:

```
$ iex --sname bar
```

Note that inside this new IEx session, we cannot access `Hello.world/0`:

```
x> Hello.world
 (UndefinedFunctionError) undefined function: Hello.world/0
  Hello.world()
```

However we can spawn a new process in `foo@computer-name` from `bar@computer-name`! Let's give it a try (where `@computer-name` is the one you see locally):

```
x> Node.spawn_link :"foo@computer-name", fn -> Hello.world end
ID<9014.59.0>
llo world
```

Elixir spawned a process in another node and returned its pid. The code then executed in the other node where the `Hello.world/0` function exists and invoked that function. Note the result of "hello world" was printed in the current node `bar` and not in `foo`. In other words, the message to be printed was sent back from `foo` to `bar`. This happens because the process spawned in the other node (`foo`) still has the group leader of the current node (`bar`). We have briefly talked about group leaders in the IO chapter.

We can send and receive message from the pid returned by `Node.spawn_link/2` as usual. Let's try a quick ping-pong example:

```
x> pid = Node.spawn_link :"foo@computer-name", fn ->
.>   receive do
.>     {:ping, client} -> send client, :pong
.>   end
.> end
ID<9014.59.0>
x> send pid, {:ping, self}
ping, #PID<0.73.0>}
x> flush
ong
k
```

From our quick exploration, we could conclude we could simply use `Node.spawn_link/2` to spawn entries in a remote node every time we need to do a distributed computation. However we have learned throughout the guide that spawning processes outside of supervision trees should be avoided if possible, so we need to look for other options.

There are three better alternatives to `Node.spawn_link/2` we could use in our implementation:

1. We could use Erlang's :rpc module to execute functions in a remote node. Inside the `bar@computer-name` above, you can call `:rpc.call :"foo@computer-name", Hello, :world, []` and it will print "hello world";

2. Another alternative is to have a server running in the other node and send requests to that node via the GenServer API. For example, you can call a remote named server using `GenServer.call({name, node}, arg)` or simply passing the remote process PID as first argument;

3. A third option is to use tasks, which we have learned in the previous chapter, as they support spawning tasks both on local and remote nodes;

The options above have different properties. Both `:rpc` and using a GenServer would serialize your requests into a single server, while tasks are effecticely running in asynchronously in the remote node, with the only serialization point being the spawning done by the supervisor.

For our routing layer, we are going to use tasks, but feel free to explore the other alternatives too.

10.2 async/await

So far we have explored tasks that are started and run in isolation, with no regard to its return value. However sometimes it is useful to run a task to compute a value and read its result later on. For such, tasks also provide the `async/await` pattern:

```
sk = Task.async(fn -> compute_something_expensive end)
s  = compute_something_else()
s + Task.await(task)
```

`async/await` provides a very simple mechanism to compute values concurrently. Not only that, `async/await` can also be used with the same `Task.Supervisor` we have used in previous chapters, we just need to call `Task.Supervisor.async/2` instead of `Task.Supervisor.start_child/2` and use `Task.await/2` to read the result later on.

10.3 Distributed tasks

Distributed tasks are exactly the same as supervised tasks. The only difference is that we pass the node name when spawning the task in the supervisor. Open up `lib/kv/supervisor.ex` from the `:kv` application and let's add a task supervisor to the tree:

```
pervisor(Task.Supervisor, [[name: KV.RouterTasks]]),
```

Now let's start two named nodes again but inside the `:kv` application:

```
$ iex --sname foo -S mix
$ iex --sname bar -S mix
```

From inside `bar@cmputer-name`, we can now spawn a task directly in the other node via the supervisor:

```
x> task = Task.Supervisor.async {KV.RouterTasks, :"foo@computer-name"}, fn ->
.>   {:ok, node()}
.> end
ask{pid: #PID<12467.88.0>, ref: #Reference<0.0.0.400>}
x> Task.await(task)
ok, :"foo@computer-name"}
```

Our first distributed task is quite-straightforward, it simply gets the name of the node the task is running on. With this knowledge in hand, let's finally write the routing code.

10.4 Routing layer

Create a file at `lib/kv/router.ex` with the following contents:

```
fmodule KV.Router do
@doc """
Dispatch the given `mod`, `fun`, `args` request
to the appropriate node according to `bucket`.
"""
def route(bucket, mod, fun, args) do
  # Get the first byte of binary
  first = :binary.first(bucket)

  # Try to find an entry in the table or raise
  entry =
    Enum.find(table, fn {enum, node} ->
      first in enum
    end) || no_entry_error(bucket)

  # If the entry node is the current node
  if elem(entry, 1) == node() do
    apply(mod, fun, args)
  else
    sup = {KV.RouterTasks, elem(entry, 1)}
    Task.Supervisor.async(sup, fn ->
      KV.Router.route(bucket, mod, fun, args)
    end) |> Task.await()
  end
end

defp no_entry_error(bucket) do
  raise "could not find entry for #{inspect bucket} in table #{inspect table}"
end

@doc """
The routing table.
"""
def table do
  # Replace computer-name by your local machine nodes.
  [{?a..?m, :"foo@computer-name"},
   {?n..?z, :"bar@computer-name"}]
end
d
```

Let's write a test to verify our router works. Create a file named `test/kv/router_test.exs` with:

```
fmodule KV.RouterTest do
use ExUnit.Case, async: true

test "route requests accross nodes" do
  assert KV.Router.route("hello", Kernel, :node, []) ==
         :"foo@computer-name"
  assert KV.Router.route("world", Kernel, :node, []) ==
         :"bar@computer-name"
end

test "raises on unknown entries" do
  assert_raise RuntimeError, ~r/could not find entry/, fn ->
    KV.Router.route(<<0>>, Kernel, :node, [])
  end
end
d
```

The first test simply invokes `Kernel.node/0`, which returns the name of the current node, based on the bucket names "hello" and "world". According to our routing table so far, we should get `foo@computer-name` and `bar@computer-name` as responses respectively.

The second test just checks the code raises for unknown entries.

In order to run the first test, we need to have two nodes running. Let's restart the node named `bar` which is going to be used by tests:

```
$ iex --sname bar -S mix
```

And now run tests with:

```
$ elixir --sname foo -S mix test
```

Our test should successfuly pass, excellent!

10.5 Test filters and tags

Although our tests pass, our testing structure is getting more complex. In particular, running tests simply with `mix test` cause failures in our suite since our test requires connection to other nodes.

Luckily, ExUnit ships with a facility to tag tests, allowing us to run specific callbacks or even filter tests altogether based on those tags.

All we need to do to tag a test is simply call `@tag` before the test name. Back to `test/kv/routest_test.exs`, let's add a `:distributed` tag:

```
ag :distributed
st "route requests accross nodes" do
```

`@tag :distributed` is equivalent to `@tag distributed: true`.

With the test properly tagged, we can now check if the node is alive on the network and, if not, we can exclude all distributed tests. Open up `test/test_helper.exs` inside the `:kv` application and add the following:

```
clude =
if Node.alive?, do: [], else: [distributed: true]

Unit.start(exclude: exclude)
```

Now run tests with `mix test`:

```
$ mix test
Excluding tags: [distributed: true]

.......

Finished in 0.1 seconds (0.1s on load, 0.01s on tests)
7 tests, 0 failures
```

This time all tests passed and ExUnit warned us that distributed tests were being excluded. If you run tests with `$ elixir --sname foo -S mix test`, one extra test shuold run and successfully pass as long as the `bar@computer-name` node is available.

The `mix test` command also allows us to dynamically include and exclude tags. For example, we can run `$ mix test --include distributed` to run distributed tests regardless of the value set in `test/test_helper.exs`. We could also pass `--exclude` to exclude a particular tag from the command line. Finally, `--only` can be used to run only tests with a particular tag:

```
$ elixir --sname foo -S mix test --only distributed
```

You can read more about filters, tags and the default tags in `ExUnit.Case` module documentation.

10.6 Application environment and configuration

So far we have hardcoded the routing table into the `KV.Router` module. However, we would like to make the table dynamic so not only we can configure it development/test/production but also allow different nodes to run with different entries in the routing table. There is a convenience in OTP that does exactly that: the application environment.

Each application has an environment that stores the application specific configuration by key. For example, we could store the routing table in the `:kv` application environment, giving it a default value and allowing other applications to change the table as needed.

Open up `apps/kv/mix.exs` and change the `application/0` function to return the following:

```
f application do
[applications: [],
 env: [routing_table: []],
 mod: {KV, []}]
d
```

We have added a new `:env` key to the application, that returns the application default environment which has an entry of key `:routing_table` and value of an empty list. It makes sense for the application environment to ship with an empty table, as the specific of the routing table depends on the testing/deployment structure.

In order to use the application environment in our code, we just need to replace `KV.Router.table/0` by the definition below:

```
oc """
e routing table.
"
f table do
Application.get_env(:kv, :routing_table)
d
```

We use `Application.get_env/2` to read the entry for `:routing_table` in `:kv`'s environment. You can find more information and other functions to manipulate the app environment in the Application module.

Since our routing table is now empty, our distributed test should fail. Restart the apps and re-run tests to see the failure:

```
$ iex --sname bar -S mix
$ elixir --sname foo -S mix test --only distributed
```

The interesting thing about the application environment is that it can be configured, not only for the current application, but for all other applications. Such configuration is done by the `config/config.exs` file. For example, we can configure IEx default prompt to another value. Just open `apps/kv/config/config.exs` and add the following to the end:

```
nfig :iex, default_prompt: ">>>"
```

Start IEx with `iex -S mix` and you can see IEx prompt has changed.

This means we can configure our `:routing_table` directly in the `config/config.exs` file as well:

```
# Replace computer-name by your local machine nodes.
nfig :kv, :routing_table,
     [{?a..?m, :"foo@computer-name"},
      {?n..?z, :"bar@computer-name"}]
```

Restart the nodes and run distributed tests again and now they should all pass.

Each application has their `config/config.exs` file and they are not shared in any way. Configuration can also be set per environment, read the contents of the config file for the `:kv` application for more information on that.

Since config files are not shared, if you run tests from the umbrella root, they will fail because the configuration we have just added to `:kv` is not available there. However, if you open up `config/config.exs` in the umbrella, it has instructions on how to import config files from children applications. You just need to invoke:

```
port_config "../apps/kv/config/config.exs"
```

The `mix run` command also accept a `--config` flag, which allows configuration files to be given on demand. This could be used to start different nodes, each with their specific configuration (for example, different routing tables).

Overall, the built-in ability to configure applications and the fact we have built our software as an umbrella application gives us plenty of options when deploying the software. We can:

- deploy the umbrella application to a node that will work as both TCP server and key-value storage;
- deploy the `:kv_server` application to work only as a TCP server as long as the routing table points only to other nodes;
- deploy only the `:kv` application when we want a node to work only as storage (no TCP access);

Once you add more applications in the future, you can continue controlling your deploy with the same level of granularity, cherry-picking which applications with which configuration are going to production. You can also consider building multiple releases with a tool like exrm, which will package the chosen applications and configuration, including the current Erlang and Elixir installations, so you can deploy the application even if the runtime is not pre-installed in the target system.

Finally, we have learned some new things in this chapter and they could be applied to the `:kv_server` application as well. We are going to leave the next steps as an exercise:

- change the `:kv_server` application to read the port from its application environment instead of using the hardcoded value of 4040;
- change and configure the `:kv_server` application to use the routing functionality instead of dispatching directly to the local `KV.Registry`. For `:kv_server` tests, you can make the routing table simply point to the current node itself;

10.7 Summing up

In this chapter we have built a simple router as a way to explore the distributed features in Elixir and in the Erlang VM and learned how to configure its routing table. This is the last chapter in our Mix and OTP guide.

Throughout the guide, we have built a very simple distributed key-value store as an opportunity to explore many constructs like gen servers, event managers, supervisors, tasks, agents, applications and more. Not only that, we have written tests for the whole application, getting familiar with ExUnit, and learned how to use the Mix build tool to accomplish a wide range of tasks.

In case you are looking for a distributed key-value store to use in production, you should definitely look into Riak which also runs in the Erlang VM. In Riak, the buckets are replicated, to avoid data loss, and instead of a router, they use consistent hashing to map a bucket to a node. A consistent hashing algorithm helps reduce the amount of data that needs to be migrated when new nodes to store buckets are added to your infrastructure.