Distributed tasks and configuration

> *This chapter is part of the Mix and OTP guide and it depends on previous chapters in this guide. For more information, read the introduction guide or check out the chapter index in the sidebar.*

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"}
]
```

News: Elixir v1.8 released

Search...

Watch the Elixir mini-documentary!

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 tell the node we found 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 on 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 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 messages 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 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(:"foo@computer-name", 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 server on a remote node by using `GenServer.call({name, node}, arg)` or passing the remote process PID as the 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:

```
{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{owner: #PID<0.122.0>, pid: #PID<12467.88.0>, ref:
#Reference<0.0.0.400>}
iex> Task.await(task)
{:ok, :"foo@computer-name"}
```

Our first distributed task 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, :node, []
%Task{owner: #PID<0.122.0>, pid: #PID<12467.89.0>, ref:
#Reference<0.0.0.404>}
iex> Task.await(task)
:"foo@computer-name"
```

The difference is that anonymous functions require 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 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 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
```

The test should pass.

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
  [
    extra_applications: [:logger],
    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 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 Distillery, 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 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.

Happy coding!

← Previous Top