Table of Contents

Mix and OTP

This file serves as your book's preface, a great place to describe your book's content and ideas.

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*** *(Open Telecom Platform)* 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 v1.2.0 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.
>
> If you have any questions or improvements to the guide, please let us know in our mailing list or issues tracker respectively. Your input is really important to help us guarantee the guides are accessible and up to date!

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 a 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.

Project compilation

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

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

  def project do
    [app: :kv,
     version: "0.1.0",
     elixir: "~> 1.3",
     build_embedded: Mix.env == :prod,
     start_permanent: Mix.env == :prod,
     deps: deps()]
  end

  def application do
    [applications: [:logger]]
  end

  defp deps do
    []
  end
end
```

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:

```
defmodule KV do
end
```

This structure is enough to compile our project:

```
$ cd kv
$ mix compile
```

Will output: Compiling 1 file (.ex) Generated kv app

The `lib/kv.ex` file was compiled, an application manifest named `kv.app` was generated and all protocols were consolidated as described in the Getting Started guide. All compilation artifacts are placed inside the `_build` directory using the options defined in the `mix.exs` file.

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

```
$ iex -S mix
```

Running tests

Mix also generated the appropriate structure for running our project tests. Mix projects usually follow the convention of having a `&lt;filename&gt;_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:

```
defmodule KVTest do
  use ExUnit.Case
  doctest KV

  test "the truth" do
    assert 1 + 1 == 2
  end
end
```

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:

```
ExUnit.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 test, 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 randomizes 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:

```
assert 1 + 1 == 3
```

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

```
1) test the truth (KVTest)
   test/kv_test.exs:5
   Assertion with == failed
   code: 1 + 1 == 3
   lhs:  2
   rhs:  3
   stacktrace:
     test/kv_test.exs:6

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

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 where the test was defined. If you copy the test location in this full second line (including the file and line number) and append it to `mix test`, Mix will load and run just that particular test:

```
$ mix test test/kv_test.exs:5
```

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.

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 run your project in production

The environment applies only to the current project. As we will see later on, any dependency you add to your project will by default run in the `:prod` environment.

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. That's what we have used in both `:build_embedded` and `:start_permanent` options:

```
def project do
  [...,
   build_embedded: Mix.env == :prod,
   start_permanent: Mix.env == :prod,
   ...]
end
```

When you compile your source code, Elixir compiles artifacts to the `_build` directory. However, in many occasions to avoid unnecessary copying, Elixir will create filesystem links from `_build` to actual source files. When true, `:build_embedded` disables this behaviour as it aims to provide everything you need to run your application inside `_build`.

Similarly, when true, the `:start_permanent` option starts your application in permanent mode, which means the Erlang VM will crash if your application's supervision tree shuts down. Notice we don't want this behaviour in dev and test because it is useful to keep the VM instance running in those environments for troubleshooting purposes.

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
```

Or on Windows:

```
> set "MIX_ENV=prod" && mix compile
```

Exploring

There is much more to Mix, and we will continue to explore it as we build our project. A general overview is available on the Mix documentation.

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 task by invoking `mix help TASK`.

Let's write some code!

The trouble with state

Elixir is an immutable language where nothing is shared by default. If we want to provide state, where we create buckets putting and reading values from multiple places, we have two main options in Elixir:

- Processes
- ETS (Erlang Term Storage)

We have already 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, 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 potentially retrieving its result at a later time.

We will explore most 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` .

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 it can be read and modified by other 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 a starting point.

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:

```
iex> {:ok, agent} = Agent.start_link fn -> [] end
{:ok, #PID<0.57.0>}
iex> Agent.update(agent, fn list -> ["eggs" | list] end)
:ok
iex> Agent.get(agent, fn list -> list end)
["eggs"]
iex> Agent.stop(agent)
:ok
```

We started an agent with an initial state of an empty list. We updated the agent's state, adding our new item to the head of the list. The second argument of `Agent.update/3` is a function that takes the agent's current state as input and returns its desired new state. Finally, we retrieved the whole list. The second argument of `Agent.get/3` is a function that takes the state as input and returns the value that `Agent.get/3` itself will return. Once we are done with the agent, we can call `Agent.stop/3` 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:

```
defmodule 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
end
```

Our first test starts 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. This will always work unless the process is named.

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 at the implementation below.

```
defmodule KV.Bucket do
  @doc """
  Starts a new bucket.
"""
  def start_link do
    Agent.start_link(fn -> %{} end)
  end

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

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

We are using a map to store our keys and values. The capture operator, `&amp;`, is introduced in the Getting Started guide.

Now that the `KV.Bucket` module has been defined, our test should pass! You can try it yourself by running: `mix test`.

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:

```
defmodule 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

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

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:

```
test "stores values by key", %{bucket: bucket} do
  # `bucket` is now the bucket from the setup block
end
```

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

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:

```
@doc """
Deletes `key` from `bucket`.

Returns the current value of `key`, if `key` exists.
"""
def delete(bucket, key) do
  Agent.get_and_update(bucket, &Map.pop(&1, key))
end
```

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

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:

```
def delete(bucket, key) do
  Agent.get_and_update(bucket, fn dict->
    Map.pop(dict, key)
  end)
end
```

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:

```
def delete(bucket, key) do
  Process.sleep(1000) # puts client to sleep
  Agent.get_and_update(bucket, fn dict ->
    Process.sleep(1000) # puts server to sleep
    Map.pop(dict, key)
  end)
end
```

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.

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:

```
CREATE shopping
OK

PUT shopping milk 1
OK

GET shopping milk
1
OK
```

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:

```
iex> Agent.start_link(fn -> %{} end, name: :shopping)
{:ok, #PID<0.43.0>}
iex> KV.Bucket.put(:shopping, "milk", 1)
:ok
iex> KV.Bucket.get(:shopping, "milk")
1
```

However, this is a terrible idea! Process 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 create our own *registry process* that holds a map 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 processes. GenServers are the go-to abstraction for building generic servers in both Elixir and *OTP*.

Our first GenServer

A GenServer is implemented in two parts: the client API and the server callbacks, either in a single module or in two different modules implementing client API in one and server callbacks in the other. The client and server run in separate processes, with the client passing messages back and forth to the server as its functions are called. Here we use a single module for both the server callbacks and client API. Create a new file at `lib/kv/registry.ex` with the following contents:

```
defmodule KV.Registry do
  use GenServer

  ## Client API

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

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

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

The first function is `start_link/3`, 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. For now, we pass an empty list

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 to `GenServer.start_link/3` and returns `{:ok, state}`, where state is a new map. 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/2` 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/2` 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.

Testing a GenServer

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:

```
defmodule 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
end
```

Our test should pass right out of the box!

We don't need to explicitly shut down the registry because it will receive a `:shutdown` signal 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, one can use the `GenServer.stop/1` function:

```
## Client API

@doc """
Stops the registry.
"""
def stop(server) do
  GenServer.stop(server)
end
```

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:

```
test "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
end
```

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`:

```
iex> {:ok, pid} = KV.Bucket.start_link
{:ok, #PID<0.66.0>}
iex> Process.monitor(pid)
#Reference<0.0.0.551>
iex> Agent.stop(pid)
:ok
iex> 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/0` all messages and notice a `:DOWN` message arrived, with the exact reference returned by monitor, notifying that 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 -&gt; pid` and another that holds `ref -&gt; 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

def init(:ok) do
  names = %{}
  refs  = %{}
  {:ok, {names, refs}}
end

def handle_call({:lookup, name}, _from, {names, _} = state) do
  {:reply, Map.fetch(names, name), state}
end

def handle_cast({:create, name}, {names, refs}) do
  if Map.has_key?(names, name) do
    {:noreply, {names, refs}}
  else
    {:ok, pid} = KV.Bucket.start_link
    ref = Process.monitor(pid)
    refs = Map.put(refs, ref, name)
    names = Map.put(names, name, pid)
    {:noreply, {names, refs}}
  end
end

def handle_info({:DOWN, ref, :process, _pid, _reason}, {names, refs}) do
  {name, refs} = Map.pop(refs, ref)
  names = Map.delete(names, name)
  {:noreply, {names, refs}}
end

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

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.

`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 didactic 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 registry 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.

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 processes 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 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:

```
{:ok, pid} = KV.Bucket.start_link
ref = Process.monitor(pid)
```

This is a bad idea, as we don't want the registry to crash when a bucket crashes! We typically avoid creating new processes directly, instead we delegate this responsibility to supervisors. As we'll see in the next chapter, supervisors rely on links and that explains why link-based APIs (`spawn_link` , `start_link` , etc) are so prevalent in Elixir and *OTP*.

Supervisor and Application

So far our application has a registry that may monitor dozens, if not hundreds, of buckets. While we think our implementation so far is quite good, no software is bug free, and failures are definitely going to happen.

When things fail, your first reaction may be: "let's rescue those errors". But in Elixir we avoid the defensive programming habit of rescuing exceptions, as commonly seen in other languages. Instead, we say "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 set up 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.

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:

```
defmodule KV.Supervisor do
  use Supervisor

  def start_link do
    Supervisor.start_link(__MODULE__, :ok)
  end

  def init(:ok) do
    children = [
      worker(KV.Registry, [KV.Registry])
    ]

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

Our supervisor has a single child so far: the registry. A worker in the format of:

```
worker(KV.Registry, [KV.Registry])
```

is going to start a process using the following call:

```
KV.Registry.start_link(KV.Registry)
```

The argument we are passing to `start_link` is the name of the process. 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. By using a name, we can guarantee the newly started process will register itself under the same name, without a need to explicitly fetch the latest pid. Notice it is also common to register the process under the same name of the module that defines it, this makes things more straight-forward when debugging or introspecting a live-system.

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, it will be the only one restarted. Since we have only one child now, that's all we need. The `Supervisor` behaviour supports many different strategies and we will discuss them in this chapter.

Since `KV.Registry.start_link/1` is now expecting an argument, we need to change our implementation to receive such argument. Open up `lib/kv/registry.ex` and replace the `start_link/0` definition by:

```
  @doc """
  Starts the registry with the given `name`.
"""
  def start_link(name) do
    GenServer.start_link(__MODULE__, :ok, name: name)
  end
```

We also need to update our tests to give a name when starting the registry. Replace the `setup` function in `test/kv/registry_test.exs` by:

```
  setup context do
    {:ok, registry} = KV.Registry.start_link(context.test)
    {:ok, registry: registry}
  end
```

`setup/2` may also receive the test context, similar to `test/3` . Besides whatever value we may add in our setup blocks, the context includes some default keys, like `:case` , `:test` , `:file` and `:line` . We have used `context.test` as a shortcut to spawn a registry with the same name of the test currently running.

Now with our tests passing, we can take our supervisor for a spin. If we start a console inside our project using `iex -S mix` , we can manually start the supervisor:

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

When we started the supervisor, the registry worker was automatically started, allowing us to create buckets without the need to manually start it.

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

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 a `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:

```
{application,kv,
             [{registered,[]},
              {description,"kv"},
              {applications,[kernel,stdlib,elixir,logger]},
              {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`, `elixir` itself, and `logger` which is specified in the application list in `mix.exs`.

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 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 are going to do our first customization soon.

Starting applications

When we define a `.app` file, which is the application specification, 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 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:

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

Oops, it's already started. Mix normally starts the whole hierarchy of applications defined in our project's `mix.exs` file and it does the same for all dependencies if they depend on other applications.

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` :

```
iex> Application.start(:kv)
:ok
```

We can stop our `:kv` application as well as the `:logger` application, which is started by default with Elixir:

```
iex> Application.stop(:kv)
:ok
iex> Application.stop(:logger)
:ok
```

And let's try to start our application again:

```
iex> Application.start(:kv)
{:error, {:not_started, :logger}}
```

Now we get an error because an application that `:kv` depends on (`:logger` in this case) isn't started. We need to either start each application manually in the correct order or call `Application.ensure_all_started` as follows:

```
iex> Application.ensure_all_started(:kv)
{:ok, [:logger, :kv]}
```

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 `iex -S 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.

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:

```
  def application do
    [applications: [:logger],
     mod: {KV, []}]
  end
```

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`:

```
defmodule KV do
  use Application

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

When we `use Application`, we need to define a couple functions, similar to when we used `Supervisor` or `GenServer`. This time 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.

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

```
iex> KV.Registry.create(KV.Registry, "shopping")
:ok
iex> KV.Registry.lookup(KV.Registry, "shopping")
{:ok, #PID<0.88.0>}
```

How do we know this is working? After all, we are creating the bucket and then looking it up; of course it should work, right? Well, remember that `KV.Registry.create/2` uses `GenServer.cast/3`, and therefore will return `:ok` regardless of whether the message finds its target or not. At that point, we don't know whether the supervisor and the server are up, and if the bucket was created. However, `KV.Registry.lookup/2` uses `GenServer.call/3`, and will block and wait for a response from the server. We do get a positive response, so we know all is up and running.

For an experiment, try reimplementing `KV.Registry.create/2` to use `GenServer.call/3` instead, and momentarily disable the application callback. Run the code above on the console again, and you will see the creation step fail straightaway.

Don't forget to bring the code back to normal before resuming this tutorial!

Projects or applications?

Mix makes a distinction between projects and applications. Based on the 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`.

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
ref = 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 new registry test:

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

    # Stop the bucket with non-normal reason
    Process.exit(bucket, :shutdown)

    # Wait until the bucket is dead
    ref = Process.monitor(bucket)
    assert_receive {:DOWN, ^ref, _, _, _}

    assert KV.Registry.lookup(registry, "shopping") == :error
  end
```

The test is similar to "removes bucket on exit" except that we are being a bit more harsh by sending `:shutdown` as the exit reason instead of `:normal`. Opposite to `Agent.stop/1`, `Process.exit/2` is an asynchronous operation, therefore we cannot simply query `KV.Registry.lookup/2` right after sending the exit signal because there will be no guarantee the bucket will be dead by then. To solve this, we also monitor the bucket during test and only query the registry once we are sure it is DOWN, avoiding race conditions.

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:

```
1) 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. Not only that, if a process is unreacheable, they are harder to introspect.

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. With this strategy, no workers are started during the supervisor initialization, and a new worker is started each time `start_child/2` is called.

Let's define our `KV.Bucket.Supervisor` in `lib/kv/bucket/supervisor.ex` as follows:

```
defmodule KV.Bucket.Supervisor do
  use Supervisor

  # A simple module attribute that stores the supervisor name
  @name KV.Bucket.Supervisor

  def start_link do
    Supervisor.start_link(__MODULE__, :ok, name: @name)
  end

  def start_bucket do
    Supervisor.start_child(@name, [])
  end

  def init(:ok) do
    children = [
      worker(KV.Bucket, [], restart: :temporary)
    ]

    supervise(children, strategy: :simple_one_for_one)
  end
end
```

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

Instead of receiving the registered process name as argument, we have simply decided to name it `KV.Bucket.Supervisor` as we won't spawn different versions of this process. We have also defined a `start_bucket/0` function that will start a bucket as a child of our supervisor named `KV.Bucket.Supervisor`. `start_bucket/0` is the function we are going to invoke instead of calling `KV.Bucket.start_link` directly in the registry.

Finally, 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:

```
iex> {:ok, _} = KV.Bucket.Supervisor.start_link
{:ok, #PID<0.70.0>}
iex> {:ok, bucket} = KV.Bucket.Supervisor.start_bucket
{:ok, #PID<0.72.0>}
iex> KV.Bucket.put(bucket, "eggs", 3)
:ok
iex> KV.Bucket.get(bucket, "eggs")
3
```

Let's change the registry to work with the buckets supervisor by rewriting how buckets are started:

```
  def handle_cast({:create, name}, {names, refs}) do
    if Map.has_key?(names, name) do
      {:noreply, {names, refs}}
    else
      {:ok, pid} = KV.Bucket.Supervisor.start_bucket
      ref = Process.monitor(pid)
      refs = Map.put(refs, ref, name)
      names = Map.put(names, name, pid)
      {:noreply, {names, refs}}
    end
  end
```

Once we perform those changes, our test suite should fail as there is no bucket supervisor. Instead of directly starting the bucket supervisor on every test, let's automatically start it as part of our main supervision tree.

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` and change `init/1` to match the following:

```
def init(:ok) do
  children = [
    worker(KV.Registry, [KV.Registry]),
    supervisor(KV.Bucket.Supervisor, [])
  ]

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

This time we have added a supervisor as child, starting it with no arguments. Re-run the test suite and now all tests should pass.

Since we have added more children to the supervisor, it is also important to evaluate if the `:one_for_one` supervision strategy is still correct. One flaw that shows up right away is the relationship between the `KV.Registry` worker process and the `KV.Bucket.Supervisor` supervisor process. If `KV.Registry` dies, all information linking `KV.Bucket` names to `KV.Bucket` processes is lost, and therefore `KV.Bucket.Supervisor` must die too- otherwise, the `KV.Bucket` processes it manages would be orphaned.

In light of this observation, we should consider moving to another supervision strategy. Two likely candidates are `:one_for_all` and `:rest_for_one`. A supervisor using the `:one_for_all` strategy will kill and restart all of its children processes whenever any one of them dies. At first glance, this would appear to suit our use case, but it also seems a little heavy-handed, because `KV.Registry` is perfectly capable of cleaning itself up if `KV.Bucket.Supervisor` dies. In this case, the `:rest_for_one` strategy comes in handy- when a child process crashes, the supervisor will only kill and restart child processes which were started *after* the crashed child. Let's rewrite our supervision tree to use this strategy instead:

```
def init(:ok) do
  children = [
    worker(KV.Registry, [KV.Registry]),
    supervisor(KV.Bucket.Supervisor, [])
  ]

  supervise(children, strategy: :rest_for_one)
end
```

Now, if the registry worker crashes, both the registry and the "rest" of `KV.Supervisor` 's children (i.e. `KV.Bucket.Supervisor`) will be restarted. However, if `KV.Bucket.Supervisor` crashes, `KV.Registry` will not be restarted, because it was started prior to `KV.Bucket.Supervisor` .

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 both `Supervisor` and `Supervisor.Spec` modules.

There are two topics left before we move on to the next chapter.

Observer

Now that we have defined our supervision tree, it is a great opportunity to introduce the Observer tool that ships with Erlang. Start your application with `iex -S mix` and key this in:

```
iex> :observer.start
```

A GUI should pop-up containing all sorts of information about our system, from general statistics to load charts as well as a list of all running processes and applications.

In the Applications tab, you will see all applications currently running in your system along side their supervision tree. You can select the `kv` application to explore it further:

Not only that, as you create new buckets on the terminal, you should see new processes spawned in the supervision tree shown in Observer:

```
iex> KV.Registry.create KV.Registry, "shopping"
:ok
```

We will leave it up to you to further explore what Observer provides. Note you can double click any process in the supervision tree to retrieve more information about it, as well as right-click a process to send "a kill signal", a perfect way to emulate failures and see if your supervisor reacts as expected.

At the end of the day, tools like Observer is one of the main reasons you want to always start processes inside supervision trees, even if they are temporary, to ensure they are always reachable and introspectable.

Shared state in tests

So far we have been starting one registry per test to ensure they are isolated:

```
setup context do
  {:ok, registry} = KV.Registry.start_link(context.test)
  {:ok, registry: registry}
end
```

Since we have now changed our registry to use `KV.Bucket.Supervisor`, which is registered globally, our tests are now relying on this shared, global supervisor even though each test has its own registry. The question is: should we?

It depends. It is ok to rely on shared global state as long as we depend only on a non-shared partition of this state. For example, every time we register a process under a given name, we are registering a process against a shared name registry. However, as long as we guarantee the names are specific to each test, by using a construct like `context.test`, we won't have concurrency or data dependency issues between tests.

Similar reasoning should be applied to our bucket supervisor. Although multiple registries may start buckets on the shared bucket supervisor, those buckets and registries are isolated from each other. We would only run into concurrency issues if we used a function like `Supervisor.count_children(KV.Bucket.Supervisor)` which would count all buckets from all registries, potentially giving different results when tests run concurrently.

Since we have relied only on a non-shared partition of the bucket supervisor so far, we don't need to worry about concurrency issues in our test suite. In case it ever becomes a problem, we can start a supervisor per test and pass it as an argument to the registry `start_link` function.

Now that our application is properly supervised and tested, let's see how we can speed things up.

ETS

Every time we need to look up a bucket, we need to send a message to the registry. In case our registry is being accessed concurrently by multiple processes, 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.

> 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.

ETS as a cache

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

```
iex> table = :ets.new(:buckets_registry, [:set, :protected])
8207
iex> :ets.insert(table, {"foo", self})
true
iex> :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`, meaning only the process that created the table can write to it, but all processes can read 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:

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

Let's change the `KV.Registry` to use ETS tables. Since our registry requires a name as argument, we are going to name the ETS table with the same name as the registry. ETS names and process names are stored in different locations, so there is no chance of conflicts.

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:

```
defmodule KV.Registry do
  use GenServer

  ## Client API

  @doc """
  Starts the registry with the given `name`.
```

```
"""
  def start_link(name) do
    # 1\. Pass the name to GenServer's init
    GenServer.start_link(__MODULE__, name, name: name)
  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) when is_atom(server) do
    # 2\. Lookup is now done directly in ETS, without accessing the server
    case :ets.lookup(server, name) do
      [{^name, pid}] -> {:ok, pid}
      [] -> :error
    end
  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

  @doc """
  Stops the registry.
"""
  def stop(server) do
    GenServer.stop(server)
  end

  ## Server callbacks

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

  # 4\. The previous handle_call callback for lookup was removed

  def handle_cast({:create, name}, {names, refs}) do
    # 5\. Read and write to the ETS table instead of the map
    case lookup(names, name) do
      {:ok, _pid} ->
        {:noreply, {names, refs}}
      :error ->
        {:ok, pid} = KV.Bucket.Supervisor.start_bucket
        ref = Process.monitor(pid)
        refs = Map.put(refs, ref, name)
        :ets.insert(names, {name, pid})
        {:noreply, {names, refs}}
    end
  end

  def handle_info({:DOWN, ref, :process, _pid, _reason}, {names, refs}) do
    # 6\. Delete from the ETS table instead of the map
    {name, refs} = Map.pop(refs, ref)
    :ets.delete(names, name)
    {:noreply, {names, refs}}
  end

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

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 broken our tests because they were using the pid of the registry process for all operations and now the registry lookup requires the ETS table name. However, since the ETS table has the same name as the registry process, it is an easy fix. Change the setup function in `test/kv/registry_test.exs` to the following:

```
    setup context do
      {:ok, _} = KV.Registry.start_link(context.test)
      {:ok, registry: context.test}
    end
```

Once we change `setup`, some tests will continue to fail. You may even notice tests pass and fail inconsistently between runs. For example, the "spawns buckets" test:

```
  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
```

may be failing on this line:

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

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`)

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 tests 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, &quot;shopping&quot;)`
2. The registry creates the bucket and updates the cache table
3. We access the information from the table with `KV.Registry.lookup(registry, &quot;shopping&quot;)`
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, &quot;shopping&quot;)`
2. We access the information from the table with `KV.Registry.lookup(ets, &quot;shopping&quot;)`
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:

```
def create(server, name) do
  GenServer.call(server, {:create, name})
end

def handle_call({:create, name}, _from, {names, refs}) do
  case lookup(names, name) do
    {:ok, pid} ->
      {:reply, pid, {names, refs}}
    :error ->
      {:ok, pid} = KV.Bucket.Supervisor.start_bucket
      ref = Process.monitor(pid)
      refs = Map.put(refs, ref, name)
      :ets.insert(names, {name, pid})
      {:reply, pid, {names, refs}}
  end
end
```

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. Generally speaking, Elixir developers prefer to use `call/2` instead of `cast/2` as it also provides back-pressure (you block until you get a reply). Using `cast/2` when not necessary can also be considered a premature optimization.

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 or two intermittent failures:

```
  1) test removes buckets on exit (KV.RegistryTest)
     test/kv/registry_test.exs:19
     Assertion with == failed
     code: KV.Registry.lookup(registry, "shopping") == :error
     lhs:  {:ok, #PID<0.109.0>}
     rhs:  :error
     stacktrace:
       test/kv/registry_test.exs:23
```

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` , the operation responsible for cleaning the ETS table, to a synchronous operation. Instead we need to find a way to guarantee the registry has processed the `:DOWN` notification sent when the bucket crashed.

An easy way to do so is by sending a synchronous request to the registry: because messages are processed in order, if the registry replies to a request sent after the `Agent.stop` call, it means that the `:DOWN` message has been processed. Let's do so by creating a "bogus" bucket, which is a synchronous request, after `Agent.stop` in both tests:

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

  # Do a call to ensure the registry processed the DOWN message
  _ = KV.Registry.create(registry, "bogus")
  assert KV.Registry.lookup(registry, "shopping") == :error
end

test "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)

  # Wait until the bucket is dead
  ref = Process.monitor(bucket)
  assert_receive {:DOWN, ^ref, _, _, _}

  # Do a call to ensure the registry processed the DOWN message
  _ = KV.Registry.create(registry, "bogus")
  assert KV.Registry.lookup(registry, "shopping") == :error
end
```

Our tests should now (always) pass!

This concludes our optimization chapter. We have used ETS as a cache mechanism where reads can happen from any processes but writes are still serialized through a single process. More importantly, we have also learned that once data can be read asynchronously, we need to be aware of the race conditions it might introduce.

Next let's discuss external and internal dependencies and how Mix helps us manage large codebases.

Dependencies and umbrella projects

In this chapter, we will 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:

```
CREATE shopping
OK

PUT shopping milk 1
OK

PUT shopping eggs 3
OK

GET shopping milk
1
OK

DELETE shopping eggs
OK
```

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.

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:

```
def deps do
  [{:plug, "~> 1.0"}]
end
```

This dependency refers to the latest version of Plug in the 1.x.x version series that has been pushed to Hex. This is indicated by the `~&gt;` 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:

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

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, dependencies 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.

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 methods to work 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:

```
def deps do
  [{:kv, git: "https://github.com/YOUR_ACCOUNT/kv.git"}]
end
```

If the repository is private though, you may need to specify the private URL `git@github.com:YOUR_ACCOUNT/kv.git`. In any case, Mix will be able to fetch it for you as long as you have the proper credentials.

Using git dependencies for internal dependencies is somewhat discouraged in Elixir. Remember that the runtime and the Elixir ecosystem already provide the concept of applications. 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 may become very hard to maintain as you will 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 while keeping all of them in a single source code repository. That is exactly the style we are going to explore in the next sections.

Let's 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!

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):

```
defmodule KvUmbrella.Mixfile do
  use Mix.Project

  def project do
    [apps_path: "apps",
     build_embedded: Mix.env == :prod,
     start_permanent: Mix.env == :prod,
     deps: deps]
  end

  defp deps do
    []
  end
end
```

What makes this project different from the previous one is simply the `apps_path: &quot;apps&quot;` 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 their own dependencies (not shared with children). We'll create new applications inside the apps directory.

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`:

```
defmodule KVServer.Mixfile do
  use Mix.Project

  def project do
    [app: :kv_server,
     version: "0.1.0",
     build_path: "../../_build",
     config_path: "../../config/config.exs",
     deps_path: "../../deps",
     lockfile: "../../mix.lock",
     elixir: "~> 1.3",
     build_embedded: Mix.env == :prod,
     start_permanent: Mix.env == :prod,
     deps: deps]
  end

  def application do
    [applications: [:logger],
     mod: {KVServer, []}]
  end

  defp deps do
    []
  end
end
```

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

```
build_path: "../../_build",
config_path: "../../config/config.exs",
deps_path: "../../deps",
lockfile: "../../mix.lock",
```

Those options mean all dependencies will be checked out to `kv_umbrella/deps`, and they will share the same build, config and lock files. This ensures dependencies will be fetched and compiled once for the whole umbrella structure, instead of once per umbrella application.

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

```
def application do
  [applications: [:logger],
   mod: {KVServer, []}]
end
```

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`:

```
defmodule 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
end
```

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.

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:

```
defp deps do
  [{:kv, in_umbrella: true}]
end
```

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`:

```
def application do
  [applications: [:logger, :kv],
   mod: {KVServer, []}]
end
```

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:

```
build_path: "../../_build",
config_path: "../../config/config.exs",
deps_path: "../../deps",
lockfile: "../../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. Dependencies between them must be explicitly listed. This allows them to be developed together, but compiled, tested and deployed independently if desired.

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 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 make 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.

Echo server

We will start our TCP server by first implementing an echo server. It will simply send a response with the 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, performs the following steps:

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

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

```
require Logger

def accept(port) do
  # The options below mean:
  #
  # 1\. `:binary` - receives data as binaries (instead of lists)
  # 2\. `packet: :line` - receives data line by line
  # 3\. `active: false` - blocks on `:gen_tcp.recv/2` until data is available
  # 4\. `reuseaddr: true` - allows us to reuse the address if the listener crashes
  #
  {:ok, socket} = :gen_tcp.listen(port,
                    [:binary, packet: :line, active: false, reuseaddr: true])
  Logger.info "Accepting connections on port #{port}"
  loop_acceptor(socket)
end

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

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 to the port until the socket becomes available and then call `loop_acceptor/1` . `loop_acceptor/1` is just a loop accepting client connections. For each accepted connection, we call `serve/1` .

`serve/1` is another loop that reads a line from the socket and writes those lines back to the socket. Note that the `serve/1` function uses the pipe operator `|&gt;` to express this flow of operations. The pipe operator evaluates the left side and passes its result as first argument to the function on the right side. The example above:

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

is equivalent to:

```
write_line(read_line(socket), socket)
```

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` . Inside IEx, run:

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

The server is now running, and you will even notice the console is blocked. Let's use a `telnet` client to access our server. There are clients available on most operating systems, and their command lines are generally 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 + ]` , typing `quit` , and pressing `&lt;Enter&gt;` , but your client may require different steps.

Once you exit the telnet client, you will likely see an error 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 future revisions of our server.

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, because it won't be restarted. That's why we must move our server to a supervision tree.

Task and gen_tcp

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

Tasks

We have learned about agents, generic servers, and supervisors. They are all meant to work with multiple messages or manage state. But what do we use when we only need to execute some task and that is it?

The Task module provides this functionality exactly. For example, it has a `start_link/3` function that receives a 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 in the `start/2` function to the following:

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

  children = [
    worker(Task, [KVServer, :accept, [4040]])
  ]

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

With this change, we are saying that we want to run `KVServer.accept(4040)` as a worker. We are hardcoding the port for now, but we will discuss ways in which this could be changed later.

Now that the server is part of the supervision tree, it should start automatically when we run the application. Type `mix run --no-halt` in the terminal, and once again use the `telnet` client to make sure that everything still 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 starts right away. However, does it *scale*?

Try to connect two telnet clients at the same time. When you do so, you will notice that the second client doesn'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. When one client is connected, we can't accept another client.

Task supervisor

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

```
defp loop_acceptor(socket) do
  {:ok, client} = :gen_tcp.accept(socket)
  serve(client)
  loop_acceptor(socket)
end
```

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

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

We are starting a linked Task directly from the acceptor process. But we've already made this mistake once. Do you remember?

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

The code above would have the same flaw: if we link the `serve(client)` task to the acceptor, a crash when serving a request would bring the 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 tactic here, except that this pattern is so common with tasks that `Task` already comes with a solution: 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:

```
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
```

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

Now we just need to change `loop_acceptor/1` to use `Task.Supervisor` to serve each request:

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

You might notice that we added a line, `:ok = :gen_tcp.controlling_process(client, pid)`. This makes the child process the "controlling process" of the `client` socket. If we didn't do this, the acceptor would bring down all the clients if it crashed because sockets are tied to the process that `accept`ed them by default.

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:

```
defmodule KVServer do
  use Application
  require Logger

  @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, reuseaddr: true])
    Logger.info "Accepting connections on port #{port}"
    loop_acceptor(socket)
  end

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

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

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

Docs, tests and with

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

```
CREATE shopping
OK

PUT shopping milk 1
OK

PUT shopping eggs 3
OK

GET shopping milk
1
OK

DELETE shopping eggs
OK
```

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

Doctests

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

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

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

```
defmodule KVServer.Command do
  @doc ~S"""
  Parses the given `line` into a command.

  ## Examples

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

  """
  def parse(line) do
    :not_implemented
  end
end
```

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

Also note that we started the documentation string using `@doc ~S&quot;&quot;&quot;`. The `~S` prevents the `\r\n` characters from being converted to a carriage return and line feed until they are evaluated in the test.

To run our doctests, we'll create a file at `test/kv_server/command_test.exs` and call `doctest KVServer.Command` in the test case:

```
defmodule KVServer.CommandTest do
  use ExUnit.Case, async: true
  doctest KVServer.Command
end
```

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:

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

Our implementation simply splits the line on whitespace and then matches the command against a list. Using `String.split/1` means our commands will be whitespace-insensitive. Leading and trailing whitespace won't matter, nor will consecutive spaces between words. Let's add some new doctests to test this behaviour along with the other commands:

```
@doc ~S"""
Parses 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"}}

Unknown commands or commands with the wrong number of
arguments return an error:

    iex> KVServer.Command.parse "UNKNOWN shopping eggs\r\n"
    {:error, :unknown_command}

    iex> KVServer.Command.parse "GET shopping\r\n"
    {:error, :unknown_command}

"""
```

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

```
def parse(line) do
  case String.split(line) do
    ["CREATE", bucket] -> {:ok, {:create, bucket}}
    ["GET", bucket, key] -> {:ok, {:get, bucket, key}}
    ["PUT", bucket, key, value] -> {:ok, {:put, bucket, key, value}}
    ["DELETE", bucket, key] -> {:ok, {:delete, bucket, key}}
    _ -> {:error, :unknown_command}
  end
end
```

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

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 the following to define two different tests:

```
iex> KVServer.Command.parse "UNKNOWN shopping eggs\r\n"
{:error, :unknown_command}

iex> KVServer.Command.parse "GET shopping\r\n"
{:error, :unknown_command}
```

Without new lines, as seen below, ExUnit compiles it into a single test:

```
iex> KVServer.Command.parse "UNKNOWN shopping eggs\r\n"
{:error, :unknown_command}
iex> KVServer.Command.parse "GET shopping\r\n"
{:error, :unknown_command}
```

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

with

As we are now able to parse commands, we can finally start implementing the logic that runs the commands. Let's add a stub definition for this function for now:

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

Before we implement this function, let's change our server to start using our new `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:

```
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
```

by the following:

```
defp 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)
end

defp read_line(socket) do
  :gen_tcp.recv(socket, 0)
end

defp write_line(socket, {:ok, text}) do
  :gen_tcp.send(socket, text)
end

defp write_line(socket, {:error, :unknown_command}) do
  # Known error. Write to the client.
  :gen_tcp.send(socket, "UNKNOWN COMMAND\r\n")
end

defp write_line(_socket, {:error, :closed}) do
  # The connection was closed, exit politely.
  exit(:shutdown)
end

defp write_line(socket, {:error, error}) do
  # Unknown error. Write to the client and exit.
  :gen_tcp.send(socket, "ERROR\r\n")
  exit(error)
end
```

If we start our server, we can now send commands to it. For now we will 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 pipelines which made the logic straight-forward to follow. However, now that we need to handle different error codes along the way, our server logic is nested inside many `case` calls.

Thankfully, Elixir v1.2 introduced a construct called `with` which allows to simplify code like above. Let's rewrite the `serve/1` function to use it:

```
defp serve(socket) do
  msg =
    with {:ok, data} <- read_line(socket),
         {:ok, command} <- KVServer.Command.parse(data),
         do: KVServer.Command.run(command)

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

Much better! Syntax-wise, `with` is quite similar to `for` comprehensions. `with` will retrieve the value returned by the right-side of `&lt;-` and match it against the pattern on the left side. If the value matches the pattern, `with` moves on to the next expression. In case there is no match, the non-matching value is returned.

In other words, we converted each expression given to `case/2` as a step in `with`. As soon as any of the steps return something that does not match `{:ok, x}`, `with` aborts, and returns the non-matching value.

You can read more about `with` in our documentation.

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:

```
@doc """
Runs the given command.
"""
def run(command)

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

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

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

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

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

The implementation is straightforward: we just dispatch to the `KV.Registry` server that we registered during the `:kv` application startup. Since our `:kv_server` depends on the `:kv` application, it is completely fine to depend on the servers/services it provides.

Note that 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 `write_line/2` function in `KV.Server` to print such error as well:

```
defp write_line(socket, {:error, :not_found}) do
  :gen_tcp.send(socket, "NOT FOUND\r\n")
end
```

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 made.

`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 between having unit tests that are isolated and can run asynchronously, or writing integration tests that work on top of the global state, but exercise our application's full stack as it is meant to be exercised in production.

So far we have only written unit tests, typically testing a single module directly. However, 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:

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

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

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

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

However, it comes with the downside that our APIs become increasingly large in order to accommodate all external parameters.

The alternative is to write integration tests that will rely on the global server names to exercise the whole stack, from the TCP server to the bucket. The downside of integration tests 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.

This time we will write an integration test. The integration test will use a TCP client that sends commands to our server and assert we are getting the desired responses.

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

```
defmodule KVServerTest do
  use ExUnit.Case

  setup do
    Application.stop(:kv)
    :ok = Application.start(:kv)
  end

  setup do
    opts = [:binary, packet: :line, active: false]
    {:ok, socket} = :gen_tcp.connect('localhost', 4040, opts)
    {:ok, socket: socket}
  end

  test "server interaction", %{socket: socket} do
    assert send_and_recv(socket, "UNKNOWN shopping\r\n") ==
           "UNKNOWN COMMAND\r\n"

    assert send_and_recv(socket, "GET shopping eggs\r\n") ==
           "NOT FOUND\r\n"

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

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

    # 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
end
```

Our integration test checks all server interaction, including unknown commands and not found errors. It is worth noting that, as with *ETS* tables and linked processes, there is no need to close the socket. 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 state, we stop and start the `:kv` application before each test. In fact, stopping the `:kv` application even prints a warning on the terminal:

```
18:12:10.698 [info] Application kv exited: :stopped
```

To avoid printing log messages during tests, ExUnit provides a neat feature called `:capture_log`. By setting `@tag :capture_log` before each test or `@moduletag :capture_log` for the whole test case, ExUnit will automatically capture anything that is logged while the test runs. In case our test fails, the captured logs will be printed alongside the ExUnit report.

Before setup, add the following call:

```
@moduletag :capture_log
```

In case the test crashes, you will see a report as follows:

```
  1) test server interaction (KVServerTest)
     test/kv_server_test.exs:17
     ** (RuntimeError) oops
     stacktrace:
       test/kv_server_test.exs:29

     The following output was logged:

     13:44:10.035 [info]  Application kv exited: :stopped
```

With this simple integration test, we start to see why integration tests may be slow. Not only can this particular test not 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 out the best testing strategy for your applications. You need to balance code quality, confidence, and test suite runtime. For example, we may start with testing the server only with integration tests, but if the server continues to grow in future 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.

In the next chapter we will finally make our system distributed by adding a bucket routing mechanism. We'll also learn about application configuration.

Distributed tasks and configuration

In this last chapter, we will go back to the `:kv` application and add a routing layer that will allow us to distribute requests 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" (`?a` represents the Unicode codepoint of the letter "a") will be dispatched to node `foo@computer-name` .

If the matching entry points to the node evaluating the request, then we've finished routing, and this node will perform the requested operation. If the matching entry points to a different node, we'll pass the request to this node, which will look at its own routing table (which may be different from the one in the first node) and act accordingly. If no entry matches, an error will be raised.

You may wonder why we don't simply tell the node we find in our routing table to perform the requested operation directly, but instead pass the routing request on to that node to process. While a routing table as simple as the one above might reasonably be shared between all nodes, passing on the routing request in this way makes it much simpler to break the routing table into smaller pieces as our application grows. Perhaps at some point, `foo@computer-name` will only be responsible for routing bucket requests, and the buckets it handles will be dispatched to different nodes. In this way, `bar@computer-name` does not need to know anything about this change.

> Note: we will be using two nodes in the same machine throughout this chapter. You are free to use two (or more) different machines in the same network but you need to do some prep work. First of all, you need to ensure all machines have a `~/.erlang.cookie` file with exactly the same value. Second, you need to guarantee epmd is running on a port that is not blocked (you can run `epmd -d` for debug info). Third, if you want to learn more about distribution in general, we recommend this great Distribunomicon chapter from Learn You Some Erlang.

Our first distributed code

Elixir ships with facilities to connect nodes and exchange information between them. In fact, we use the same concepts of processes, message passing and receiving messages when working in a distributed environment because Elixir processes are *location transparent*. This means that when sending a message, it doesn't matter if the recipient process is on the same node or on 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`, so I see `foo@jv` in the example above, but you will get a different result. We will use `foo@computer-name` in the following examples and you should update them accordingly when trying out the code.

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

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

If you have another computer on 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. In either case, give it the short name of `bar`:

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

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

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

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

```
iex> Node.spawn_link :"foo@computer-name", fn -> Hello.world end
#PID<9014.59.0>
hello world
```

Elixir spawned a process on another node and returned its pid. The code then executed on the other node where the `Hello.world/0` function exists and invoked that function. Note that the result of "hello world" was printed on the current node `bar` and not on `foo` . In other words, the message to be printed was sent back from `foo` to `bar` . This happens because the process spawned on 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:

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

From our quick exploration, we could conclude that we should simply use `Node.spawn_link/2` to spawn processes on a remote node every time we need to do a distributed computation. However we have learned throughout this 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` that we could use in our implementation:

1. We could use Erlang's :rpc module to execute functions on a remote node. Inside the `bar@computer-name` shell above, you can call `:rpc.call(:&quot;foo@computer-name&quot;, Hello, :world, [])` and it will print "hello world"
2. We could have a server running on 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. We could use tasks, which we have learned about in a previous chapter, as they can be spawned on both local and remote nodes

The options above have different properties. Both `:rpc` and using a GenServer would serialize your requests on a single server, while tasks are effectively running asynchronously on 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.

async/await

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

```
task = Task.async(fn -> compute_something_expensive end)
res  = compute_something_else()
res + 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.

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 on the supervisor. Open up `lib/kv/supervisor.ex` from the `:kv` application. Let's add a task supervisor as the last child of the tree:

```
supervisor(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@computer-name`, we can now spawn a task directly on the other node via the supervisor:

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

Our first distributed task simply retrieves the name of the node the task is running on. Notice we have given an anonymous function to `Task.Supervisor.async/2` but, in distributed cases, it is preferable to give the module, function and arguments explicitly:

```
iex> task = Task.Supervisor.async {KV.RouterTasks, :"foo@computer-name"}, Kernel, :nod
e, []
%Task{pid: #PID<12467.88.0>, ref: #Reference<0.0.0.400>}
iex> Task.await(task)
:"foo@computer-name"
```

The difference is that anonymous functions requires the target node to have exactly the same code version as the caller. Using module, function and arguments is more robust because you only need to find a function with matching arity in the given module.

With this knowledge in hand, let's finally write the routing code.

Routing layer

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

```
defmodule KV.Router do
  @doc """
  Dispatch the given `mod`, `fun`, `args` request
  to the appropriate node based on the `bucket`.
"""
  def route(bucket, mod, fun, args) do
    # Get the first byte of the 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
      {KV.RouterTasks, elem(entry, 1)}
      |> Task.Supervisor.async(KV.Router, :route, [bucket, mod, fun, args])
      |> 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 with your local machine name.
    [{?a..?m, :"foo@computer-name"},
     {?n..?z, :"bar@computer-name"}]
  end
end
```

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

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

  test "route requests across 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
end
```

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 that the code raises for unknown entries.

In order to run the first test, we need to have two nodes running. Move into `apps/kv` and 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 successfully pass. Excellent!

Test filters and tags

Although our tests pass, our testing structure is getting more complex. In particular, running tests with only `mix test` causes failures in our suite, since our test requires a connection to another node.

Luckily, ExUnit ships with a facility to tag tests, allowing us to run specific callbacks or even filter tests altogether based on those tags. We have already used the `:capture_log` tag in the previous chapter, which has its semantics specified by ExUnit itself.

This time let's add a `:distributed` tag to `test/kv/router_test.exs`:

```
@tag :distributed
test "route requests across nodes" do
```

Writing `@tag :distributed` is equivalent to writing `@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:

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

ExUnit.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, 1 skipped
```

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 should 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.

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. This allows us not only to configure development/test/production, but also to allow different nodes to run with different entries in the routing table. There is a feature of *OTP* that does exactly that: the application environment.

Each application has an environment that stores the application's 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:

```
def application do
  [applications: [],
   env: [routing_table: []],
   mod: {KV, []}]
end
```

We have added a new `:env` key to the application. It 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 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` with the definition below:

```
@doc """
The routing table.
"""
def table do
  Application.fetch_env!(:kv, :routing_table)
end
```

We use `Application.fetch_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 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:

```
config :iex, default_prompt: ">>>"
```

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

This means we can also configure our `:routing_table` directly in the `apps/kv/config/config.exs` file:

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

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

Since Elixir v1.2, all umbrella applications share their configurations, thanks to this line in `config/config.exs` in the umbrella root that loads the configuration of all children:

```
import_config "../apps/*/config/config.exs"
```

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

Overall, the built-in ability to configure applications and the fact that 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)

As we add more applications in the future, we can continue controlling our 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 we can deploy the application even if the runtime is not pre-installed on 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

Summing up

In this chapter we have built a simple router as a way to explore the distributed features of Elixir and 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 generic servers, supervisors, tasks, agents, applications and more. Not only that, we have written tests for the whole application, got familiar with ExUnit, and learned how to use the Mix build tool to accomplish a wide range of tasks.

If 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.

There are many more lessons to learn and we hope you had fun so far!