Elixir Documentation

Release

Elixir community

Sep 18, 2017

Contents

This is an experiment to organize the different kinds of Elixir documentation in a new way.

This site has been generated from this version of the source.

CHAPTER 1

Getting started guides

Introduction to Elixir

1 Introduction

{% include toc.html %}

Welcome!

In this tutorial we are going to teach you the Elixir foundation, the language syntax, how to define modules, how to manipulate the characteristics of common data structures and more. This chapter will focus on ensuring Elixir is installed and that you can successfully run Elixir's Interactive Shell, called IEx.

Our requirements are:

- Erlang - Version 17.0 onwards
- Elixir - Version 1.0.0 onwards

Let's get started!

> If you find any errors in the tutorial or on the website, please report a bug or send a pull request to our issue tracker. If you suspect it is a language bug, please let us know in the language issue tracker.

1.1 Installation

If you still haven't installed Elixir, run to our installation page. Once you are done, you can run `elixir -v` to get the current Elixir version.

1.2 Interactive mode

When you install Elixir, you will have three new executables: `iex`, `elixir` and `elixirc`. If you compiled Elixir from source or are using a packaged version, you can find these inside the `bin` directory.

For now, let's start by running `iex` (or `iex.bat` if you are on Windows) which stands for Interactive Elixir. In interactive mode, we can type any Elixir expression and get its result. Let's warm up with some basic expressions:

```
Interactive Elixir - press Ctrl+C to exit (type h() ENTER for help)

iex> 40 + 2
42
iex> "hello" <> " world"
"hello world"
```

It seems we are ready to go! We will use the interactive shell quite a lot in the next chapters to get a bit more familiar with the language constructs and basic types, starting in the next chapter.

1.3 Running scripts

After getting familiar with the basics of the language you may want to try writing simple programs. This can be accomplished by putting Elixir code into a file and executing it with `elixir`:

```
$ cat simple.exs
IO.puts "Hello world
from Elixir"

$ elixir simple.exs
Hello world
from Elixir
```

Later on we will learn how to compile Elixir code (in Chapter 8) and how to use the Mix build tool (in the Mix & OTP guide). For now, let's move on to Chapter 2.

2 Basic types

{% include toc.html %}

In this chapter we will learn more about Elixir basic types: integers, floats, booleans, atoms, strings, lists and tuples. Some basic types are:

```
iex> 1          # integer
iex> 0x1F       # integer
iex> 1.0        # float
iex> true       # boolean
iex> :atom      # atom / symbol
iex> "elixir"   # string
iex> [1, 2, 3]  # list
iex> {1, 2, 3}  # tuple
```

2.1 Basic arithmetic

Open up `iex` and type the following expressions:

```
iex> 1 + 2
3
iex> 5 * 5
25
iex> 10 / 2
5.0
```

Notice that `10 / 2` returned a float `5.0` instead of an integer `5`. This is expected. In Elixir, the operator `/` always returns a float. If you want to do integer division or get the division remainder, you can invoke the `div` and `rem` functions:

```
iex> div(10, 2)
5
iex> div 10, 2
5
iex> rem 10, 3
1
```

Notice that parentheses are not required in order to invoke a function.

Elixir also supports shortcut notations for entering binary, octal and hexadecimal numbers:

```
iex> 0b1010
10
iex> 0o777
511
iex> 0x1F
31
```

Float numbers require a dot followed by at least one digit and also support `e` for the exponent number:

```
iex> 1.0
1.0
iex> 1.0e-10
1.0e-10
```

Floats in Elixir are 64 bit double precision.

You can invoke the `round` function to get the closest integer to a given float, or the `trunc` function to get the integer part of a float.

```
iex> round 3.58
4
iex> trunc 3.58
3
```

2.2 Booleans

Elixir supports `true` and `false` as booleans:

```
iex> true
true
iex> true == false
false
```

Elixir provides a bunch of predicate functions to check for a value type. For example, the `is_boolean/1` function can be used to check if a value is a boolean or not:

> Note: Functions in Elixir are identified by name and by number of arguments (i.e. arity). Therefore, `is_boolean/1` identifies a function named `is_boolean` that takes 1 argument. `is_boolean/2` identifies a different (nonexistent) function with the same name but different arity.

```
iex> is_boolean(true)
true
```

```
iex> is_boolean(1)
false
```

You can also use `is_integer/1`, `is_float/1` or `is_number/1` to check, respectively, if an argument is an integer, a float or either.

> Note: At any moment you can type `h` in the shell to print information on how to use the shell. The `h` helper can also be used to access documentation for any function. For example, typing `h is_integer/1` is going to print the documentation for the `is_integer/1` function. It also works with operators and other constructs (try `h ==/2`).

2.3 Atoms

Atoms are constants where their name is their own value. Some other languages call these symbols:

```
iex> :hello
:hello
iex> :hello == :world
false
```

The booleans `true` and `false` are, in fact, atoms:

```
iex> true == :true
true
iex> is_atom(false)
true
iex> is_boolean(:false)
true
```

2.4 Strings

Strings in Elixir are inserted between double quotes, and they are encoded in UTF-8:

```
iex> "hellö"
"hellö"

Note: if you are running on Windows, there is a chance your terminal
does not use UTF-8 by default. You can change the encoding of your
current session by running ``chcp 65001``.
```

Elixir also supports string interpolation:

```
iex> "hellö #{:world}"
"hellö world"
```

Strings can have line breaks in them or introduce them using escape sequences:

```
iex> "hello
...> world"
"hello\nworld"
iex> "hello\nworld"
"hello\nworld"
```

You can print a string using the `IO.puts/1` function from the `IO` module:

```
iex> IO.puts "hello\nworld"
hello
world
:ok
```

Notice the `IO.puts/1` function returns the atom `:ok` as result after printing.

Strings in Elixir are represented internally by binaries which are sequences of bytes:

```
iex> is_binary("hellö")
true
```

We can also get the number of bytes in a string:

```
iex> byte_size("hellö")
6
```

Notice the number of bytes in that string is 6, even though it has 5 characters. That's because the character "ö" takes 2 bytes to be represented in UTF-8. We can get the actual length of the string, based on the number of characters, by using the `String.length/1` function:

```
iex> String.length("hellö")
5
```

The String module contains a bunch of functions that operate on strings as defined in the Unicode standard:

```
iex> String.upcase("hellö")
"HELLÖ"
```

2.5 Anonymous functions

Functions are delimited by the keywords `fn` and `end`:

```
iex> add = fn a, b -> a + b end
#Function<12.71889879/2 in :erl_eval.expr/5>
iex> is_function(add)
true
iex> is_function(add, 2)
true
iex> is_function(add, 1)
false
iex> add.(1, 2)
3
```

Functions are "first class citizens" in Elixir meaning they can be passed as arguments to other functions just as integers and strings can. In the example, we have passed the function in the variable `add` to the `is_function/1` function which correctly returned `true`. We can also check the arity of the function by calling `is_function/2`.

Note a dot (`.`) between the variable and parenthesis is required to invoke an anonymous function.

Anonymous functions are closures, and as such they can access variables that are in scope when the function is defined:

```
iex> add_two = fn a -> add.(a, 2) end
#Function<6.71889879/1 in :erl_eval.expr/5>
iex> add_two.(2)
4
```

Keep in mind that a variable assigned inside a function does not affect its surrounding environment:

```
iex> x = 42
42
iex> (fn -> x = 0 end).()
0
iex> x
42
```

2.6 (Linked) Lists

Elixir uses square brackets to specify a list of values. Values can be of any type:

```
iex> [1, 2, true, 3]
[1, 2, true, 3]
iex> length [1, 2, 3]
3
```

Two lists can be concatenated and subtracted using the `++/2` and `--/2` operators:

```
iex> [1, 2, 3] ++ [4, 5, 6]
[1, 2, 3, 4, 5, 6]
iex> [1, true, 2, false, 3, true] -- [true, false]
[1, 2, 3, true]
```

Throughout the tutorial, we will talk a lot about the head and tail of a list. The head is the first element of a list and the tail is the remainder of a list. They can be retrieved with the functions `hd/1` and `tl/1`. Let's assign a list to a variable and retrieve its head and tail:

```
iex> list = [1,2,3]
iex> hd(list)
1
iex> tl(list)
[2, 3]
```

Getting the head or the tail of an empty list is an error:

```
iex> hd []
** (ArgumentError) argument error
```

Sometimes you will create a list and it will return a value in single-quotes. For example:

```
iex> [11, 12, 13]
'\v\f\r'
iex> [104, 101, 108, 108, 111]
'hello'
```

When Elixir sees a list of printable ASCII numbers, Elixir will print that as a char list (literally a list of characters). Char lists are quite common when interfacing with existing Erlang code.

Keep in mind single-quoted and double-quoted representations are not equivalent in Elixir as they are represented by different types:

```
iex> 'hello' == "hello"
false
```

Single-quotes are char lists, double-quotes are strings. We will talk more about them in the "Binaries, strings and char lists" chapter.

2.7 Tuples

Elixir uses curly brackets to define tuples. Like lists, tuples can hold any value:

```
iex> {:ok, "hello"}
{:ok, "hello"}
iex> tuple_size {:ok, "hello"}
2
```

Tuples store elements contiguously in memory. This means accessing a tuple element per index or getting the tuple size is a fast operation (indexes start from zero):

```
iex> tuple = {:ok, "hello"}
{:ok, "hello"}
iex> elem(tuple, 1)
"hello"
iex> tuple_size(tuple)
2
```

It is also possible to set an element at a particular index in a tuple with `put_elem/3`:

```
iex> tuple = {:ok, "hello"}
{:ok, "hello"}
iex> put_elem(tuple, 1, "world")
{:ok, "world"}
iex> tuple
{:ok, "hello"}
```

Notice that `put_elem/3` returned a new tuple. The original tuple stored in the `tuple` variable was not modified because Elixir data types are immutable. By being immutable, Elixir code is easier to reason about as you never need to worry if a particular code is mutating your data structure in place.

By being immutable, Elixir also helps eliminate common cases where concurrent code has race conditions because two different entities are trying to change a data structure at the same time.

2.8 Lists or tuples?

What is the difference between lists and tuples?

Lists are stored in memory as linked lists, meaning that each element in a list holds its value and points to the following element until the end of the list is reached. We call each pair of value and pointer a **cons cell**:

```
iex> list = [1|[2|[3|[]]]]
[1, 2, 3]
```

This means accessing the length of a list is a linear operation: we need to traverse the whole list in order to figure out its size. Updating a list is fast as long as we are prepending elements:

```
iex> [0] ++ list
[0, 1, 2, 3]
iex> list ++ [4]
[1, 2, 3, 4]
```

The first operation is fast because we are simply adding a new cons that points to the remaining of `list`. The second one is slow because we need to rebuild the whole list and add a new element to the end.

Tuples, on the other hand, are stored contiguously in memory. This means getting the tuple size or accessing an element by index is fast. However, updating or adding elements to tuples is expensive because it requires copying the whole tuple in memory.

Those performance characteristics dictate the usage of those data structures. One very common use case for tuples is to use them to return extra information from a function. For example, `File.read/1` is a function that can be used to read file contents and it returns tuples:

```
iex> File.read("path/to/existing/file")
{:ok, "... contents ..."}
iex> File.read("path/to/unknown/file")
{:error, :enoent}
```

If the path given to `File.read/1` exists, it returns a tuple with the atom `:ok` as the first element and the file contents as the second. Otherwise, it returns a tuple with `:error` and the error description.

Most of the time, Elixir is going to guide you to do the right thing. For example, there is a `elem/2` function to access a tuple item but there is no built-in equivalent for lists:

```
iex> tuple = {:ok, "hello"}
{:ok, "hello"}
iex> elem(tuple, 1)
"hello"
```

When "counting" the number of elements in a data structure, Elixir also abides by a simple rule: the function should be named `size` if the operation is in constant time (i.e. the value is pre-calculated) or `length` if the operation requires explicit counting.

For example, we have used 4 counting functions so far: `byte_size/1` (for the number of bytes in a string), `tuple_size/1` (for the tuple size), `length/1` (for the list length) and `String.length/1` (for the number of characters in a string). That said, we use `byte_size` to get the number of bytes in a string, which is cheap, but retrieving the number of unicode characters uses `String.length`, since the whole string needs to be iterated.

Elixir also provides `Port`, `Reference` and `PID` as data types (usually used in process communication), and we will take a quick look at them when talking about processes. For now, let's take a look at some of the basic operators that go with our basic types.

3 Basic operators

{% include toc.html %}

In the previous chapter, we saw Elixir provides `+`, `-`, `*`, `/` as arithmetic operators, plus the functions `div/2` and `rem/2` for integer division and remainder.

Elixir also provides `++` and `--` to manipulate lists:

```
iex> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
iex> [1,2,3] -- [2]
[1,3]
```

String concatenation is done with `<>`:

```
iex> "foo" <> "bar"
"foobar"
```

Elixir also provides three boolean operators: `or`, `and` and `not`. These operators are strict in the sense that they expect a boolean (`true` or `false`) as their first argument:

```
iex> true and true
true
iex> false or is_atom(:example)
true
```

Providing a non-boolean will raise an exception:

```
iex> 1 and true
** (ArgumentError) argument error
```

`or` and `and` are short-circuit operators. They only execute the right side if the left side is not enough to determine the result:

```
iex> false and error("This error will never be raised")
false

iex> true or error("This error will never be raised")
true

Note: If you are an Erlang developer, ``and`` and ``or`` in Elixir
actually map to the ``andalso`` and ``orelse`` operators in Erlang.
```

Besides these boolean operators, Elixir also provides `||`, `&&` and `!` which accept arguments of any type. For these operators, all values except `false` and `nil` will evaluate to true:

```
# or
iex> 1 || true
1
iex> false || 11
11

# and
iex> nil && 13
nil
iex> true && 17
17

# !
iex> !true
false
iex> !1
false
iex> !nil
true
```

As a rule of thumb, use `and`, `or` and `not` when you are expecting booleans. If any of the arguments are non-boolean, use `&&`, `||` and `!`.

Elixir also provides ==, !=, ===, !==, <=, >=, < and > as comparison operators:

```
iex> 1 == 1
true
iex> 1 != 2
true
iex> 1 < 2
true
```

The difference between == and === is that the latter is more strict when comparing integers and floats:

```
iex> 1 == 1.0
true
iex> 1 === 1.0
false
```

In Elixir, we can compare two different data types:

```
iex> 1 < :atom
true
```

The reason we can compare different data types is pragmatism. Sorting algorithms don't need to worry about different data types in order to sort. The overall sorting order is defined below:

```
number < atom < reference < functions < port < pid < tuple < maps < list < bitstring
```

You don't actually need to memorize this ordering, but it is important just to know an order exists.

Well, that is it for the introduction. In the next chapter, we are going to discuss some basic functions, data type conversions and a bit of control-flow.

4 Pattern matching

{% include toc.html %}

In this chapter, we will show how the = operator in Elixir is actually a match operator and how to use it to pattern match inside data structures. Finally, we will learn about the pin operator ^ used to access previously bound values.

4.1 The match operator

We have used the = operator a couple times to assign variables in Elixir:

```
iex> x = 1
1
iex> x
1
```

In Elixir, the = operator is actually called *the match operator*. Let's see why:

```
iex> 1 = x
1
iex> 2 = x
** (MatchError) no match of right hand side value: 1
```

Notice that `1 = x` is a valid expression, and it matched because both the left and right side are equal to 1. When the sides do not match, a `MatchError` is raised.

A variable can only be assigned on the left side of =:

```
iex> 1 = unknown
** (RuntimeError) undefined function: unknown/0
```

Since there is no variable `unknown` previously defined, Elixir imagined you were trying to call a function named `unknown/0`, but such a function does not exist.

4.2 Pattern matching

The match operator is not only used to match against simple values, but it is also useful for destructuring more complex data types. For example, we can pattern match on tuples:

```
iex> {a, b, c} = {:hello, "world", 42}
{:hello, "world", 42}
iex> a
:hello
iex> b
"world"
```

A pattern match will error in the case that the sides can't match. This is, for example, the case when the tuples have different sizes:

```
iex> {a, b, c} = {:hello, "world"}
** (MatchError) no match of right hand side value: {:hello, "world"}
```

And also when comparing different types:

```
iex> {a, b, c} = [:hello, "world", "!"]
** (MatchError) no match of right hand side value: [:hello, "world", "!"]
```

More interestingly, we can match on specific values. The example below asserts that the left side will only match the right side when the right side is a tuple that starts with the atom `:ok`:

```
iex> {:ok, result} = {:ok, 13}
{:ok, 13}
iex> result
13

iex> {:ok, result} = {:error, :oops}
** (MatchError) no match of right hand side value: {:error, :oops}
```

We can pattern match on lists:

```
iex> [a, b, c] = [1, 2, 3]
[1, 2, 3]
iex> a
1
```

A list also supports matching on its own head and tail:

```
iex> [head | tail] = [1, 2, 3]
[1, 2, 3]
iex> head
1
iex> tail
[2, 3]
```

Similar to the `hd/1` and `tl/1` functions, we can't match an empty list with a head and tail pattern:

```
iex> [h|t] = []
** (MatchError) no match of right hand side value: []
```

The `[head | tail]` format is not only used on pattern matching but also for prepending items to a list:

```
iex> list = [1, 2, 3]
[1, 2, 3]
iex> [0|list]
[0, 1, 2, 3]
```

Pattern matching allows developers to easily destructure data types such as tuples and lists. As we will see in following chapters, it is one of the foundations of recursion in Elixir and applies to other types as well, like maps and binaries.

4.3 The pin operator

Variables in Elixir can be rebound:

```
iex> x = 1
1
iex> x = 2
2
```

The pin operator ^ can be used when there is no interest in rebinding a variable but rather in matching against its value prior to the match:

```
iex> x = 1
1
iex> ^x = 2
** (MatchError) no match of right hand side value: 2
iex> {x, ^x} = {2, 1}
{2, 1}
iex> x
2
```

Notice that if a variable is mentioned more than once in a pattern, all references should bind to the same pattern:

```
iex> {x, x} = {1, 1}
1
iex> {x, x} = {1, 2}
** (MatchError) no match of right hand side value: {1, 2}
```

In some cases, you don't care about a particular value in a pattern. It is a common practice to bind those values to the underscore, _. For example, if only the head of the list matters to us, we can assign the tail to underscore:

```
iex> [h | _] = [1, 2, 3]
[1, 2, 3]
iex> h
1
```

The variable _ is special in that it can never be read from. Trying to read from it gives an unbound variable error:

```
iex> _
** (CompileError) iex:1: unbound variable _
```

Although pattern matching allows us to build powerful constructs, its usage is limited. For instance, you cannot make function calls on the left side of a match. The following example is invalid:

```
iex> length([1,[2],3]) = 3
** (CompileError) iex:1: illegal pattern
```

This finishes our introduction to pattern matching. As we will see in the next chapter, pattern matching is very common in many language constructs.

5 case, cond and if

{% include toc.html %}

In this chapter, we will learn about the `case`, `cond` and `if` control-flow structures.

5.1 case

`case` allows us to compare a value against many patterns until we find a matching one:

```
iex> case {1, 2, 3} do
...>   {4, 5, 6} ->
...>     "This clause won't match"
...>   {1, x, 3} ->
...>     "This clause will match and bind x to 2 in this clause"
...>   _ ->
...>     "This clause would match any value"
...> end
```

If you want to pattern match against an existing variable, you need to use the ^ operator:

```
iex> x = 1
1
iex> case 10 do
...>   ^x -> "Won't match"
...>   _  -> "Will match"
...> end
```

Clauses also allow extra conditions to be specified via guards:

```
iex> case {1, 2, 3} do
...>   {1, x, 3} when x > 0 ->
...>     "Will match"
...>   _ ->
...>     "Won't match"
...> end
```

The first clause above will only match when `x` is positive.

5.2 Expressions in guard clauses.

The Erlang Virtual Machine (VM) only allows a limited set of expressions in guards:

- comparison operators (==, !=, ===, !==, >, <, <=, >=)
- boolean operators (`and`, `or`) and negation operators (`not`, `!`)
- arithmetic operators (+, -, *, /)
- <> and ++ as long as the left side is a literal
- the `in` operator
- all the following type check functions:
  - `is_atom/1`
  - `is_binary/1`
  - `is_bitstring/1`

    - is_boolean/1
    - is_float/1
    - is_function/1
    - is_function/2
    - is_integer/1
    - is_list/1
    - is_map/1
    - is_number/1
    - is_pid/1
    - is_port/1
    - is_reference/1
    - is_tuple/1
- plus these functions:
    - abs(number)
    - bit_size(bitstring)
    - byte_size(bitstring)
    - div(integer, integer)
    - elem(tuple, n)
    - hd(list)
    - length(list)
    - map_size(map)
    - node()
    - node(pid | ref | port)
    - rem(integer, integer)
    - round(number)
    - self()
    - tl(list)
    - trunc(number)
    - tuple_size(tuple)

Keep in mind errors in guards do not leak but simply make the guard fail:

```
iex> hd(1)
** (ArgumentError) argument error
    :erlang.hd(1)
iex> case 1 do
...>   x when hd(x) -> "Won't match"
...>   x -> "Got: #{x}"
...> end
"Got 1"
```

If none of the clauses match, an error is raised:

```
iex> case :ok do
...>   :error -> "Won't match"
...> end
** (CaseClauseError) no case clause matching: :ok
```

Note anonymous functions can also have multiple clauses and guards:

```
iex> f = fn
...>   x, y when x > 0 -> x + y
...>   x, y -> x * y
...> end
#Function<12.71889879/2 in :erl_eval.expr/5>
iex> f.(1, 3)
4
iex> f.(-1, 3)
-3
```

The number of arguments in each anonymous function clause needs to be the same, otherwise an error is raised.

5.3 cond

`case` is useful when you need to match against different values. However, in many circumstances, we want to check different conditions and find the first one that evaluates to true. In such cases, one may use `cond`:

```
iex> cond do
...>   2 + 2 == 5 ->
...>     "This will not be true"
...>   2 * 2 == 3 ->
...>     "Nor this"
...>   1 + 1 == 2 ->
...>     "But this will"
...> end
"But this will"
```

This is equivalent to `else if` clauses in many imperative languages (although used way less frequently here).

If none of the conditions return true, an error is raised. For this reason, it may be necessary to add a final condition, equal to `true`, which will always match:

```
iex> cond do
...>   2 + 2 == 5 ->
...>     "This is never true"
...>   2 * 2 == 3 ->
...>     "Nor this"
...>   true ->
...>     "This is always true (equivalent to else)"
...> end
```

Finally, note `cond` considers any value besides `nil` and `false` to be true:

```
iex> cond do
...>   hd([1,2,3]) ->
...>     "1 is considered as true"
...> end
"1 is considered as true"
```

5.4 if and unless

Besides `case` and `cond`, Elixir also provides the macros `if/2` and `unless/2` which are useful when you need to check for just one condition:

```
iex> if true do
...>   "This works!"
...> end
"This works!"
iex> unless true do
...>   "This will never be seen"
...> end
nil
```

If the condition given to `if/2` returns `false` or `nil`, the body given between `do/end` is not executed and it simply returns `nil`. The opposite happens with `unless/2`.

They also support `else` blocks:

```
iex> if nil do
...>   "This won't be seen"
...> else
...>   "This will"
...> end
"This will"

Note: An interesting note regarding ``if/2`` and ``unless/2`` is
that they are implemented as macros in the language; they aren't
special language constructs as they would be in many languages. You
can check the documentation and the source of ``if/2`` in `the
``Kernel`` module docs </docs/stable/elixir/Kernel.html>`__. The
``Kernel`` module is also where operators like ``+/2`` and functions
like ``is_function/2`` are defined, all automatically imported and
available in your code by default.
```

5.5 do/end blocks

At this point, we have learned four control structures: `case`, `cond`, `if` and `unless`, and they were all wrapped in `do/end` blocks. It happens we could also write `if` as follows:

```
iex> if true, do: 1 + 2
3
```

In Elixir, `do/end` blocks are a convenience for passing a group of expressions to `do:`. These are equivalent:

```
iex> if true do
...>   a = 1 + 2
...>   a + 10
...> end
13
iex> if true, do: (
...>   a = 1 + 2
...>   a + 10
...> )
13
```

We say the second syntax is using **keyword lists**. We can pass `else` using this syntax:

```
iex> if false, do: :this, else: :that
:that
```

One thing to keep in mind when using `do`/`end` blocks is they are always bound to the outermost function call. For example, the following expression:

```
iex> is_number if true do
...>  1 + 2
...> end
```

Would be parsed as:

```
iex> is_number(if true) do
...>  1 + 2
...> end
```

Which leads to an undefined function error as Elixir attempts to invoke `is_number/2`. Adding explicit parentheses is enough to resolve the ambiguity:

```
iex> is_number(if true do
...>  1 + 2
...> end)
true
```

Keyword lists play an important role in the language and are quite common in many functions and macros. We will explore them a bit more in a future chapter. Now it is time to talk about "Binaries, strings and char lists".

6 Binaries, strings and char lists

{% include toc.html %}

In "Basic types", we learned about strings and used the `is_binary/1` function for checks:

```
iex> string = "hello"
"hello"
iex> is_binary string
true
```

In this chapter, we will understand what binaries are, how they associate with strings, and what a single-quoted value, `'like this'`, means in Elixir.

6.1 UTF-8 and Unicode

A string is a UTF-8 encoded binary. In order to understand exactly what we mean by that, we need to understand the difference between bytes and code points.

The Unicode standard assigns code points to many of the characters we know. For example, the letter `a` has code point `97` while the letter `ł` has code point `322`. When writing the string `"hełło"` to disk, we need to convert this code point to bytes. If we adopted a rule that said one byte represents one code point, we wouldn't be able to write `"hełło"`, because it uses the code point `322` for `ł`, and one byte can only represent a number from `0` to `255`. But of course, given you can actually read `"hełło"` on your screen, it must be represented *somehow*. That's where encodings come in.

When representing code points in bytes, we need to encode them somehow. Elixir chose the UTF-8 encoding as its main and default encoding. When we say a string is a UTF-8 encoded binary, we mean a string is a bunch of bytes organized in a way to represent certain code points, as specified by the UTF-8 encoding.

Since we have code points like `ł` assigned to the number `322`, we actually need more than one byte to represent it. That's why we see a difference when we calculate the `byte_size/1` of a string compared to its `String.length/1`:

```
iex> string = "hełło"
"hełło"
iex> byte_size string
7
iex> String.length string
5
```

UTF-8 requires one byte to represent the code points `h`, `e` and `o`, but two bytes to represent `ł`. In Elixir, you can get a code point's value by using `?`:

```
iex> ?a
97
iex> ?ł
322
```

You can also use the functions in *the ''String'* module </docs/stable/elixir/String.html>'__ to split a string in its code points:

```
iex> String.codepoints("hełło")
["h", "e", "ł", "ł", "o"]
```

You will see that Elixir has excellent support for working with strings. It also supports many of the Unicode operations. In fact, Elixir passes all the tests showcased in the article "The string type is broken".

However, strings are just part of the story. If a string is a binary, and we have used the `is_binary/1` function, Elixir must have an underlying type empowering strings. And it does. Let's talk about binaries!

6.2 Binaries (and bitstrings)

In Elixir, you can define a binary using `<<>>`:

```
iex> <<0, 1, 2, 3>>
<<0, 1, 2, 3>>
iex> byte_size <<0, 1, 2, 3>>
4
```

A binary is just a sequence of bytes. Of course, those bytes can be organized in any way, even in a sequence that does not make them a valid string:

```
iex> String.valid?(<<239, 191, 191>>)
false
```

The string concatenation operation is actually a binary concatenation operator:

```
iex> <<0, 1>> <> <<2, 3>>
<<0, 1, 2, 3>>
```

A common trick in Elixir is to concatenate the null byte `<<0>>` to a string to see its inner binary representation:

```
iex> "hełło" <> <<0>>
<<104, 101, 197, 130, 197, 130, 111, 0>>
```

Each number given to a binary is meant to represent a byte and therefore must go up to 255. Binaries allow modifiers to be given to store numbers bigger than 255 or to convert a code point to its utf8 representation:

```
iex> <<255>>
<<255>>
iex> <<256>> # truncated
<<0>>
iex> <<256 :: size(16)>> # use 16 bits (2 bytes) to store the number
<<1, 0>>
iex> <<256 :: utf8>> # the number is a code point
"Ā"
iex> <<256 :: utf8, 0>>
<<196, 128, 0>>
```

If a byte has 8 bits, what happens if we pass a size of 1 bit?

```
iex> <<1 :: size(1)>>
<<1::size(1)>>
iex> <<2 :: size(1)>> # truncated
<<0::size(1)>>
iex> is_binary(<< 1 :: size(1)>>)
false
iex> is_bitstring(<< 1 :: size(1)>>)
true
iex> bit_size(<< 1 :: size(1)>>)
1
```

The value is no longer a binary, but a bitstring – just a bunch of bits! So a binary is a bitstring where the number of bits is divisible by 8!

We can also pattern match on binaries / bitstrings:

```
iex> <<0, 1, x>> = <<0, 1, 2>>
<<0, 1, 2>>
iex> x
2
iex> <<0, 1, x>> = <<0, 1, 2, 3>>
** (MatchError) no match of right hand side value: <<0, 1, 2, 3>>
```

Note each entry in the binary is expected to match exactly 8 bits. However, we can match on the rest of the binary modifier:

```
iex> <<0, 1, x :: binary>> = <<0, 1, 2, 3>>
<<0, 1, 2, 3>>
iex> x
<<2, 3>>
```

The pattern above only works if the binary is at the end of <<>>. Similar results can be achieved with the string concatenation operator <>:

```
iex> "he" <> rest = "hello"
"hello"
iex> rest
"llo"
```

This finishes our tour of bitstrings, binaries and strings. A string is a UTF-8 encoded binary, and a binary is a bitstring where the number of bits is divisible by 8. Although this shows the flexibility Elixir provides to work with bits and bytes, 99% of the time you will be working with binaries and using the `is_binary/1` and `byte_size/1` functions.

6.3 Char lists

A char list is nothing more than a list of characters:

```
iex> 'hełło'
[104, 101, 322, 322, 111]
iex> is_list 'hełło'
true
iex> 'hello'
'hello'
```

You can see that, instead of containing bytes, a char list contains the code points of the characters between single-quotes (note that iex will only output code points if any of the chars is outside the ASCII range). So while double-quotes represent a string (i.e. a binary), single-quotes represents a char list (i.e. a list).

In practice, char lists are used mostly when interfacing with Erlang, in particular old libraries that do not accept binaries as arguments. You can convert a char list to a string and back by using the `to_string/1` and `to_char_list/1` functions:

```
iex> to_char_list "hełło"
[104, 101, 322, 322, 111]
iex> to_string 'hełło'
"hełło"
iex> to_string :hello
"hello"
iex> to_string 1
"1"
```

Note that those functions are polymorphic. They not only convert char lists to strings, but also integers to strings, atoms to strings, and so on.

With binaries, strings, and char lists out of the way, it is time to talk about key-value data structures.

7 Keywords, maps and dicts

{% include toc.html %}

So far we haven't discussed any associative data structures, i.e. data structures that are able to associate a certain value (or multiple values) to a key. Different languages call these different names like dictionaries, hashes, associative arrays, maps, etc.

In Elixir, we have two main associative data structures: keyword lists and maps. It's time to learn more about them!

7.1 Keyword lists

In many functional programming languages, it is common to use a list of 2-item tuples as the representation of an associative data structure. In Elixir, when we have a list of tuples and the first item of the tuple (i.e. the key) is an atom, we call it a keyword list:

```
iex> list = [{:a, 1}, {:b, 2}]
[a: 1, b: 2]
iex> list == [a: 1, b: 2]
true
iex> list[:a]
1
```

As you can see above, Elixir supports a special syntax for defining such lists, and underneath they just map to a list of tuples. Since they are simply lists, all operations available to lists, including their performance characteristics, also apply to keyword lists.

For example, we can use ++ to add new values to a keyword list:

```
iex> list ++ [c: 3]
[a: 1, b: 2, c: 3]
iex> [a: 0] ++ list
[a: 0, a: 1, b: 2]
```

Note that values added to the front are the ones fetched on lookup:

```
iex> new_list = [a: 0] ++ list
[a: 0, a: 1, b: 2]
iex> new_list[:a]
0
```

Keyword lists are important because they have three special characteristics:

- Keys must be atoms.
- Keys are ordered, as specified by the developer.
- Keys can be given more than once.

For example, the Ecto library makes use of both features to provide an elegant DSL for writing database queries:

```
query = from w in Weather,
      where: w.prcp > 0,
      where: w.temp < 20,
     select: w
```

Those features are what prompted keyword lists to be the default mechanism for passing options to functions in Elixir. In chapter 5, when we discussed the `if/2` macro, we mentioned the following syntax is supported:

```
iex> if false, do: :this, else: :that
:that
```

The `do:` and `else:` pairs are keyword lists! In fact, the call above is equivalent to:

```
iex> if(false, [do: :this, else: :that])
:that
```

In general, when the keyword list is the last argument of a function, the square brackets are optional.

In order to manipulate keyword lists, Elixir provides *the ‘‘Keyword‘* module </docs/stable/elixir/Keyword.html>‘__. Remember though keyword lists are simply lists, and as such they provide the same linear performance characteristics as lists. The longer the list, the longer it will take to find a key, to count the number of items, and so on. For this reason, keyword lists are used in Elixir mainly as options. If you need to store many items or guarantee one-key associates with at maximum one-value, you should use maps instead.

Although we can pattern match on keyword lists, it is rarely done in practice since pattern matching on lists require the number of items and their order to match:

```
iex> [a: a] = [a: 1]
[a: 1]
iex> a
1
iex> [a: a] = [a: 1, b: 2]
```

```
** (MatchError) no match of right hand side value: [a: 1, b: 2]
iex> [b: b, a: a] = [a: 1, b: 2]
** (MatchError) no match of right hand side value: [a: 1, b: 2]
```

7.2 Maps

Whenever you need a key-value store, maps are the "go to" data structure in Elixir. A map is created using the `%{}` syntax:

```
iex> map = %{:a => 1, 2 => :b}
%{2 => :b, :a => 1}
iex> map[:a]
1
iex> map[2]
:b
iex> map[:c]
nil
```

Compared to keyword lists, we can already see two differences:

- Maps allow any value as a key.
- Maps' keys do not follow any ordering.

If you pass duplicate keys when creating a map, the last one wins:

```
iex> %{1 => 1, 1 => 2}
%{1 => 2}
```

When all the keys in a map are atoms, you can use the keyword syntax for convenience:

```
iex> map = %{a: 1, b: 2}
%{a: 1, b: 2}
```

In contrast to keyword lists, maps are very useful with pattern matching:

```
iex> %{} = %{:a => 1, 2 => :b}
%{:a => 1, 2 => :b}
iex> %{:a => a} = %{:a => 1, 2 => :b}
%{:a => 1, 2 => :b}
iex> a
1
iex> %{:c => c} = %{:a => 1, 2 => :b}
** (MatchError) no match of right hand side value: %{2 => :b, :a => 1}
```

As shown above, a map matches as long as the given keys exist in the given map. Therefore, an empty map matches all maps.

The ''Map' module </docs/stable/elixir/Map.html>'__ provides a very similar API to the `Keyword` module with convenience functions to manipulate maps:

```
iex> Map.get(%{:a => 1, 2 => :b}, :a)
1
iex> Map.to_list(%{:a => 1, 2 => :b})
[{2, :b}, {:a, 1}]
```

One interesting property about maps is that they provide a particular syntax for updating and accessing atom keys:

```
iex> map = %{:a => 1, 2 => :b}
%{:a => 1, 2 => :b}

iex> map.a
1
iex> map.c
** (KeyError) key :c not found in: %{2 => :b, :a => 1}

iex> %{map | :a => 2}
%{:a => 2, 2 => :b}
iex> %{map | :c => 3}
** (ArgumentError) argument error
```

Both access and update syntaxes above require the given keys to exist. For example, accessing and updating the `:c` key failed there is no `:c` in the map.

Elixir developers typically prefer to use the `map.field` syntax and pattern matching instead of the functions in the `Map` module when working with maps because they lead to an assertive style of programming. This blog post provides insight and examples on how you get more concise and faster software by writing assertive code in Elixir.

> Note: Maps were recently introduced into the Erlang VM with EEP 43. Erlang 17 provides a partial implementation of the EEP, where only "small maps" are supported. This means maps have good performance characteristics only when storing at maximum a couple of dozens keys. To fill in this gap, Elixir also provides *the ''HashDict'* module </docs/stable/elixir/HashDict.html>'__ which uses a hashing algorithm to provide a dictionary that supports hundreds of thousands keys with good performance.

7.3 Dicts

In Elixir, both keyword lists and maps are called dictionaries. In other words, a dictionary is like an interface (we call them behaviours in Elixir) and both keyword lists and maps modules implement this interface.

This interface is defined in the *the ''Dict'* module </docs/stable/elixir/Dict.html>'__ module which also provides an API that delegates to the underlying implementations:

```
iex> keyword = []
[]
iex> map = %{}
%{}
iex> Dict.put(keyword, :a, 1)
[a: 1]
iex> Dict.put(map, :a, 1)
%{a: 1}
```

The `Dict` module allows any developer to implement their own variation of `Dict`, with specific characteristics, and hook into existing Elixir code. The `Dict` module also provides functions that are meant to work across dictionaries. For example, `Dict.equal?/2` can compare two dictionaries of any kind.

That said, you may be wondering, which of `Keyword`, `Map` or `Dict` modules should you use in your code? The answer is: it depends.

If your code is expecting one specific data structure as argument, use the respective module as it leads to more assertive code. For example, if you expect a keyword as an argument, explicitly use the `Keyword` module instead of `Dict`. However, if your API is meant to work with any dictionary, use the `Dict` module (and make sure to write tests that pass different dict implementations as arguments).

This concludes our introduction to associative data structures in Elixir. You will find out that given keyword lists and maps, you will always have the right tool to tackle problems that require associative data structures in Elixir.

8 Modules

{% include toc.html %}

In Elixir we group several functions into modules. We've already used many different modules in the previous chapters like *the ''String'* module </docs/stable/elixir/String.html>'__:

```
iex> String.length "hello"
5
```

In order to create our own modules in Elixir, we use the `defmodule` macro. We use the `def` macro to define functions in that module:

```
iex> defmodule Math do
...>   def sum(a, b) do
...>     a + b
...>   end
...> end

iex> Math.sum(1, 2)
3
```

In the following sections, our examples are going to get a bit more complex, and it can be tricky to type them all in the shell. It's about time for us to learn how to compile Elixir code and also how to run Elixir scripts.

8.1 Compilation

Most of the time it is convenient to write modules into files so they can be compiled and reused. Let's assume we have a file named `math.ex` with the following contents:

```
defmodule Math do
  def sum(a, b) do
    a + b
  end
end
```

This file can be compiled using `elixirc`:

```
$ elixirc math.ex
```

This will generate a file named `Elixir.Math.beam` containing the bytecode for the defined module. If we start `iex` again, our module definition will be available (provided that `iex` is started in the same directory the bytecode file is in):

```
iex> Math.sum(1, 2)
3
```

Elixir projects are usually organized into three directories:

- ebin - contains the compiled bytecode
- lib - contains elixir code (usually `.ex` files)
- test - contains tests (usually `.exs` files)

When working on actual projects, the build tool called `mix` will be responsible for compiling and setting up the proper paths for you. For learning purposes, Elixir also supports a scripted mode which is more flexible and does not generate any compiled artifacts.

8.2 Scripted mode

In addition to the Elixir file extension `.ex`, Elixir also supports `.exs` files for scripting. Elixir treats both files exactly the same way, the only difference is in intention. `.ex` files are meant to be compiled while `.exs` files are used for scripting, without the need for compilation. For instance, we can create a file called `math.exs`:

```
defmodule Math do
  def sum(a, b) do
    a + b
  end
end

IO.puts Math.sum(1, 2)
```

And execute it as:

```
$ elixir math.exs
```

The file will be compiled in memory and executed, printing "3" as the result. No bytecode file will be created. In the following examples, we recommend you write your code into script files and execute them as shown above.

8.3 Named functions

Inside a module, we can define functions with `def/2` and private functions with `defp/2`. A function defined with `def/2` can be invoked from other modules while a private function can only be invoked locally.

```
defmodule Math do
  def sum(a, b) do
    do_sum(a, b)
  end

  defp do_sum(a, b) do
    a + b
  end
end

Math.sum(1, 2)     #=> 3
Math.do_sum(1, 2)  #=> ** (UndefinedFunctionError)
```

Function declarations also support guards and multiple clauses. If a function has several clauses, Elixir will try each clause until it finds one that matches. Here is an implementation of a function that checks if the given number is zero or not:

```
defmodule Math do
  def zero?(0) do
    true
  end

  def zero?(x) when is_number(x) do
    false
  end
end

Math.zero?(0)   #=> true
Math.zero?(1)   #=> false
```

```
Math.zero?([1,2,3])
#=> ** (FunctionClauseError)
```

Giving an argument that does not match any of the clauses raises an error.

8.4 Function capturing

Throughout this tutorial, we have been using the notation `name/arity` to refer to functions. It happens that this notation can actually be used to retrieve a named function as a function type. Let's start `iex` and run the `math.exs` file defined above:

```
$ iex math.exs
```

```
iex> Math.zero?(0)
true
iex> fun = &Math.zero?/1
&Math.zero?/1
iex> is_function fun
true
iex> fun.(0)
true
```

Local or imported functions, like `is_function/1`, can be captured without the module:

```
iex> &is_function/1
&:erlang.is_function/1
iex> (&is_function/1).(fun)
true
```

Note the capture syntax can also be used as a shortcut for creating functions:

```
iex> fun = &(&1 + 1)
#Function<6.71889879/1 in :erl_eval.expr/5>
iex> fun.(1)
2
```

The `&1` represents the first argument passed into the function. `&(&1+1)` above is exactly the same as `fn x -> x + 1 end`. The syntax above is useful for short function definitions.

In the same fashion if you want to call a function from a module, you can do `&Module.function()`:

```
iex> fun = &List.flatten(&1, &2)
&List.flatten/2
iex> fun.([1, [[2], 3]], [4, 5])
[1, 2, 3, 4, 5]
```

`&List.flatten(&1, &2)` is the same as writing `fn(list, tail) -> List.flatten(list, tail) end`. You can read more about the capture operator `&` in *the ‘‘Kernel.SpecialForms‘* documentation </docs/stable/elixir/Kernel.SpecialForms.html#&/1>‘__.

8.5 Default arguments

Named functions in Elixir also support default arguments:

```
defmodule Concat do
  def join(a, b, sep \\ " ") do
    a <> sep <> b
  end
end

IO.puts Concat.join("Hello", "world")      #=> Hello world
IO.puts Concat.join("Hello", "world", "_") #=> Hello_world
```

Any expression is allowed to serve as a default value, but it won't be evaluated during the function definition; it will simply be stored for later use. Every time the function is invoked and any of its default values have to be used, the expression for that default value will be evaluated:

```
defmodule DefaultTest do
  def dowork(x \\ IO.puts "hello") do
    x
  end
end
```

```
iex> DefaultTest.dowork
hello
:ok
iex> DefaultTest.dowork 123
123
iex> DefaultTest.dowork
hello
:ok
```

If a function with default values has multiple clauses, it is recommended to create a function head (without an actual body), just for declaring defaults:

```
defmodule Concat do
  def join(a, b \\ nil, sep \\ " ")

  def join(a, b, _sep) when is_nil(b) do
    a
  end

  def join(a, b, sep) do
    a <> sep <> b
  end
end

IO.puts Concat.join("Hello", "world")      #=> Hello world
IO.puts Concat.join("Hello", "world", "_") #=> Hello_world
IO.puts Concat.join("Hello")               #=> Hello
```

When using default values, one must be careful to avoid overlapping function definitions. Consider the following example:

```
defmodule Concat do
  def join(a, b) do
    IO.puts "***First join"
    a <> b
  end

  def join(a, b, sep \\ " ") do
```

```
    IO.puts "***Second join"
    a <> sep <> b
  end
end
```

If we save the code above in a file named "concat.ex" and compile it, Elixir will emit the following warning:

```
concat.ex:7: this clause cannot match because a previous clause at line 2 always␣
↪matches
```

The compiler is telling us that invoking the `join` function with two arguments will always choose the first definition of `join` whereas the second one will only be invoked when three arguments are passed:

```
$ iex concat.exs
```

```
iex> Concat.join "Hello", "world"
***First join
"Helloworld"
```

```
iex> Concat.join "Hello", "world", "_"
***Second join
"Hello_world"
```

This finishes our short introduction to modules. In the next chapters, we will learn how to use named functions for recursion, explore Elixir lexical directives that can be used for importing functions from other modules and discuss module attributes.

9 Recursion

{% include toc.html %}

9.1 Loops through recursion

Due to immutability, loops in Elixir (and in any functional programming language) are written differently from imperative languages. For example, in an imperative language (like C in the following example), one would write:

```
for(i = 0; i < array.length; i++) {
  array[i] = array[i] * 2
}
```

In the example above, we are mutating both the array and the variable `i`. Mutating is not possible in Elixir. Instead, functional languages rely on recursion: a function is called recursively until a condition is reached that stops the recursive action from continuing. No data is mutated in this process. Consider the example below that prints a string an arbitrary amount of times:

```
defmodule Recursion do
  def print_multiple_times(msg, n) when n <= 1 do
    IO.puts msg
  end

  def print_multiple_times(msg, n) do
    IO.puts msg
    print_multiple_times(msg, n - 1)
  end
```

```
end

Recursion.print_multiple_times("Hello!", 3)
# Hello!
# Hello!
# Hello!
```

Similar to `case`, a function may have many clauses. A particular clause is executed when the arguments passed to the function match the clause's argument patterns and its guard evaluates to `true`.

When `print_multiple_times/2` is initially called in the example above, the argument `n` is equal to `3`.

The first clause has a guard which says "use this definition if and only if `n` is less than or equal to `1`". Since this is not the case, Elixir proceeds to the next clause's definition.

The second definition matches the pattern and has no guard so it will be executed. It first prints our `msg` and then calls itself passing `n - 1` (`2`) as the second argument.

Our `msg` is printed and `print_multiple_times/2` is called again this time with the second argument set to `1`. Because `n` is now set to `1`, the guard in our first definition of `print_multiple_times/2` evaluates to true, and we execute this particular definition. The `msg` is printed, and there is nothing left to execute.

We defined `print_multiple_times/2` so that no matter what number is passed as the second argument it either triggers our first definition (known as a "base case") or it triggers our second definition which will ensure that we get exactly one step closer to our base case.

9.2 "Reduce" and "map" algorithms

Let's now see how we can use the power of recursion to sum a list of numbers:

```
defmodule Math do
  def sum_list([head|tail], accumulator) do
    sum_list(tail, head + accumulator)
  end

  def sum_list([], accumulator) do
    accumulator
  end
end

Math.sum_list([1, 2, 3], 0) #=> 6
```

We invoke `sum_list` with the list `[1, 2, 3]` and the initial value `0` as arguments. We will try each clause until we find one that matches according to the pattern matching rules. In this case, the list `[1, 2, 3]` matches against `[head|tail]` which bounds `head` to `1` and `tail` to `[2, 3]`; `accumulator` is set to `0`.

Then, we add the head of the list to the accumulator `head + accumulator` and call `sum_list` again, recursively, passing the tail of the list as its first argument. The tail will once again match `[head|tail]` until the list is empty, as seen below:

```
sum_list [1, 2, 3], 0
sum_list [2, 3], 1
sum_list [3], 3
sum_list [], 6
```

When the list is empty, it will match the final clause which returns the final result of `6`.

The process of taking a list and "reducing" it down to one value is known as a "reduce" algorithm and is central to functional programming.

What if we instead want to double all of the values in our list?

```
defmodule Math do
  def double_each([head|tail]) do
    [head * 2|double_each(tail)]
  end

  def double_each([]) do
    []
  end
end

Math.double_each([1, 2, 3]) #=> [2, 4, 6]
```

Here we have used recursion to traverse a list doubling each element and returning a new list. The process of taking a list and "mapping" over it is known as a "map" algorithm.

Recursion and tail call optimization are an important part of Elixir and are commonly used to create loops. However, when programming in Elixir you will rarely use recursion as above to manipulate lists.

The `Enum module </docs/stable/elixir/Enum.html>`__, which we're going to see in the next chapter, already provides many conveniences for working with lists. For instance, the examples above could be written as:

```
iex> Enum.reduce([1, 2, 3], 0, fn(x, acc) -> x + acc end)
6
iex> Enum.map([1, 2, 3], fn(x) -> x * 2 end)
[2, 4, 6]
```

Or, using the capture syntax:

```
iex> Enum.reduce([1, 2, 3], 0, &+/2)
6
iex> Enum.map([1, 2, 3], &(&1 * 2))
[2, 4, 6]
```

Let's take a deeper look at `Enumerables` and, while we're at it, their lazy counterpart, `Streams`.

10 Enumerables and Streams

{% include toc.html %}

10.1 Enumerables

Elixir provides the concept of enumerables and *the ``Enum`* module </docs/stable/elixir/Enum.html>`__ to work with them. We have already learned two enumerables: lists and maps.

```
iex> Enum.map([1, 2, 3], fn x -> x * 2 end)
[2, 4, 6]
iex> Enum.map(%{1 => 2, 3 => 4}, fn {k, v} -> k * v end)
[2, 12]
```

The `Enum` module provides a huge range of functions to transform, sort, group, filter and retrieve items from enumerables. It is one of the modules developers use frequently in their Elixir code.

Elixir also provides ranges:

```
iex> Enum.map(1..3, fn x -> x * 2 end)
[2, 4, 6]
iex> Enum.reduce(1..3, 0, &+/2)
6
```

Since the Enum module was designed to work across different data types, its API is limited to functions that are useful across many data types. For specific operations, you may need to reach to modules specific to the data types. For example, if you want to insert an element at a given position in a list, you should use the `List.insert_at/3` function from *the ‘‘List‘* module </docs/stable/elixir/List.html>‘__, as it would make little sense to insert a value into, for example, a range.

We say the functions in the `Enum` module are polymorphic because they can work with diverse data types. In particular, the functions in the `Enum` module can work with any data type that implements *the ‘‘Enumerable‘* protocol </docs/stable/elixir/Enumerable.html>‘__. We are going to discuss Protocols in a later chapter, for now we are going to move on to a specific kind of enumerable called streams.

10.2 Eager vs Lazy

All the functions in the `Enum` module are eager. Many functions expect an enumerable and return a list back:

```
iex> odd? = &(rem(&1, 2) != 0)
#Function<6.80484245/1 in :erl_eval.expr/5>
iex> Enum.filter(1..3, odd?)
[1, 3]
```

This means that when performing multiple operations with `Enum`, each operation is going to generate an intermediate list until we reach the result:

```
iex> 1..100_000 |> Enum.map(&(&1 * 3)) |> Enum.filter(odd?) |> Enum.sum
7500000000
```

The example above has a pipeline of operations. We start with a range and then multiply each element in the range by 3. This first operation will now create and return a list with `100_000` items. Then we keep all odd elements from the list, generating a new list, now with `50_000` items, and then we sum all entries.

10.2.1 The pipe operator

The `|>` symbol used in the snippet above is the **pipe operator**: it simply takes the output from the expression on its left side and passes it as the input to the function call on its right side. It’s similar to the Unix `|` operator. Its purpose is to highlight the flow of data being transformed by a series of functions. To see how it can make the code cleaner, have a look at the example above rewritten without using the `|>` operator:

```
iex> Enum.sum(Enum.filter(Enum.map(1..100_000, &(&1 * 3)), odd?))
7500000000
```

Find more about the pipe operator **‘by reading its documentation <http://elixir-lang.org/docs/stable/elixir/Kernel.html#|>/2>‘__**.

10.3 Streams

As an alternative to `Enum`, Elixir provides *the ‘‘Stream‘* module </docs/stable/elixir/Stream.html>‘__ which supports lazy operations:

```
iex> 1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(odd?) |> Enum.sum
7500000000
```

Streams are lazy, composable enumerables. Instead of generating intermediate lists, streams create a series of computations that are invoked only when we pass it to the `Enum` module. Streams are useful when working with large, *possibly infinite*, collections.

They are lazy because, as shown in the example above, `1..100_000 |> Stream.map(&(&1 * 3))` returns a data type, an actual stream, that represents the `map` computation over the range `1..100_000`:

```
iex> 1..100_000 |> Stream.map(&(&1 * 3))
#Stream<[enum: 1..100000, funs: [#Function<34.16982430/1 in Stream.map/2>]]>
```

Furthermore, they are composable because we can pipe many stream operations:

```
iex> 1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(odd?)
#Stream<[enum: 1..100000, funs: [...]]>
```

Many functions in the `Stream` module accept any enumerable as argument and return a stream as result. It also provides functions for creating streams, possibly infinite. For example, `Stream.cycle/1` can be used to create a stream that cycles a given enumerable infinitely. Be careful to not call a function like `Enum.map/2` on such streams, as they would cycle forever:

```
iex> stream = Stream.cycle([1, 2, 3])
#Function<15.16982430/2 in Stream.cycle/1>
iex> Enum.take(stream, 10)
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1]
```

On the other hand, `Stream.unfold/2` can be used to generate values from a given initial value:

```
iex> stream = Stream.unfold("hełło", &String.next_codepoint/1)
#Function<39.75994740/2 in Stream.unfold/2>
iex> Enum.take(stream, 3)
["h", "e", "ł"]
```

Another interesting function is `Stream.resource/3` which can be used to wrap around resources, guaranteeing they are opened right before enumeration and closed afterwards, even in case of failures. For example, we can use it to stream a file:

```
iex> stream = File.stream!("path/to/file")
#Function<18.16982430/2 in Stream.resource/3>
iex> Enum.take(stream, 10)
```

The example above will fetch the first 10 lines of the file you have selected. This means streams can be very useful for handling large files or even slow resources like network resources.

The amount of functions and functionality in `Enum </docs/stable/elixir/Enum.html>`__ and `Stream </docs/stable/elixir/Stream.html>`__ modules can be daunting at first but you will get familiar with them case by case. In particular, focus on the `Enum` module first and only move to `Stream` for the particular scenarios where laziness is required to either deal with slow resources or large, possibly infinite, collections.

Next we'll look at a feature central to Elixir, Processes, which allows us to write concurrent, parallel and distributed programs in an easy and understandable way.

11 Processes

{% include toc.html %}

In Elixir, all code runs inside processes. Processes are isolated from each other, run concurrent to one another and communicate via message passing. Processes are not only the basis for concurrency in Elixir, but they also provide the means for building distributed and fault-tolerant programs.

Elixir's processes should not be confused with operating system processes. Processes in Elixir are extremely lightweight in terms of memory and CPU (unlike threads in many other programming languages). Because of this, it is not uncommon to have dozens of thousands of processes running simultaneously.

In this chapter, we will learn about the basic constructs for spawning new processes, as well as sending and receiving messages between different processes.

11.1 spawn

The basic mechanism for spawning new processes is with the auto-imported `spawn/1` function:

```
iex> spawn fn -> 1 + 2 end
#PID<0.43.0>
```

`spawn/1` takes a function which it will execute in another process.

Notice `spawn/1` returns a PID (process identifier). At this point, the process you spawned is very likely dead. The spawned process will execute the given function and exit after the function is done:

```
iex> pid = spawn fn -> 1 + 2 end
#PID<0.44.0>
iex> Process.alive?(pid)
false

Note: you will likely get different process identifiers than the
ones we are getting in this guide.
```

We can retrieve the PID of the current process by calling `self/0`:

```
iex> self()
#PID<0.41.0>
iex> Process.alive?(self())
true
```

Processes get much more interesting when we are able to send and receive messages.

11.2 send and receive

We can send messages to a process with `send/2` and receive them with `receive/1`:

```
iex> send self(), {:hello, "world"}
{:hello, "world"}
iex> receive do
...>   {:hello, msg} -> msg
...>   {:world, msg} -> "won't match"
...> end
"world"
```

When a message is sent to a process, the message is stored in the process mailbox. The `receive/1` block goes through the current process mailbox searching for a message that matches any of the given patterns. `receive/1` supports many clauses, like `case/2`, as well as guards in the clauses.

If there is no message in the mailbox matching any of the patterns, the current process will wait until a matching message arrives. A timeout can also be specified:

```
iex> receive do
...>   {:hello, msg}  -> msg
...> after
...>   1_000 -> "nothing after 1s"
...> end
"nothing after 1s"
```

A timeout of 0 can be given when you already expect the message to be in the mailbox.

Let's put all together and send messages between processes:

```
iex> parent = self()
#PID<0.41.0>
iex> spawn fn -> send(parent, {:hello, self()}) end
#PID<0.48.0>
iex> receive do
...>   {:hello, pid} -> "Got hello from #{inspect pid}"
...> end
"Got hello from #PID<0.48.0>"
```

While in the shell, you may find the helper `flush/0` quite useful. It flushes and prints all the messages in the mailbox.

```
iex> send self(), :hello
:hello
iex> flush()
:hello
:ok
```

11.3 Links

The most common form of spawning in Elixir is actually via `spawn_link/1`. Before we show an example with `spawn_link/1`, let's try to see what happens when a process fails:

```
iex> spawn fn -> raise "oops" end
#PID<0.58.0>

[error] Error in process <0.58.0> with exit value: ...
```

It merely logged an error but the spawning process is still running. That's because processes are isolated. If we want the failure in one process to propagate to another one, we should link them. This can be done with `spawn_link/1`:

```
iex> spawn_link fn -> raise "oops" end
#PID<0.41.0>

** (EXIT from #PID<0.41.0>) an exception was raised:
    ** (RuntimeError) oops
        :erlang.apply/2
```

When a failure happens in the shell, the shell automatically traps the failure and shows it nicely formatted. In order to understand what would really happen in our code, let's use `spawn_link/1` inside a file and run it:

```
# spawn.exs
spawn_link fn -> raise "oops" end
```

```
receive do
  :hello -> "let's wait until the process fails"
end
```

This time the process failed and brought the parent process down as they are linked. Linking can also be done manually by calling `Process.link/1`. We recommend you to take a look at *the ‘‘Process‘* module </docs/stable/elixir/Process.html>‘__ for other functionality provided by processes.

Process and links play an important role when building fault-tolerant systems. In Elixir applications, we often link our processes to supervisors which will detect when a process dies and start a new process in its place. This is only possible because processes are isolated and don't share anything by default. And if processes are isolated, there is no way a failure in a process will crash or corrupt the state of another.

While other languages would require us to catch/handle exceptions, in Elixir we are actually fine with letting processes fail because we expect supervisors to properly restart our systems. "Failing fast" is a common philosophy when writing Elixir software!

Before moving to the next chapter, let's see one of the most common use cases for creating processes in Elixir.

11.4 Tasks

When making our processes crash in the previous section, you may have noticed the error messages were rather poor:

```
iex> spawn fn -> raise "oops" end
#PID<0.58.0>

[error] Error in process <0.58.0> with exit value: ...
```

With `spawn/1` and `spawn_link/1` functions, the error messages are generated directly by the Virtual Machine and therefore compact and lacking in details. In practice, developers would rather use the functions in the Task module, more explicitly, `Task.start/1` and `Task.start_link/1`:

```
iex(1)> Task.start fn -> raise "oops" end
{:ok, #PID<0.55.0>}

15:22:33.046 [error] Task #PID<0.55.0> started from #PID<0.53.0> terminating
Function: #Function<20.90072148/0 in :erl_eval.expr/5>
    Args: []
** (exit) an exception was raised:
    ** (RuntimeError) oops
        (elixir) lib/task/supervised.ex:74: Task.Supervised.do_apply/2
        (stdlib) proc_lib.erl:239: :proc_lib.init_p_do_apply/3
```

Besides providing better error logging, there are a couple other differences: `start/1` and `start_link/1` return `{:ok, pid}` rather than just the PID. This is what enables Tasks to be used in supervision tree. Furthermore, tasks provides convenience functions, like `Task.async/1` and `Task.await/1`, and functionality to ease distribution.

We will explore those functionalities in the ***Mix and OTP guide***, for now it is enough to remember to use Tasks to get better logging.

11.5 State

We haven't talked about state so far in this guide. If you are building an application that requires state, for example, to keep your application configuration, or you need to parse a file and keep it in memory, where would you store it?

Processes are the most common answer to this question. We can write processes that loop infinitely, maintain state, and send and receive messages. As an example, let's write a module that starts new processes that work as a key-value store in a file named `kv.exs`:

```
defmodule KV do
  def start_link do
    Task.start_link(fn -> loop(%{}) end)
  end

  defp loop(map) do
    receive do
      {:get, key, caller} ->
        send caller, Map.get(map, key)
        loop(map)
      {:put, key, value} ->
        loop(Map.put(map, key, value))
    end
  end
end
```

Note that the `start_link` function basically spawns a new process that runs the `loop/1` function, starting with an empty map. The `loop/1` function then waits for messages and performs the appropriate action for each message. In case of a `:get` message, it sends a message back to the caller and calls `loop/1` again, to wait for a new message. While the `:put` message actually invokes `loop/1` with a new version of the map, with the given `key` and `value` stored.

Let's give it a try by running `iex kv.exs`:

```
iex> {:ok, pid} = KV.start_link
#PID<0.62.0>
iex> send pid, {:get, :hello, self()}
{:get, :hello, #PID<0.41.0>}
iex> flush
nil
```

At first, the process map has no keys, so sending a `:get` message and then flushing the current process inbox returns `nil`. Let's send a `:put` message and try it again:

```
iex> send pid, {:put, :hello, :world}
#PID<0.62.0>
iex> send pid, {:get, :hello, self()}
{:get, :hello, #PID<0.41.0>}
iex> flush
:world
```

Notice how the process is keeping a state and we can get and update this state by sending the process messages. In fact, any process that knows the `pid` above will be able to send it messages and manipulate the state.

It is also possible to register the `pid`, giving it a name, and allowing everyone that knows the name to send it messages:

```
iex> Process.register(pid, :kv)
true
iex> send :kv, {:get, :hello, self()}
{:get, :hello, #PID<0.41.0>}
iex> flush
:world
```

Using processes around state and name registering are very common patterns in Elixir applications. However, most of the time, we won't implement those patterns manually as above, but by using one of the many of the abstractions that

ships with Elixir. For example, Elixir provides agents which are simple abstractions around state:

```
iex> {:ok, pid} = Agent.start_link(fn -> %{} end)
{:ok, #PID<0.72.0>}
iex> Agent.update(pid, fn map -> Map.put(map, :hello, :world) end)
:ok
iex> Agent.get(pid, fn map -> Map.get(map, :hello) end)
:world
```

A `:name` option could also be given to `Agent.start_link/2` and it would be automatically registered. Besides agents, Elixir provides an API for building generic servers (called GenServer), generic event managers and event handlers (called GenEvent), tasks and more, all powered by processes underneath. Those, along with supervision trees, will be explored with more detail in the ***Mix and OTP guide*** which will build a complete Elixir application from start to finish.

For now, let's move on and explore the world of I/O in Elixir.

12 IO and the file system

{% include toc.html %}

This chapter is a quick introduction to input/output mechanisms and file-system-related tasks, as well as to related modules like `IO </docs/stable/elixir/IO.html>`__, `File </docs/stable/elixir/File.html>`__ and `Path </docs/stable/elixir/Path.html>`__.

We had originally sketched this chapter to come much earlier in the getting started guide. However, we noticed the IO system provides a great opportunity to shed some light on some philosophies and curiosities of Elixir and the VM.

12.1 The `IO` module

The `IO` module is the main mechanism in Elixir for reading and writing to standard input/output (`:stdio`), standard error (`:stderr`), files and other IO devices. Usage of the module is pretty straightforward:

```
iex> IO.puts "hello world"
hello world
:ok
iex> IO.gets "yes or no? "
yes or no? yes
"yes\n"
```

By default, functions in the IO module read from the standard input and write to the standard output. We can change that by passing, for example, `:stderr` as an argument (in order to write to the standard error device):

```
iex> IO.puts :stderr, "hello world"
hello world
:ok
```

12.2 The `File` module

The `File </docs/stable/elixir/File.html>`__ module contains functions that allow us to open files as IO devices. By default, files are opened in binary mode, which requires developers to use the specific `IO.binread/2` and `IO.binwrite/2` functions from the `IO` module:

```
iex> {:ok, file} = File.open "hello", [:write]
{:ok, #PID<0.47.0>}
iex> IO.binwrite file, "world"
:ok
iex> File.close file
:ok
iex> File.read "hello"
{:ok, "world"}
```

A file can also be opened with `:utf8` encoding, which tells the `File` module to interpret the bytes read from the file as UTF-8-encoded bytes.

Besides functions for opening, reading and writing files, the `File` module has many functions to work with the file system. Those functions are named after their UNIX equivalents. For example, `File.rm/1` can be used to remove files, `File.mkdir/1` to create directories, `File.mkdir_p/1` to create directories and all their parent chain. There are even `File.cp_r/2` and `File.rm_rf/2` to respectively copy and remove files and directories recursively (i.e., copying and removing the contents of the directories too).

You will also notice that functions in the `File` module have two variants: one "regular" variant and another variant which has the same name as the regular version but with a trailing bang (`!`). For example, when we read the `"hello"` file in the example above, we use `File.read/1`. Alternatively, we can use `File.read!/1`:

```
iex> File.read "hello"
{:ok, "world"}
iex> File.read! "hello"
"world"
iex> File.read "unknown"
{:error, :enoent}
iex> File.read! "unknown"
** (File.Error) could not read file unknown: no such file or directory
```

Notice that when the file does not exist, the version with `!` raises an error. The version without `!` is preferred when you want to handle different outcomes using pattern matching:

```
case File.read(file) do
  {:ok, body}      -> # do something with the `body`
  {:error, reason} -> # handle the error caused by `reason`
end
```

However, if you expect the file to be there, the bang variation is more useful as it raises a meaningful error message. Avoid writing:

```
{:ok, body} = File.read(file)
```

as, in case of an error, `File.read/1` will return `{:error, reason}` and the pattern matching will fail. You will still get the desired result (a raised error), but the message will be about the pattern which doesn't match (thus being cryptic in respect to what the error actually is about).

If you don't want to handle a possible error (i.e., you want it to bubble up), prefer using `File.read!/1`.

12.3 The Path module

The majority of the functions in the `File` module expect paths as arguments. Most commonly, those paths will be regular binaries. The `Path </docs/stable/elixir/Path.html>`__ module provides facilities for working with such paths:

```
iex> Path.join("foo", "bar")
"foo/bar"
iex> Path.expand("~/hello")
"/Users/jose/hello"
```

Using functions from the `Path` module as opposed to just manipulating binaries is preferred since the `Path` module takes care of different operating systems transparently. For example, `Path.join/2` joins a path with slashes (`/`) on Unix-like systems and with backslashes (`\\`) on Windows.

With this we have covered the main modules that Elixir provides for dealing with IO and interacting with the file system. In the next sections, we will discuss some advanced topics regarding IO. Those sections are not necessary in order to write Elixir code, so feel free to skip them, but they do provide a nice overview of how the IO system is implemented in the VM and other curiosities.

12.4 Processes and group leaders

You may have noticed that `File.open/2` returns a tuple like `{:ok, pid}`:

```
iex> {:ok, file} = File.open "hello", [:write]
{:ok, #PID<0.47.0>}
```

That happens because the `IO` module actually works with processes (see chapter 11). When you write `IO.write(pid, binary)`, the `IO` module will send a message to the process identified by `pid` with the desired operation. Let's see what happens if we use our own process:

```
iex> pid = spawn fn ->
...>   receive do: (msg -> IO.inspect msg)
...> end
#PID<0.57.0>
iex> IO.write(pid, "hello")
{:io_request, #PID<0.41.0>, #PID<0.57.0>, {:put_chars, :unicode, "hello"}}
** (ErlangError) erlang error: :terminated
```

After `IO.write/2`, we can see the request sent by the `IO` module (a four-elements tuple) printed out. Soon after that, we see that it fails since the `IO` module expected some kind of result that we did not supply.

The `StringIO </docs/stable/elixir/StringIO.html>`__ module provides an implementation of the `IO` device messages on top of strings:

```
iex> {:ok, pid} = StringIO.open("hello")
{:ok, #PID<0.43.0>}
iex> IO.read(pid, 2)
"he"
```

By modelling IO devices with processes, the Erlang VM allows different nodes in the same network to exchange file processes in order to read/write files in between nodes. Of all IO devices, there is one that is special to each process: the **group leader**.

When you write to `:stdio`, you are actually sending a message to the group leader, which writes to the standard-input file descriptor:

```
iex> IO.puts :stdio, "hello"
hello
:ok
iex> IO.puts Process.group_leader, "hello"
hello
:ok
```

The group leader can be configured per process and is used in different situations. For example, when executing code in a remote terminal, it guarantees messages in a remote node are redirected and printed in the terminal that triggered the request.

12.5 iodata and chardata

In all of the examples above, we used binaries when writing to files. In the chapter "Binaries, strings and char lists", we mentioned how strings are simply bytes while char lists are lists with code points.

The functions in `IO` and `File` also allow lists to be given as arguments. Not only that, they also allow a mixed list of lists, integers and binaries to be given:

```
iex> IO.puts 'hello world'
hello world
:ok
iex> IO.puts ['hello', ?\s, "world"]
hello world
:ok
```

However, this requires some attention. A list may represent either a bunch of bytes or a bunch of characters and which one to use depends on the encoding of the IO device. If the file is opened without encoding, the file is expected to be in raw mode, and the functions in the `IO` module starting with `bin*` must be used. Those functions expect an `iodata` as argument; i.e., they expect a list of integers representing bytes and binaries to be given.

On the other hand, `:stdio` and files opened with `:utf8` encoding work with the remaining functions in the `IO` module. Those functions expect a `char_data` as argument, that is, a list of characters or strings.

Although this is a subtle difference, you only need to worry about those details if you intend to pass lists to those functions. Binaries are already represented by the underlying bytes and as such their representation is always raw.

This finishes our tour of IO devices and IO related functionality. We have learned about four Elixir modules - `IO </docs/stable/elixir/IO.html>`__, `File </docs/stable/elixir/File.html>`__, `Path </docs/stable/elixir/Path.html>`__ and `StringIO </docs/stable/elixir/StringIO.html>`__ - as well as how the VM uses processes for the underlying IO mechanisms and how to use `chardata` and `iodata` for IO operations.

13 alias, require and import

{% include toc.html %}

In order to facilitate software reuse, Elixir provides three directives. As we are going to see below, they are called directives because they have **lexical scope**.

13.1 alias

`alias` allows you to set up aliases for any given module name. Imagine our `Math` module uses a special list implementation for doing math specific operations:

```
defmodule Math do
  alias Math.List, as: List
end
```

From now on, any reference to `List` will automatically expand to `Math.List`. In case one wants to access the original `List`, it can be done by prefixing the module name with `Elixir.`:

```
List.flatten             #=> uses Math.List.flatten
Elixir.List.flatten      #=> uses List.flatten
Elixir.Math.List.flatten #=> uses Math.List.flatten

Note: All modules defined in Elixir are defined inside a main Elixir
namespace. However, for convenience, you can omit "Elixir." when
referencing them.
```

Aliases are frequently used to define shortcuts. In fact, calling `alias` without an `:as` option sets the alias automatically to the last part of the module name, for example:

```
alias Math.List
```

Is the same as:

```
alias Math.List, as: List
```

Note that `alias` is **lexically scoped**, which allows you to set aliases inside specific functions:

```
defmodule Math do
  def plus(a, b) do
    alias Math.List
    # ...
  end

  def minus(a, b) do
    # ...
  end
end
```

In the example above, since we are invoking `alias` inside the function `plus/2`, the alias will just be valid inside the function `plus/2`. `minus/2` won't be affected at all.

13.2 require

Elixir provides macros as a mechanism for meta-programming (writing code that generates code).

Macros are chunks of code that are executed and expanded at compilation time. This means, in order to use a macro, we need to guarantee its module and implementation are available during compilation. This is done with the `require` directive:

```
iex> Integer.is_odd(3)
** (CompileError) iex:1: you must require Integer before invoking the macro Integer.
↪is_odd/1
iex> require Integer
nil
iex> Integer.is_odd(3)
true
```

In Elixir, `Integer.is_odd/1` is defined as a macro so that it can be used as a guard. This means that, in order to invoke `Integer.is_odd/1`, we need to first require the `Integer` module.

In general a module does not need to be required before usage, except if we want to use the macros available in that module. An attempt to call a macro that was not loaded will raise an error. Note that like the `alias` directive, `require` is also lexically scoped. We will talk more about macros in a later chapter.

13.3 import

We use `import` whenever we want to easily access functions or macros from other modules without using the fully-qualified name. For instance, if we want to use the `duplicate/2` function from the `List` module several times, we can simply import it:

```
iex> import List, only: [duplicate: 2]
nil
iex> duplicate :ok, 3
[:ok, :ok, :ok]
```

In this case, we are importing only the function `duplicate` (with arity 2) from `List`. Although `:only` is optional, its usage is recommended in order to avoid importing all the functions of a given module inside the namespace. `:except` could also be given as an option in order to import everything in a module *except* a list of functions.

`import` also supports `:macros` and `:functions` to be given to `:only`. For example, to import all macros, one could write:

```
import Integer, only: :macros
```

Or to import all functions, you could write:

```
import Integer, only: :functions
```

Note that `import` is **lexically scoped** too. This means that we can import specific macros or functions inside function definitions:

```
defmodule Math do
  def some_function do
    import List, only: [duplicate: 2]
    duplicate(:ok, 10)
  end
end
```

In the example above, the imported `List.duplicate/2` is only visible within that specific function. `duplicate/2` won't be available in any other function in that module (or any other module for that matter).

Note that `import`ing a module automatically `requires` it.

13.4 Aliases

At this point you may be wondering: what exactly an Elixir alias is and how is it represented?

An alias in Elixir is a capitalized identifier (like `String`, `Keyword`, etc) which is converted to an atom during compilation. For instance, the `String` alias translates by default to the atom `:"Elixir.String"`:

```
iex> is_atom(String)
true
iex> to_string(String)
"Elixir.String"
iex> :"Elixir.String" == String
true
```

By using the `alias/2` directive, we are simply changing what an alias translates to.

Aliases work as described because in the Erlang VM (and consequently Elixir) modules are represented by atoms. For example, that's the mechanism we use to call Erlang modules:

```
iex> :lists.flatten([1, [2], 3])
[1, 2, 3]
```

This is also the mechanism that allows us to dynamically call a given function in a module:

```
iex> mod = :lists
:lists
iex> mod.flatten([1, [2], 3])
[1, 2, 3]
```

We are simply calling the function `flatten` on the atom `:lists`.

13.5 Nesting

Now that we have talked about aliases, we can talk about nesting and how it works in Elixir. Consider the following example:

```
defmodule Foo do
  defmodule Bar do
  end
end
```

The example above will define two modules: `Foo` and `Foo.Bar`. The second can be accessed as `Bar` inside `Foo` as long as they are in the same lexical scope.

If later the `Bar` module is moved outside the `Foo` module definition, it will need to be referenced by its full name (`Foo.Bar`) or an alias will need to be set using the `alias` directive discussed above. The `Bar` module definition will change too. This code is equivalent to the example above:

```
defmodule Foo.Bar do
end

defmodule Foo do
  alias Foo.Bar, as: Bar
end
```

The code above is exactly the same as:

```
defmodule Elixir.Foo do
  defmodule Elixir.Foo.Bar do
  end
  alias Elixir.Foo.Bar, as: Bar
end
```

Note: in Elixir, you don't have to define the `Foo` module before being able to define the `Foo.Bar` module, as the language translates all module names to atoms anyway. You can define arbitrarily-nested modules without defining any module in the chain (e.g., `Foo.Bar.Baz` without defining `Foo` or `Foo.Bar` first).

As we will see in later chapters, aliases also play a crucial role in macros, to guarantee they are hygienic.

With this we are almost finishing our tour about Elixir modules. The last topic to cover is module attributes.

14 Module attributes

{% include toc.html %}

Module attributes in Elixir serve three purposes:

1. They serve to annotate the module, often with information to be used by the user or the VM.
2. They work as constants.
3. They work as a temporary module storage to be used during compilation.

Let's check each case, one by one.

14.1 As annotations

Elixir brings the concept of module attributes from Erlang. For example:

```
defmodule MyServer do
  @vsn 2
end
```

In the example above, we are explicitly setting the version attribute for that module. `@vsn` is used by the code reloading mechanism in the Erlang VM to check if a module has been updated or not. If no version is specified, the version is set to the MD5 checksum of the module functions.

Elixir has a handful of reserved attributes. Here are just a few of them, the most commonly used ones:

- `@moduledoc` - provides documentation for the current module.
- `@doc` - provides documentation for the function or macro that follows the attribute.
- `@behaviour` - (notice the British spelling) used for specifying an OTP or user-defined behaviour.
- `@before_compile` - provides a hook that will be invoked before the module is compiled. This makes it possible to inject functions inside the module exactly before compilation.

`@moduledoc` and `@doc` are by far the most used attributes, and we expect you to use them a lot. Elixir treats documentation as first-class and provides many functions to access documentation.

Let's go back to the `Math` module defined in the previous chapters, add some documentation and save it to the `math.ex` file:

```
defmodule Math do
  @moduledoc """
  Provides math-related functions.

  ## Examples

      iex> Math.sum(1, 2)
      3

  """

  @doc """
  Calculates the sum of two numbers.
  """
  def sum(a, b), do: a + b
end
```

Elixir promotes the use of markdown with heredocs to write readable documentation. Heredocs are multiline strings, they start and end with triple quotes, keeping the formatting of the inner text. We can access the documentation of any compiled module directly from IEx:

```
$ elixirc math.ex
$ iex
```

```
iex> h Math # Access the docs for the module Math
...
iex> h Math.sum # Access the docs for the sum function
...
```

We also provide a tool called ExDoc which is used to generate HTML pages from the documentation.

You can take a look at the docs for Module for a complete list of supported attributes. Elixir also uses attributes to define typespecs, via:

- `@spec` - provides a specification for a function.
- `@callback` - provides a specification for the behaviour callback.
- `@type` - defines a type to be used in `@spec`.
- `@typep` - defines a private type to be used in `@spec`.
- `@opaque` - defines an opaque type to be used in `@spec`.

This section covers built-in attributes. However, attributes can also be used by developers or extended by libraries to support custom behaviour.

14.2 As constants

Elixir developers will often use module attributes to be used as constants:

```
defmodule MyServer do
  @initial_state %{host: "147.0.0.1", port: 3456}
  IO.inspect @initial_state
end

Note: Unlike Erlang, user defined attributes are not stored in the
module by default. The value exists only during compilation time. A
developer can configure an attribute to behave closer to Erlang by
calling
```Module.register_attribute/3`` </docs/stable/elixir/Module.html#register_attribute/
↪3>`__.
```

Trying to access an attribute that was not defined will print a warning:

```
defmodule MyServer do
  @unknown
end
warning: undefined module attribute @unknown, please remove access to @unknown or␣
↪explicitly set it to nil before access
```

Finally, attributes can also be read inside functions:

```
defmodule MyServer do
  @my_data 14
  def first_data, do: @my_data
  @my_data 13
  def second_data, do: @my_data
end

MyServer.first_data #=> 14
MyServer.second_data #=> 13
```
```

Notice that reading an attribute inside a function takes a snapshot of its current value. In other words, the value is read at compilation time and not at runtime. As we are going to see, this makes attributes useful to be used as storage during module compilation.

14.3 As temporary storage

One of the projects in the Elixir organization is *the ‘‘Plug‘* project <https://github.com/elixir-lang/plug>‘__, which is meant to be a common foundation for building web libraries and frameworks in Elixir.

The Plug library also allows developers to define their own plugs which can be run in a web server:

```
defmodule MyPlug do
  use Plug.Builder

  plug :set_header
  plug :send_ok

  def set_header(conn, _opts) do
    put_resp_header(conn, "x-header", "set")
  end

  def send_ok(conn, _opts) do
    send(conn, 200, "ok")
  end
end

IO.puts "Running MyPlug with Cowboy on http://localhost:4000"
Plug.Adapters.Cowboy.http MyPlug, []
```

In the example above, we have used the `plug/1` macro to connect functions that will be invoked when there is a web request. Internally, every time you call `plug/1`, the Plug library stores the given argument in a `@plugs` attribute. Just before the module is compiled, Plug runs a callback that defines a method (`call/2`) which handles http requests. This method will run all plugs inside `@plugs` in order.

In order to understand the underlying code, we'd need macros, so we will revisit this pattern in the meta-programming guide. However the focus here is exactly on how using module attributes as storage allow developers to create DSLs.

Another example comes from the ExUnit framework which uses module attributes as annotation and storage:

```
defmodule MyTest do
  use ExUnit.Case

  @tag :external
  test "contacts external service" do
    # ...
  end
end
```

Tags in ExUnit are used to annotate tests. Tags can be later used to filter tests. For example, you can avoid running external tests on your machine because they are slow and dependent on other services, while they can still be enabled in your build system.

We hope this section shines some light on how Elixir supports meta-programming and how module attributes play an important role when doing so.

In the next chapters we'll explore structs and protocols before moving to exception handling and other constructs like sigils and comprehensions.

15 Structs

{% include toc.html %}

In chapter 7 we learned about maps:

```
iex> map = %{a: 1, b: 2}
%{a: 1, b: 2}
iex> map[:a]
1
iex> %{map | a: 3}
%{a: 3, b: 2}
```

Structs are extensions built on top of maps that provide compile-time checks and default values.

15.1 Defining structs

To define a struct, the `defstruct` construct is used:

```
iex> defmodule User do
...>   defstruct name: "John", age: 27
...> end
```

The keyword list used with `defstruct` defines what fields the struct will have along with their default values.

Structs take the name of the module they're defined in. In the example above, we defined a struct named `User`.

We can now create `User` structs by using a syntax similar to the one used to create maps:

```
iex> %User{}
%User{age: 27, name: "John"}
iex> %User{name: "Meg"}
%User{age: 27, name: "Meg"}
```

Structs provide *compile-time* guarantees that only the fields (and *all* of them) defined through `defstruct` will be allowed to exist in a struct:

```
iex> %User{oops: :field}
** (CompileError) iex:3: unknown key :oops for struct User
```

15.2 Accessing and updating structs

When we discussed maps, we showed how we can access and update the fields of a map. The same techniques (and the same syntax) apply to structs as well:

```
iex> john = %User{}
%User{age: 27, name: "John"}
iex> john.name
"John"
iex> meg = %{john | name: "Meg"}
%User{age: 27, name: "meg"}
iex> %{meg | oops: :field}
** (ArgumentError) argument error
```

When using the update syntax (`|`), the VM is aware that no new keys will be added to the struct, allowing the maps underneath to share their structure in memory. In the example above, both `john` and `meg` share the same key structure in memory.

Structs can also be used in pattern matching, both for matching on the value of specific keys as well as for ensuring that the matching value is a struct of the same type as the matched value.

```
iex> %User{name: name} = john
%User{age: 27, name: "John"}
iex> name
"John"
iex> %User{} = %{}
** (MatchError) no match of right hand side value: %{}
```

15.3 Structs are just bare maps underneath

In the example above, pattern matching works because underneath structs are just bare maps with a fixed set of fields. As maps, structs store a "special" field named `__struct__` that holds the name of the struct:

```
iex> is_map(john)
true
iex> john.__struct__
User
```

Notice that we referred to structs as **bare** maps because none of the protocols implemented for maps are available for structs. For example, you can't enumerate nor access a struct:

```
iex> john = %User{}
%User{age: 27, name: "John"}
iex> john[:name]
** (Protocol.UndefinedError) protocol Access not implemented for %User{age: 27, name:
↪"John"}
iex> Enum.each john, fn({field, value}) -> IO.puts(value) end
** (Protocol.UndefinedError) protocol Enumerable not implemented for %User{age: 27,␣
↪name: "John"}
```

A struct also is not a dictionary and therefore can't be used with the functions from the `Dict` module:

```
iex> Dict.get(%User{}, :name)
** (UndefinedFunctionError) undefined function: User.fetch/2
```

However, since structs are just maps, they work with the functions from the `Map` module:

```
iex> kurt = Map.put(%User{}, :name, "Kurt")
%User{age: 27, name: "Kurt"}
iex> Map.merge(kurt, %User{name: "Takashi"})
%User{age: 27, name: "Takashi"}
iex> Map.keys(john)
[:__struct__, :age, :name]
```

We will cover how structs interact with protocols in the next chapter.

16 Protocols

{% include toc.html %}

Protocols are a mechanism to achieve polymorphism in Elixir. Dispatching on a protocol is available to any data type as long as it implements the protocol. Let's see an example.

In Elixir, only `false` and `nil` are treated as false. Everything else evaluates to true. Depending on the application, it may be important to specify a `blank?` protocol that returns a boolean for other data types that should be considered blank. For instance, an empty list or an empty binary could be considered blanks.

We could define this protocol as follows:

```
defprotocol Blank do
  @doc "Returns true if data is considered blank/empty"
  def blank?(data)
end
```

The protocol expects a function called `blank?` that receives one argument to be implemented. We can implement this protocol for different Elixir data types as follows:

```
# Integers are never blank
defimpl Blank, for: Integer do
  def blank?(_), do: false
end

# Just empty list is blank
defimpl Blank, for: List do
  def blank?([]), do: true
  def blank?(_),  do: false
end

# Just empty map is blank
defimpl Blank, for: Map do
  # Keep in mind we could not pattern match on %{} because
  # it matches on all maps. We can however check if the size
  # is zero (and size is a fast operation).
  def blank?(map), do: map_size(map) == 0
end

# Just the atoms false and nil are blank
defimpl Blank, for: Atom do
  def blank?(false), do: true
  def blank?(nil),   do: true
  def blank?(_),     do: false
end
```

And we would do so for all native data types. The types available are:

- `Atom`
- `BitString`
- `Float`
- `Function`
- `Integer`
- `List`
- `Map`
- `PID`
- `Port`
- `Reference`
- `Tuple`

Now with the protocol defined and implementations in hand, we can invoke it:

```
iex> Blank.blank?(0)
false
iex> Blank.blank?([])
true
iex> Blank.blank?([1, 2, 3])
false
```

Passing a data type that does not implement the protocol raises an error:

```
iex> Blank.blank?("hello")
** (Protocol.UndefinedError) protocol Blank not implemented for "hello"
```

16.1 Protocols and structs

The power of Elixir's extensibility comes when protocols and structs are used together.

In the previous chapter, we have learned that although structs are maps, they do not share protocol implementations with maps. Let's define a `User` struct as in the previous chapter:

```
iex> defmodule User do
...>   defstruct name: "john", age: 27
...> end
{:module, User,
 <<70, 79, 82, ...>>, {:__struct__, 0}}
```

And then check:

```
iex> Blank.blank?(%{})
true
iex> Blank.blank?(%User{})
** (Protocol.UndefinedError) protocol Blank not implemented for %User{age: 27, name:
↪"john"}
```

Instead of sharing protocol implementation with maps, structs require their own protocol implementation:

```
defimpl Blank, for: User do
  def blank?(_), do: false
end
```

If desired, you could come up with your own semantics for a user being blank. Not only that, you could use structs to build more robust data types, like queues, and implement all relevant protocols, such as `Enumerable` and possibly `Blank`, for this data type.

In many cases though, developers may want to provide a default implementation for structs, as explicitly implementing the protocol for all structs can be tedious. That's when falling back to Any comes in handy.

16.2 Falling back to Any

It may be convenient to provide a default implementation for all types. This can be achieved by setting `@fallback_to_any` to `true` in the protocol definition:

```
defprotocol Blank do
  @fallback_to_any true
```

```
  def blank?(data)
end
```

Which can now be implemented as:

```
defimpl Blank, for: Any do
  def blank?(_), do: false
end
```

Now all data types (including structs) that we have not implemented the `Blank` protocol for will be considered non-blank.

16.3 Built-in protocols

Elixir ships with some built-in protocols. In previous chapters, we have discussed the `Enum` module which provides many functions that work with any data structure that implements the `Enumerable` protocol:

```
iex> Enum.map [1, 2, 3], fn(x) -> x * 2 end
[2,4,6]
iex> Enum.reduce 1..3, 0, fn(x, acc) -> x + acc end
6
```

Another useful example is the `String.Chars` protocol, which specifies how to convert a data structure with characters to a string. It's exposed via the `to_string` function:

```
iex> to_string :hello
"hello"
```

Notice that string interpolation in Elixir calls the `to_string` function:

```
iex> "age: #{25}"
"age: 25"
```

The snippet above only works because numbers implement the `String.Chars` protocol. Passing a tuple, for example, will lead to an error:

```
iex> tuple = {1, 2, 3}
{1, 2, 3}
iex> "tuple: #{tuple}"
** (Protocol.UndefinedError) protocol String.Chars not implemented for {1, 2, 3}
```

When there is a need to "print" a more complex data structure, one can simply use the `inspect` function, based on the `Inspect` protocol:

```
iex> "tuple: #{inspect tuple}"
"tuple: {1, 2, 3}"
```

The `Inspect` protocol is the protocol used to transform any data structure into a readable textual representation. This is what tools like IEx use to print results:

```
iex> {1, 2, 3}
{1,2,3}
iex> %User{}
%User{name: "john", age: 27}
```

Keep in mind that, by convention, whenever the inspected value starts with #, it is representing a data structure in non-valid Elixir syntax. This means the inspect protocol is not reversible as information may be lost along the way:

```
iex> inspect &(&1+2)
"#Function<6.71889879/1 in :erl_eval.expr/5>"
```

There are other protocols in Elixir but this covers the most common ones.

17 Comprehensions

{% include toc.html %}

In Elixir, it is common to loop over an Enumerable, often filtering out some results and mapping values into another list. Comprehensions are syntactic sugar for such constructs: they group those common tasks into the `for` special form.

For example, we can map a list of integers into their squared values:

```
iex> for n <- [1, 2, 3, 4], do: n * n
[1, 4, 9, 16]
```

A comprehension is made of three parts: generators, filters and collectables.

17.1 Generators and filters

In the expression above, `n <- [1, 2, 3, 4]` is the **generator**. It is literally generating values to be used in the comprehension. Any enumerable can be passed in the right-hand side of the generator expression:

```
iex> for n <- 1..4, do: n * n
[1, 4, 9, 16]
```

Generator expressions also support pattern matching on their left-hand side; all non-matching patterns are *ignored*. Imagine that, instead of a range, we have a keyword list where the key is the atom `:good` or `:bad` and we only want to compute the square of the `:good` values:

```
iex> values = [good: 1, good: 2, bad: 3, good: 4]
iex> for {:good, n} <- values, do: n * n
[1, 4, 16]
```

Alternatively to pattern matching, filters can be used to filter some particular elements out. For example, we can get filter out all the multiples of 3 and get the square of the remaining values only:

```
iex> multiple_of_3? = fn(n) -> rem(n, 3) == 0 end
iex> for n <- 0..5, multiple_of_3?.(n), do: n * n
[0, 9]
```

Comprehensions filter out all elements for which the filter expression returns `false` or `nil`; all other values are kept.

Comprehensions generally provide a much more concise representation than using the equivalent functions from the `Enum` and `Stream` modules. Furthermore, comprehensions also allow multiple generators and filters to be given. Here is an example that receives a list of directories and deletes all files in those directories:

```
for dir  <- dirs,
    file <- File.ls!(dir),
    path = Path.join(dir, file),
    File.regular?(path) do
```

```
  File.rm!(path)
end
```

Keep in mind that variable assignments inside the comprehension, be it in generators, filters or inside the block, are not reflected outside of the comprehension.

17.2 Bitstring generators

Bitstring generators are also supported and are very useful when you need to comprehend over bitstring streams. The example below receives a list of pixels from a binary with their respective red, green and blue values and converts them into tuples of three elements each:

```
iex> pixels = <<213, 45, 132, 64, 76, 32, 76, 0, 0, 234, 32, 15>>
iex> for <<r::8, g::8, b::8 <- pixels>>, do: {r, g, b}
[{213, 45, 132}, {64, 76, 32}, {76, 0, 0}, {234, 32, 15}]
```

A bitstring generator can be mixed with the "regular" enumerable generators and provides filters as well.

17.3 Results other than lists

In the examples above, all the comprehensions returned lists as their result. However, the result of a comprehension can be inserted into different data structures by passing the `:into` option to the comprehension.

For example, a bitstring generator can be used with the `:into` option in order to easily remove all spaces in a string:

```
iex> for <<c <- " hello world ">>, c != ?\s, into: "", do: <<c>>
"helloworld"
```

Sets, maps and other dictionaries can also be given to the `:into` option. In general, `:into` accepts any structure that implements the `Collectable` protocol.

A common use case of `:into` can be transforming values in a map, without touching the keys:

```
iex> for {key, val} <- %{"a" => 1, "b" => 2}, into: %{}, do: {key, val * val}
%{"a" => 1, "b" => 4}
```

Let's make another example using streams. Since the `IO` module provides streams (that are both `Enumerables` and `Collectables`), an echo terminal that echoes back the upcased version of whatever is typed can be implemented using comprehensions:

```
iex> stream = IO.stream(:stdio, :line)
iex> for line <- stream, into: stream do
...>   String.upcase(line) <> "\n"
...> end
```

Now type any string into the terminal and you will see that the same value will be printed in upper-case. Unfortunately, this example also got your IEx shell stuck in the comprehension, so you will need to hit `Ctrl+C` twice to get out of it. :)

18 Sigils

{% include toc.html %}

We have already learned that Elixir provides double-quoted strings and single-quoted char lists. However, this only covers the surface of structures that have textual representation in the language. Atoms are, for example, are mostly created via the `:atom` representation.

One of Elixir's goals is extensibility: developers should be able to extend the language to fit any particular domain. Computer science has become such a wide field that it is impossible for a language to tackle many fields as part of its core. Our best bet is to rather make the language extensible, so developers, companies and communities can extend the language to their relevant domains.

In this chapter, we are going to explore sigils, which are one of the mechanisms provided by the language for working with textual representations. Sigils start with the tilde (~) character which is followed by a letter (which identifies the sigil) and then a delimiter; optionally, modifiers can be added after the final delimiter.

18.1 Regular expressions

The most common sigil in Elixir is `~r`, which is used to create regular expressions:

```
# A regular expression that matches strings which contain "foo" or "bar":
iex> regex = ~r/foo|bar/
~r/foo|bar/
iex> "foo" =~ regex
true
iex> "bat" =~ regex
false
```

Elixir provides Perl-compatible regular expressions (regexes), as implemented by the PCRE library. Regexes also support modifiers. For example, the `i` modifier makes a regular expression case insensitive:

```
iex> "HELLO" =~ ~r/hello/
false
iex> "HELLO" =~ ~r/hello/i
true
```

Check out the `Regex module </docs/stable/elixir/Regex.html>‘__ for more information on other modifiers and the supported operations with regular expressions.

So far, all examples have used `/` to delimit a regular expression. However sigils support 8 different delimiters:

```
~r/hello/
~r|hello|
~r"hello"
~r'hello'
~r(hello)
~r[hello]
~r{hello}
~r<hello>
```

The reason behind supporting different delimiters is that different delimiters can be more suited for different sigils. For example, using parentheses for regular expressions may be a confusing choice as they can get mixed with the parentheses inside the regex. However, parentheses can be handy for other sigils, as we will see in the next section.

18.2 Strings, char lists and words sigils

Besides regular expressions, Elixir ships with three other sigils.

18.2.1 Strings

The `~s` sigil is used to generate strings, like double quotes are. The `~s` sigil is useful, for example, when a string contains both double and single quotes:

```
iex> ~s(this is a string with "double" quotes, not 'single' ones)
"this is a string with \"double\" quotes, not 'single' ones"
```

18.2.2 Char lists

The `~c` sigil is used to generate char lists:

```
iex> ~c(this is a char list containing 'single quotes')
'this is a char list containing \'single quotes\''
```

18.2.3 Word lists

The `~w` sigil is used to generate lists of words (*words* are just regular strings). Inside the `~w` sigil, words are separated by whitespace.

```
iex> ~w(foo bar bat)
["foo", "bar", "bat"]
```

The `~w` sigil also accepts the `c`, `s` and `a` modifiers (for char lists, strings and atoms, respectively) which specify the data type of the elements of the resulting list:

```
iex> ~w(foo bar bat)a
[:foo, :bar, :bat]
```

18.3 Interpolation and escaping in sigils

Besides lowercase sigils, Elixir supports uppercase sigils to deal with escaping characters and interpolation. While both `~s` and `~S` will return strings, the former allows escape codes and interpolation while the latter does not:

```
iex> ~s(String with escape codes \x26 #{"inter" <> "polation"})
"String with escape codes & interpolation"
iex> ~S(String without escape codes and without #{interpolation})
"String without escape codes and without \#{interpolation}"
```

The following escape codes can be used in strings and char lists:

- `\"` – double quote
- `\'` – single quote
- `\\` – single backslash
- `\a` – bell/alert
- `\b` – backspace
- `\d` - delete
- `\e` - escape

- \f - form feed
- \n – newline
- \r – carriage return
- \s – space
- \t – tab
- \v – vertical tab
- \0 - null byte
- \xDD - character with hexadecimal representation DD (e.g., \x13)
- \x{D...} - character with hexadecimal representation with one or more hexadecimal digits (e.g., \x{abc13})

Sigils also support heredocs, that is, triple double- or single-quotes as separators:

```
iex> ~s"""
...> this is
...> a heredoc string
...> """
```

The most common use case for heredoc sigils is when writing documentation. For example, writing escape characters in documentation would soon become error prone because of the need to double-escape some characters:

```
@doc """
Converts double-quotes to single-quotes.

## Examples

    iex> convert("\\\"foo\\\"")
    "'foo'"

"""
def convert(...)
```

By using using ~S, this problem can be avoided altogether:

```
@doc ~S"""
Converts double-quotes to single-quotes.

## Examples

    iex> convert("\"foo\"")
    "'foo'"

"""
def convert(...)
```

18.4 Custom sigils

As hinted at the beginning of this chapter, sigils in Elixir are extensible. In fact, using the sigil ~r/foo/i is equivalent to calling the sigil_r function with a binary and a char list as argument:

```
iex> sigil_r(<<"foo">>, 'i')
~r"foo"i
```

We can access the documentation for the `~r` sigil via the `sigil_r` function:

```
iex> h sigil_r
...
```

We can also provide our own sigils by simply implementing functions that follow the `sigil_{identifier}` pattern. For example, let's implement the `~i` sigil that returns an integer (with the optional `n` modifier to make it negative):

```
iex> defmodule MySigils do
...>   def sigil_i(string, []), do: String.to_integer(string)
...>   def sigil_i(string, [?n]), do: -String.to_integer(string)
...> end
iex> import MySigils
iex> ~i(13)
13
iex> ~i(42)n
-42
```

Sigils can also be used to do compile-time work with the help of macros. For example, regular expressions in Elixir are compiled into an efficient representation during compilation of the source code, therefore skipping this step at runtime. If you're interested in the subject, we recommend you to learn more about macros and check out how sigils are implemented in the `Kernel` module (where the `sigil_*` functions are defined).

19 try, catch and rescue

{% include toc.html %}

Elixir has three error mechanisms: errors, throws and exits. In this chapter we will explore each of them and include remarks about when each should be used.

19.1 Errors

Errors (or *exceptions*) are used when exceptional things happen in the code. A sample error can be retrieved by trying to add a number into an atom:

```
iex> :foo + 1
** (ArithmeticError) bad argument in arithmetic expression
     :erlang.+(:foo, 1)
```

A runtime error can be raised any time by using `raise/1`:

```
iex> raise "oops"
** (RuntimeError) oops
```

Other errors can be raised with `raise/2` passing the error name and a list of keyword arguments:

```
iex> raise ArgumentError, message: "invalid argument foo"
** (ArgumentError) invalid argument foo
```

You can also define your own errors by creating a module and using the `defexception` construct inside it; this way, you'll create an error with the same name as the module it's defined in. The most common case is to define a custom exception with a message field:

```
iex> defmodule MyError do
iex>   defexception message: "default message"
iex> end
iex> raise MyError
** (MyError) default message
iex> raise MyError, message: "custom message"
** (MyError) custom message
```

Errors can be **rescued** using the `try/rescue` construct:

```
iex> try do
...>   raise "oops"
...> rescue
...>   e in RuntimeError -> e
...> end
%RuntimeError{message: "oops"}
```

The example above rescues the runtime error and returns the error itself which is then printed in the `iex` session. In practice, however, Elixir developers rarely use the `try/rescue` construct. For example, many languages would force you to rescue an error when a file cannot be opened successfully. Elixir instead provides a `File.read/1` function which returns a tuple containing informations about whether the file was successfully opened:

```
iex> File.read "hello"
{:error, :enoent}
iex> File.write "hello", "world"
:ok
iex> File.read "hello"
{:ok, "world"}
```

There is no `try/rescue` here. In case you want to handle multiple outcomes of opening a file, you can simply use pattern matching with the `case` construct:

```
iex> case File.read "hello" do
...>   {:ok, body}      -> IO.puts "Success: #{body}"
...>   {:error, reason} -> IO.puts "Error: #{reason}"
...> end
```

At the end of the day, it's up to your application to decide if an error while opening a file is exceptional or not. That's why Elixir doesn't impose exceptions on `File.read/1` and many other functions. Instead, it leaves it up to the developer to choose the best way to proceed.

For the cases where you do expect a file to exist (and the lack of that file is truly an *error*) you can simply use `File.read!/1`:

```
iex> File.read! "unknown"
** (File.Error) could not read file unknown: no such file or directory
    (elixir) lib/file.ex:305: File.read!/1
```

Many functions in the standard library follow the pattern of having a counterpart that raises an exception instead of returning tuples to match against. The convention is to create a function (`foo`) which returns `{:ok, result}` or `{:error, reason}` tuples and another function (`foo!`, same name but with a trailing `!`) that takes the same arguments as `foo` but which raises an exception if there's an error. `foo!` should return the result (not wrapped in a tuple) if everything goes fine. The `` `File `` module </docs/stable/elixir/File.html>`__ is a good example of this convention.

In Elixir, we avoid using `try/rescue` because **we don't use errors for control flow**. We take errors literally: they are reserved to unexpected and/or exceptional situations. In case you actually need flow control constructs, *throws* should be used. That's what we are going to see next.

19.2 Throws

In Elixir, a value can be thrown and later be caught. `throw` and `catch` are reserved for situations where it is not possible to retrieve a value unless by using `throw` and `catch`.

Those situations are quite uncommon in practice except when interfacing with libraries that does not provide a proper API. For example, let's imagine the `Enum` module did not provide any API for finding a value and that we needed to find the first multiple of 13 in a list of numbers:

```
iex> try do
...>   Enum.each -50..50, fn(x) ->
...>     if rem(x, 13) == 0, do: throw(x)
...>   end
...>   "Got nothing"
...> catch
...>   x -> "Got #{x}"
...> end
"Got -39"
```

Since `Enum` *does* provide a proper API, in practice `Enum.find/2` is the way to go:

```
iex> Enum.find -50..50, &(rem(&1, 13) == 0)
-39
```

19.3 Exits

All Elixir code runs inside processes that communicate with each other. When a process dies of "natural causes" (e.g., unhandled exceptions), it sends an `exit` signal. A process can also die by explicitly sending an exit signal:

```
iex> spawn_link fn -> exit(1) end
#PID<0.56.0>
** (EXIT from #PID<0.56.0>) 1
```

In the example above, the linked process died by sending an `exit` signal with value of 1. The Elixir shell automatically handles those messages and prints them to the terminal.

`exit` can also be "caught" using `try/catch`:

```
iex> try do
...>   exit "I am exiting"
...> catch
...>   :exit, _ -> "not really"
...> end
"not really"
```

Using `try/catch` is already uncommon and using it to catch exits is even more rare.

`exit` signals are an important part of the fault tolerant system provided by the Erlang VM. Processes usually run under supervision trees which are themselves processes that just wait for `exit` signals from the supervised processes. Once an exit signal is received, the supervision strategy kicks in and the supervised process is restarted.

It is exactly this supervision system that makes constructs like `try/catch` and `try/rescue` so uncommon in Elixir. Instead of rescuing an error, we'd rather "fail fast" since the supervision tree will guarantee our application will go back to a known initial state after the error.

19.4 After

Sometimes it's necessary to ensure that a resource is cleaned up after some action that could potentially raise an error. The `try/after` construct allows you to do that. For example, we can open a file and guarantee it will be closed (even if something goes wrong) with a `try/after` block:

```
iex> {:ok, file} = File.open "sample", [:utf8, :write]
iex> try do
...>   IO.write file, "olá"
...>   raise "oops, something went wrong"
...> after
...>   File.close(file)
...> end
** (RuntimeError) oops, something went wrong
```

19.5 Variables scope

It is important to bear in mind that variables defined inside `try/catch/rescue/after` blocks do not leak to the outer context. This is because the `try` block may fail and as such the variables may never be bound in the first place. In other words, this code is invalid:

```
iex> try do
...>   from_try = true
...> after
...>   from_after = true
...> end
iex> from_try
** (RuntimeError) undefined function: from_try/0
iex> from_after
** (RuntimeError) undefined function: from_after/0
```

This finishes our introduction to `try`, `catch` and `rescue`. You will find they are used less frequently in Elixir than in other languages although they may be handy in some situations where a library or some particular code is not playing "by the rules".

20 Typespecs and behaviours

{% include toc.html %}

20.1 Types and specs

Elixir is a dynamically typed language, so all types in Elixir are inferred by the runtime. Nonetheless, Elixir comes with **typespecs**, which are a notation used for:

1. declaring custom data types;
2. declaring typed function signatures (specifications).

20.1.1 Function specifications

By default, Elixir provides some basic types, such as `integer` or `pid`, as well as more complex types: for example, the `round/1` function, which rounds a float to its nearest integer, takes a `number` as an argument (an `integer` or a `float`) and returns an `integer`. As you can see in its documentation, `round/1`'s typed signature is written as:

```
round(number) :: integer
```

`::` means that the function on the left side *returns* a value whose type is what's on the right side. Function specs are written with the `@spec` directive, placed right before the function definition. The `round/1` function could be written as:

```
@spec round(number) :: integer
def round(number), do: # implementation...
```

Elixir supports compound types as well. For example, a list of integers has type `[integer]`. You can see all the types provided by Elixir in the typespecs docs.

20.1.2 Defining custom types

While Elixir provides a lot of useful built-in types, it's convenient to define custom types when appropriate. This can be done when defining modules modules through the `@type` directive.

Say we have a `LousyCalculator` module, which performs the usual arithmetic operations (sum, product and so on) but, instead of returning numbers, it returns tuples with the result of an operation as the first element and a random offense as the second element.

```
defmodule LousyCalculator do
  @spec add(number, number) :: {number, String.t}
  def add(x, y), do: {x + y, "You need a calculator to do that?!"}

  @spec multiply(number, number) :: {number, String.t}
  def multiply(x, y), do: {x * y, "Jeez, come on!"}
end
```

As you can see in the example, tuples are a compound type and each tuple is identified by the types inside it. To understand why `String.t` is not written as `string`, have another look at the typespecs docs.

Defining function specs this way works, but it quickly becomes annoying since we're repeating the type `{number, String.t}` over and over. We can use the `@type` directory in order to declare our own custom type.

```
defmodule LousyCalculator do
  @typedoc """
  Just a number followed by a string.
  """
  @type number_with_offense :: {number, String.t}

  @spec add(number, number) :: number_with_offense
  def add(x, y), do: {x + y, "You need a calculator to do that?!"}

  @spec multiply(number, number) :: number_with_offense
  def multiply(x, y), do: {x * y, "Jeez, come on!"}
end
```

The `@typedoc` directive, similarly to the `@doc` and `@moduledoc` directives, is used to document custom types.

Custom types defined through `@type` are exported and available outside the module they're defined in:

```
defmodule PoliteCalculator do
  @spec add(number, number) :: number
  def add(x, y), do: make_polite(LousyCalculator.add(x, y))

  @spec make_polite(LousyCalculator.number_with_offense) :: number
```

```
  defp make_polite({num, _offense}), do: num
end
```

If you want to keep a custom type private, you can use the `@typep` directive instead of `@type`.

20.1.3 Static code analysis

Typespecs are not only useful to developers and as additional documentation. The Erlang tool Dialyzer, for example, uses typespecs in order to perform static analysis of code. That's why, in the `PoliteCalculator` example, we wrote a spec for the `make_polite/1` function even if it was defined as a private function.

20.2 Behaviours

Many modules share the same public API. Take a look at Plug, which, as it description states, is a **specification** for composable modules in web applications. Each *plug* is a module which **has to** implement at least two public functions: `init/1` and `call/2`.

Behaviors provide a way to:

- define a set of functions that have to be implemented by a module;
- ensure that a module implements all the functions in that set.

If you have to, you can think of behaviours like interfaces in object oriented languages like Java: a set of function signatures that a module has to implement.

20.2.1 Defining behaviours

Say we have want to implement a bunch of parsers, each parsing structured data: for example, a JSON parser and a YAML parser. Each of these two parsers will *behave* the same way: both will provide a `parse/1` function and a `extensions/0` function. The `parse/1` function will return an Elixir representation of the structured data, while the `extensions/0` function will return a list of file extensions that can be used for each type of data (e.g., `.json` for JSON files).

We can create a `Parser` behaviour:

```
defmodule Parser do
  use Behaviour

  defcallback parse(String.t) :: any
  defcallback extensions() :: [String.t]
end
```

Modules adopting the `Parser` behaviour will have to implement all the functions defined with `defcallback`. As you can see, `defcallback` expects a function name but also a function specification like the ones used with the `@spec` directive we saw above.

20.2.2 Adopting behaviours

Adopting a behaviour is straightforward:

```
defmodule JSONParser do
  @behaviour Parser

  def parse(str), do: # ... parse JSON
  def extensions, do: ["json"]
end
```

```
defmodule YAMLParser do
  @behaviour Parser

  def parse(str), do: # ... parse YAML
  def extensions, do: ["yml"]
end
```

If a module adopting a given behaviour doesn't implement one of the callbacks required by that behaviour, a compile-time warning will be generated.

Mix - a build tool for Elixir

1 Introduction to Mix

{% include toc.html %}

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 v0.15.0 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!

1.1 Our first project

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

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

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

Mix will create a directory named `kv` with 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.

1.2 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: "0.0.1",
     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:

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

Notice the `lib/kv.ex` file was compiled and `kv.app` file was generated. All this took place in a directory structure of its own, inside the `_build` folder. This `.app` file is generated with the information from the `application/0` function in the `mix.exs` file. We will further explore `mix.exs` configuration features in future chapters.

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

```
$ iex -S mix
```

1.3 Running tests

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

```
defmodule KVTest do
  use ExUnit.Case

  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 </docs/stable/ex_unit/ExUnit.Case.html>`__ `` 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 tests, 0 failures

Randomized with seed 540224
```

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

Furthermore, you can see that ExUnit prints a dot for each successful test and automatically 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:4
   Assertion with == failed
   code: 1 + 1 == 3
   lhs:  2
   rhs:  3
   stacktrace:
     test/kv_test.exs:5

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

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

In the second line of the failure, right below the test name, there is the location 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:4
```

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

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

1.4 Environments

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

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

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

By default, these environments behave the same and all the configurations we have seen so far will affect all three environments. Customization per environment can be done by accessing *the ''Mix.env`* function </docs/stable/mix/Mix.html#env/1>`__ in your `mix.exs` file, which returns the current environment as an atom:

```
def project do
  [deps_path: deps_path(Mix.env)]
end

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

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

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

1.5 Exploring

There is much more to Mix, and we will continue to explore it as we build our project. 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!

2 Agent

{% include toc.html %}

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

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

2.1 The trouble with state

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

- Processes
- ETS (Erlang Term Storage)

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

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

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

2.2 Agents

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

```
$ iex -S mix
```

And play a bit with agents:

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

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

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

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

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

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

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

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

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

Note that we are using a HashDict to store our state instead of a `Map`, because in the current version of Elixir maps are less efficient when holding a large number of keys.

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

2.3 ExUnit callbacks

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

Let's rewrite the test case to use callbacks:

```
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 </docs/stable/ex_unit/ExUnit.Case.html>‘__ and more about callbacks in `ExUnit.Callbacks` docs </docs/stable/ex_unit/ExUnit.Callbacks.html>‘__.

2.4 Other Agent actions

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

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

Returns the current value of `key`, if `key` exists.
"""
def delete(bucket, key) do
  Agent.get_and_update(bucket, &HashDict.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 Agents to learn more about them.

2.5 Client/Server in Agents

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

```
def delete(bucket, key) do
  Agent.get_and_update(bucket, fn dict->
    HashDict.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
  :timer.sleep(1000) # puts client to sleep
  Agent.get_and_update(bucket, fn dict ->
    :timer.sleep(1000) # puts server to sleep
    HashDict.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.

3 GenServer

{% include toc.html %}

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

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

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

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

3.1 Our first GenServer

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

```
defmodule KV.Registry do
  use GenServer

  ## Client API

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

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

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

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

  ## Server Callbacks

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2 Testing a 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!

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

```
## Client API

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

## Server Callbacks

def handle_call(:stop, _from, state) do
  {:stop, :normal, :ok, state}
end
```

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

3.3 The need for monitoring

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

```
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()` 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 -> pid` and another that holds `ref -> name`. Then we need to monitor the buckets on `handle_cast/2` as well as implement a `handle_info/2` callback to handle the monitoring messages. The full server callbacks implementation is shown below:

```
## Server callbacks

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

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

def handle_call(:stop, _from, state) do
  {:stop, :normal, :ok, state}
end

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

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

3.4 call, cast or info?

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

1. `handle_call/3` must be used for synchronous requests. This should be the default choice as waiting for the server reply is a useful backpressure mechanism.
2. `handle_cast/2` must be used for asynchronous requests, when you don't care about a reply. A cast does not even guarantee the server has received the message and, for this reason, must be used sparingly. For example, the `create/2` function we have defined in this chapter should have used `call/2`. We have used `cast/2` for 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 supervisor to crash, because no clause would match.

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

3.5 Monitors or links?

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

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

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

```
{: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 will explore solutions to this problem when we talk about supervisors. In a nutshell, we typically avoid creating new processes directly. Instead, we delegate this responsibility to supervisors. As we'll see, supervisors work with links, and that explains why link-based APIs (`spawn_link`, `start_link`, etc) are so prevalent in Elixir and OTP.

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

4 GenEvent

{% include toc.html %}

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

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

4.1 Event managers

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

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

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

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

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

```
iex> defmodule Forwarder do
...>   use GenEvent
...>   def handle_event(event, parent) do
...>     send parent, event
...>     {:ok, parent}
...>   end
...> end
iex> GenEvent.add_handler(manager, Forwarder, self())
:ok
iex> GenEvent.sync_notify(manager, {:hello, :world})
:ok
iex> flush
{:hello, :world}
:ok
```

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

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

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

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

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

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

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

4.2 Registry events

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

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

```
defmodule Forwarder do
  use GenEvent

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

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

  GenEvent.add_mon_handler(manager, Forwarder, self())
  {:ok, registry: registry}
end

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

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

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

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

Finally, notice we called `GenEvent.add_mon_handler/3` instead of `GenEvent.add_handler/3`. This function adds a handler, as we know, and also tells the event manager to monitor the current process. If the current process dies, the event handler is automatically removed. This makes sense because, in the `Forwarder` case, we should stop forwarding messages if the recipient of those messages (`self()`/the test process) is no longer alive.

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

```
defmodule KV.Registry do
  use GenServer

  ## Client API

  @doc """
  Starts the registry.
  """
  def start_link(event_manager, opts \\ []) do
    # 1. start_link now expects the event manager as argument
    GenServer.start_link(__MODULE__, event_manager, opts)
  end
```

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

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

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

  ## Server callbacks

  def init(events) do
    # 2. The init callback now receives the event manager.
    #    We have also changed the manager state from a tuple
    #    to a map, allowing us to add new fields in the future
    #    without needing to rewrite all callbacks.
    names = HashDict.new
    refs  = HashDict.new
    {:ok, %{names: names, refs: refs, events: events}}
  end

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

  def handle_cast({:create, name}, state) do
    if HashDict.get(state.names, name) do
      {:noreply, state}
    else
      {:ok, pid} = KV.Bucket.start_link()
      ref = Process.monitor(pid)
      refs = HashDict.put(state.refs, ref, name)
      names = HashDict.put(state.names, name, pid)
      # 3. Push a notification to the event manager on create
      GenEvent.sync_notify(state.events, {:create, name, pid})
      {:noreply, %{state | names: names, refs: refs}}
    end
  end

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

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

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

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

4.3 Event streams

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

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

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

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

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

5 Supervisor and Application

{% include toc.html %}

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

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

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

5.1 Our first Supervisor

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

```
defmodule KV.Supervisor do
  use Supervisor

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

  @manager_name KV.EventManager
  @registry_name KV.Registry

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

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

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

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

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

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

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

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

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

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

5.2 Understanding applications

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

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

```
{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` and `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 automatically for us.

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

5.2.1 Starting applications

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

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

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

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

Oops, it's already started.

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

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

Now we get an error because an application that `:kv` depends on (`:logger` in this case) hasn't been 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. But since we passed the `--no-start` flag, 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]}
iex> Application.stop(:kv)
18:12:10.698 [info] Application kv exited :stopped
:ok
```

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

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

5.2.2 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: [],
   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 only need to define a `start/2` function. If we wanted to specify custom behaviour on application stop, we could define a `stop/1` function, as well. In this case, the one automatically defined by `use Application` is fine.

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

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

Excellent!

5.2.3 Projects or applications?

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

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

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

5.3 Simple one for one supervisors

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

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

```
{:ok, pid} = KV.Bucket.start_link()
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 test:

```
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)
  assert_receive {:exit, "shopping", ^bucket}
  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. Instead of using `Agent.stop/1`, we are sending an exit signal to shutdown the bucket. Since the bucket is linked to the registry, which is then linked to the test process, killing the bucket causes the registry to crash which then causes the test process to crash too:

```
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.

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

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

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

  def start_link(opts \\ []) do
```

```
    Supervisor.start_link(__MODULE__, :ok, opts)
  end

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

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

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

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

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

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

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

```
iex> {:ok, sup} = KV.Bucket.Supervisor.start_link
{:ok, #PID<0.70.0>}
iex> {:ok, bucket} = KV.Bucket.Supervisor.start_bucket(sup)
{: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. We are going to follow the same strategy we did with the events manager, where we will explicitly pass the buckets supervisor pid to `KV.Registry.start_link/3`. Let's start by changing the setup callback in `test/kv/registry_test.exs` to do so:

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

  GenEvent.add_mon_handler(manager, Forwarder, self())
  {:ok, registry: registry}
end
```

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

```
## Client API

@doc """
Starts the registry.
"""
def start_link(event_manager, buckets, opts \\ []) do
```

```
  # 1. Pass the buckets supervisor as argument
  GenServer.start_link(__MODULE__, {event_manager, buckets}, opts)
end

## Server callbacks

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

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

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

5.4 Supervision trees

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

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

```
@manager_name KV.EventManager
@registry_name KV.Registry
@bucket_sup_name KV.Bucket.Supervisor

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

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

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

Also remember that the order in which children are declared is important. Since the registry depends on the buckets

supervisor, the buckets supervisor must be listed before it in the children list.

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

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

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

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

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

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

6 ETS

{% include toc.html %}

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

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

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

6.1 ETS as a cache

ETS allows us to store any Erlang/Elixir term in an in-memory table. Working with ETS tables is done via *erlang's* ':ets'' module <http://www.erlang.org/doc/man/ets.html>'__:

```
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`, which means that only the process that created the table can write to it, but all processes can read it from it. Those are actually the default values, so we will skip them from now on.

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

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

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

```
defmodule KV.Registry do
  use GenServer

  ## Client API

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

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

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

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

  ## Server callbacks

  def init({table, events, buckets}) do
```

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

  # 4. The previous handle_call callback for lookup was removed

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

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

  def handle_info(_msg, state) do
    {:noreply, state}
  end
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 definitely broken our tests. For starters, there is a new argument we need to pass to `KV.Registry.start_link/3`. Let's start amending our tests in `test/kv/registry_test.exs` by rewriting the `setup` callback:

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

  GenEvent.add_mon_handler(manager, Forwarder, self())
  {:ok, registry: registry, ets: :registry_table}
end
```

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

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

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

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

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

to:

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

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

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

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

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

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

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

may be failing on this line:

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

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

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

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

6.2 Race conditions?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  defmodule Forwarder do
    use GenEvent

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

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

    GenEvent.add_mon_handler(manager, Forwarder, self())
    {:ok, registry: registry, ets: :registry_table}
  end

  test "sends events on create and crash", %{registry: registry, ets: ets} do
```

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

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

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

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

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

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

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

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

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

```
@manager_name KV.EventManager
@registry_name KV.Registry
@ets_registry_name KV.Registry
@bucket_sup_name KV.Bucket.Supervisor

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

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

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

conflicts.

6.3 ETS as persistent storage

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

This is extremely convenient as a default behaviour, and we can use it even more to our advantage. Remember that there is a dependency between the registry and the buckets supervisor. If the registry dies, we want the buckets supervisor to die too, because once the registry dies all information linking the bucket name to the bucket process is lost. However, what if we could keep the registry data even if the registry process crashes? If we are able to do so, we remove the dependency between the registry and the buckets supervisor, making the `:one_for_one` strategy the perfect strategy for our supevisor.

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

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

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

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

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

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

```
def init({table, events, buckets}) do
  refs = HashDict.new
  {:ok, %{names: table, refs: refs, events: events, buckets: buckets}}
end
```

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

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

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

```
  GenEvent.add_mon_handler(manager, Forwarder, self())
  registry
end
```

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

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

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

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

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

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

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

Run the new test and it will fail with:

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

That's what we expected. If the bucket is not being monitored, the registry is not notified when it dies and therefore no event is sent. We can fix this by changing `KV.Registry`'s `init/1` callback one last time to setup monitors for all existing entries in the table:

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

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

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

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

we may run into as data becomes shared between the server and all clients.

7 Dependencies and umbrella projects

{% include toc.html %}

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

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

```
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.

7.1 External dependencies

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

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

```
def deps do
  [{:plug, "~> 0.5.0"}]
end
```

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

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

```
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, dependecies are automatically compiled for you. You can read more about deps by typing `mix help deps`, and in the documentation for the Mix.Tasks.Deps module.

7.2 Internal dependencies

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

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

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

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

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

However, using git dependencies for internal dependencies is somewhat discouraged in Elixir. Remember that the runtime and the Elixir ecosystem already 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 can become very hard to maintain, because now you will have to spend a lot of time managing those git repositories rather than writing your code.

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

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

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

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

So let's get started!

7.3 Umbrella projects

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

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

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

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

  def project do
    [apps_path: "apps",
     deps: deps]
  end

  defp deps do
    []
  end
end
```

What makes this project different from the previous one is simply the `apps_path: "apps"` entry in the project definition. This means this project will act as an umbrella. Such projects do not have source files nor tests, although they can have dependencies which are only available for themselves. We'll create new application projects inside the apps directory. We call these applications "umbrella children".

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

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

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

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

  def project do
    [app: :kv_server,
     version: "0.0.1",
     deps_path: "../../deps",
     lockfile: "../../mix.lock",
     elixir: "~> 0.14.1-dev",
     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 two lines to the project definition:

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

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

```
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.

7.4 In umbrella dependencies

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

```
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:

```
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. Each application has its independent configuration, and dependencies in between them must be explicitly listed. This allows them to be developed together, but compiled, tested and deployed independently if desired.

7.5 Summing up

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

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

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

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

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

8 Task and gen_tcp

{% include toc.html %}

In this chapter, we are going to learn how to use *Erlang's* ':gen_tcp'' module <http://erlang.org/doc/man/gen_tcp.html>'__ to serve requests. In future chapters we will expand our server so it can actually serve the commands. This will also provide a great opportunity to explore Elixir's `Task` module.

8.1 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:

```
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` - block on `:gen_tcp.recv/2` until data is available
  #
  {:ok, socket} = :gen_tcp.listen(port,
                    [:binary, packet: :line, active: false])
  IO.puts "Accepting connections on port #{port}"
  loop_acceptor(socket)
end
```

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

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

  serve(client)
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 pipeline operator* ‘‘|>‘ </docs/stable/elixir/Kernel.html#|>/2>‘__ to express this flow of operations. The pipeline 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)

When using the ``|>`` operator, it is important to add parentheses
to the function calls due to how operator precedence works. In
particular, this code:

::

    1..10 |> Enum.filter &(&1 <= 5) |> Enum.map &(&1 * 2)

Actually translates to:

::

    1..10 |> Enum.filter(&(&1 <= 5) |> Enum.map(&(&1 * 2)))

Which is not what we want, since the function given to
``Enum.filter/2`` is the one passed as first argument to
``Enum.map/2``. The solution is to use explicit parentheses:

::
```

```
    1..10 |> Enum.filter(&(&1 <= 5)) |> Enum.map(&(&1 * 2))
```

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

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

Start an iex session inside the `kv_server` application with `iex -S mix`. 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 <http://en.wikipedia.org/wiki/Telnet>'__ 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 `<Enter>`, 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 inside a supervision tree.

8.2 Tasks

We have learned about agents, generic servers, and event managers. 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 `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.

8.3 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
```

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` 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 tasks already come 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 </docs/stable/elixir/Task.Supervisor.html>`__ 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/2` to use `Task.Supervisor` to serve each request:

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

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

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

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

    opts = [strategy: :one_for_one, name: KVServer.Supervisor]
```

```
    Supervisor.start_link(children, opts)
  end

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

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

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

    serve(socket)
  end

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

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

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

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

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

9 Docs, tests and pipelines

{% include toc.html %}

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.

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

Also note that we started the documentation string using `@doc ~S"""`. 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 </docs/stable/ex_unit/ExUnit.DocTest.html>‘__.

9.2 Pipelines

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

```
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
```

with 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, msg) do
  :gen_tcp.send(socket, format_msg(msg))
end

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

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 pipes which made the logic straight-forward to understand:

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

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

Thankfully, there is a project called elixir-pipes that provides exactly this functionality! Let's give it a try.

Open up your `apps/kv_server/mix.exs` file and change both `application/0` and `deps/0` functions to the following:

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

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

Run `mix deps.get` to get the dependency, and rewrite the `serve/1` function to use the `pipe_matching/3` functionality now available to us:

```
defp serve(socket) do
  import Pipe

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

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

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

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

9.3 Running commands

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

```
@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.

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 `format_msg/1` function in `KV.Server` to nicely show not found messages too:

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

And our server functionality is almost complete! We just need to add tests. This time, we have left tests for last because there are some important considerations to be 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 been chosing the unit test approach. For example, in order to make `KVServer.Command.run/1` testable as a unit we would need to change its implementation to not send commands directly to the `KV.Registry` process but instead pass a server as argument. This means we would need to change `run`'s signature to `def run(command, pid)` and the implementation for the `:create` command would look like:

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

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.

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

Let's implement our 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 with reason :stopped
```

If desired, we can avoid printing this warning by turning the error_logger off and on in the test setup:

```
setup do
  Logger.remove_backend(:console)
  Application.stop(:kv)
  :ok = Application.start(:kv)
  Logger.add_backend(:console, flush: true)
  :ok
end
```

With this simple integration test, we start to see why integration tests may be slow. Not only can this particular test not be 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.

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

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

10 Distributed tasks and configuration

{% include toc.html %}

In this last chapter, we will go back to the `:kv` application and add a routing layer that allows 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.

10.1 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 `jv@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(:"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 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 the 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.

10.2 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 </docs/stable/elixir/Task.Supervisor.html>`__ we have used in previous chapters. We just need to call `Task.Supervisor.async/2` instead of `Task.Supervisor.start_child/2` and use `Task.await/2` to read the result later on.

10.3 Distributed tasks

Distributed tasks are exactly the same as supervised tasks. The only difference is that we pass the node name when spawning the task on the supervisor. Open up `lib/kv/supervisor.ex` from the `:kv` application. Let's add a task supervisor to 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 is straightforward: it simply gets the name of the node the task is running on. With this knowledge in hand, let's finally write the routing code.

10.4 Routing layer

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

```
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
      sup = {KV.RouterTasks, elem(entry, 1)}
      Task.Supervisor.async(sup, fn ->
        KV.Router.route(bucket, mod, fun, args)
      end) |> Task.await()
    end
  end

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

  @doc """
  The routing table.
  """
  def table do
    # Replace computer-name 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 accross nodes" do
    assert KV.Router.route("hello", Kernel, :node, []) ==
           :"foo@computer-name"
    assert KV.Router.route("world", Kernel, :node, []) ==
           :"bar@computer-name"
  end

  test "raises on unknown entries" do
    assert_raise RuntimeError, ~r/could not find entry/, fn ->
      KV.Router.route(<<0>>, Kernel, :node, [])
    end
  end
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. Let's restart the node named `bar`, which is going to be used by tests:

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

And now run tests with:

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

Our test should successfuly pass. Excellent!

10.5 Test filters and tags

Although our tests pass, our testing structure is getting more complex. In particular, running tests 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.

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

```
@tag :distributed
test "route requests accross 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
```

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 </docs/stable/ex_unit/ExUnit.Case.html>‘__.

10.6 Application environment and configuration

So far we have hardcoded the routing table into the `KV.Router` module. However, we would like to make the table dynamic. 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 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.get_env(:kv, :routing_table)
end
```

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

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

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

The interesting thing about the application environment is that it can be configured not only for the current application, but for all 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 configure our `:routing_table` directly in the `config/config.exs` file as well:

```
# 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.

Each application has its own `config/config.exs` file and they are not shared in any way. Configuration can also be set per environment. Read the contents of the config file for the `:kv` application for more information on how to do so.

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

```
import_config "../apps/kv/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. We 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

10.7 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, event managers, supervisors, tasks, agents, applications and more. Not only that, we have written tests for the whole application, getting familiar with ExUnit, and learned how to use the Mix build tool to accomplish a wide range of tasks.

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.

Meta-programming in Elixir

1 Quote and unquote

{% include toc.html %}

An Elixir program can be represented by its own data structures. In this chapter, we will learn what those structures look like and how to compose them. The concepts learned in this chapter are the building blocks for macros, which we are going to take a deeper look at in the next chapter.

1.1 Quoting

The building block of an Elixir program is a tuple with three elements. For example, the function call `sum(1, 2, 3)` is represented internally as:

```
{:sum, [], [1, 2, 3]}
```

You can get the representation of any expression by using the `quote` macro:

```
iex> quote do: sum(1, 2, 3)
{:sum, [], [1, 2, 3]}
```

The first element is the function name, the second is a keyword list containing metadata and the third is the arguments list.

Operators are also represented as such tuples:

```
iex> quote do: 1 + 2
{:+, [context: Elixir, import: Kernel], [1, 2]}
```

Even a map is represented as a call to `%{}`:

```
iex> quote do: %{1 => 2}
{:%{}, [], [{1, 2}]}
```

Variables are also represented using such triplets, except the last element is an atom, instead of a list:

```
iex> quote do: x
{:x, [], Elixir}
```

When quoting more complex expressions, we can see that the code is represented in such tuples, which are often nested inside each other in a structure resembling a tree. Many languages would call such representations an Abstract Syntax Tree (AST). Elixir calls them quoted expressions:

```
iex> quote do: sum(1, 2 + 3, 4)
{:sum, [], [1, {:+, [context: Elixir, import: Kernel], [2, 3]}, 4]}
```

Sometimes when working with quoted expressions, it may be useful to get the textual code representation back. This can be done with `Macro.to_string/1`:

```
iex> Macro.to_string(quote do: sum(1, 2 + 3, 4))
"sum(1, 2 + 3, 4)"
```

In general, the tuples above are structured according to the following format:

```
{tuple | atom, list, list | atom}
```

- The first element is an atom or another tuple in the same representation;
- The second element is a keyword list containing metadata, like numbers and contexts;
- The third element is either a list of arguments for the function call or an atom. When this element is an atom, it means the tuple represents a variable.

Besides the tuple defined above, there are five Elixir literals that, when quoted, return themselves (and not a tuple). They are:

```
:sum          #=> Atoms
1.0           #=> Numbers
[1, 2]        #=> Lists
"strings"     #=> Strings
{key, value} #=> Tuples with two elements
```

Most Elixir code has a straight-forward translation to its underlying quoted expression. We recommend you try out different code samples and see what the results are. For example, what does `String.upcase("foo")` expand to? We have also learned that `if(true, do: :this, else: :that)` is the same as `if true do :this else :that end`. How does this affirmation hold with quoted expressions?

1.2 Unquoting

Quote is about retrieving the inner representation of some particular chunk of code. However, sometimes it may be necessary to inject some other particular chunk of code inside the representation we want to retrieve.

For example, imagine you have a variable `number` which contains the number you want to inject inside a quoted expression.

```
iex> number = 13
iex> Macro.to_string(quote do: 11 + number)
"11 + number"
```

That's not what we wanted, since the value of the `number` variable has not been injected and `number` has been quoted in the expression. In order to inject the *value* of the `number` variable, `unquote` has to be used inside the quoted representation:

```
iex> number = 13
iex> Macro.to_string(quote do: 11 + unquote(number))
"11 + 13"
```

`unquote` can even be used to inject function names:

```
iex> fun = :hello
iex> Macro.to_string(quote do: unquote(fun)(:world))
"hello(:world)"
```

In some cases, it may be necessary to inject many values inside a list. For example, imagine you have a list containing `[1, 2, 6]` and we want to inject `[3, 4, 5]` into it. Using `unquote` won't yield the desired result:

```
iex> inner = [3, 4, 5]
iex> Macro.to_string(quote do: [1, 2, unquote(inner), 6])
"[1, 2, [3, 4, 5], 6]"
```

That's when `unquote_splicing` becomes handy:

```
iex> inner = [3, 4, 5]
iex> Macro.to_string(quote do: [1, 2, unquote_splicing(inner), 6])
"[1, 2, 3, 4, 5, 6]"
```

Unquoting is very useful when working with macros. When writing macros, developers are able to receive code chunks and inject them inside other code chunks, which can be used to transform code or write code that generates code during compilation.

1.3 Escaping

As we saw at the beginning of this chapter, only some values are valid quoted expressions in Elixir. For example, a map is not a valid quoted expression. Neither is a tuple with four elements. However, such values *can* be expressed as a quoted expression:

```
iex> quote do: %{1 => 2}
{:%{}, [], [{1, 2}]}
```

In some cases, you may need to inject such *values* into *quoted expressions*. To do that, we need to first escape those values into quoted expressions with the help of `Macro.escape/1`:

```
iex> map = %{hello: :world}
iex> Macro.escape(map)
{:%{}, [], [hello: :world]}
```

Macros receive quoted expressions and must return quoted expressions. However, sometimes during the execution of a macro, you may need to work with values and making a distinction between values and quoted expressions will be required.

In other words, it is important to make a distinction between a regular Elixir value (like a list, a map, a process, a reference, etc) and a quoted expression. Some values, such as integers, atoms and strings, have a quoted expression equal to the value itself. Other values, like maps, need to be explicitly converted. Finally, values like functions and references cannot be converted to a quoted expression at all.

You can read more about `quote` and `unquote` in the `` `Kernel.SpecialForms `` module </docs/stable/elixir/Kernel.SpecialForms.html>‘__. Documentation for `Macro.escape/1` and other functions related to quoted expressions can be found in the `` `Macro `` module </docs/stable/elixir/Macro.html>‘__.

In this introduction we have laid the groundwork to finally write our first macro, so let’s move to the next chapter.

2 Macros

{% include toc.html %}

Macros can be defined in Elixir using `defmacro/2`.

> For this chapter, we will be using files instead of running code samples in IEx. That’s because the code samples will span multiple lines of code and typing them all in IEx can be counter-productive. You should be able to run the code samples by saving them into a `macros.exs` file and running it with `elixir macros.exs` or `iex macros.exs`.

2.1 Our first macro

In order to better understand how macros work, let’s create a new module where we are going to implement `unless`, which does the opposite of `if`, as a macro and as a function:

```
defmodule Unless do
  def fun_unless(clause, expression) do
    if(!clause, do: expression)
  end

  defmacro macro_unless(clause, expression) do
    quote do
      if(!unquote(clause), do: unquote(expression))
    end
  end
end
```

The function receives the arguments and passes them to `if`. However, as we learned in the previous chapter, the macro will receive quoted expressions, inject them into the quote, and finally return another quoted expression.

Let's start `iex` with the module above:

```
$ iex macros.exs
```

And play with those definitions:

```
iex> require Unless
iex> Unless.macro_unless true, IO.puts "this should never be printed"
nil
iex> Unless.fun_unless true, IO.puts "this should never be printed"
"this should never be printed"
nil
```

Note that in our macro implementation, the sentence was not printed, although it was printed in our function implementation. That's because the arguments to a function call are evaluated before calling the function. However, macros do not evaluate their arguments. Instead, they receive the arguments as quoted expressions which are then transformed into other quoted expressions. In this case, we have rewritten our `unless` macro to become an `if` behind the scenes.

In other words, when invoked as:

```
Unless.macro_unless true, IO.puts "this should never be printed"
```

Our `macro_unless` macro received the following:

{% raw %}

```
macro_unless(true, {{:., [], [{:aliases, [], [:IO]}, :puts]}, [], ["this should never␣
↪be printed"]})
```

{% endraw %}

And it then returned a quoted expression as follows:

{% raw %}

```
{:if, [], [
  {:!, [], [true]},
  {{:., [], [IO, :puts], [], ["this should never be printed"]}}]}
```

{% endraw %}

We can actually verify that this is the case by using `Macro.expand_once/2`:

```
iex> expr = quote do: Unless.macro_unless(true, IO.puts "this should never be printed
↪")
iex> res  = Macro.expand_once(expr, __ENV__)
iex> IO.puts Macro.to_string(res)
if(!true) do
  IO.puts("this should never be printed")
end
:ok
```

`Macro.expand_once/2` receives a quoted expression and expands it according to the current environment. In this case, it expanded/invoked the `Unless.macro_unless/2` macro and returned its result. We then proceeded to convert the returned quoted expression to a string and print it (we will talk about `__ENV__` later in this chapter).

That's what macros are all about. They are about receiving quoted expressions and transforming them into something else. In fact, `unless/2` in Elixir is implemented as a macro:

```
defmacro unless(clause, options) do
  quote do
    if(!unquote(clause), do: unquote(options))
  end
end
```

Constructs such as `unless/2`, `defmacro/2`, `def/2`, `defprotocol/2`, and many others used throughout this getting started guide are implemented in pure Elixir, often as a macros. This means that the constructs being used to build the language can be used by developers to extend the language to the domains they are working on.

We can define any function and macro we want, including ones that override the built-in definitions provided by Elixir. The only exceptions are Elixir special forms which are not implemented in Elixir and therefore cannot be overridden, *the full list of special forms is available in ‘‘Kernel.SpecialForms‘* </docs/stable/elixir/Kernel.SpecialForms.html>‘__.

2.2 Macros hygiene

Elixir macros have late resolution. This guarantees that a variable defined inside a quote won't conflict with a variable defined in the context where that macro is expanded. For example:

```
defmodule Hygiene do
  defmacro no_interference do
    quote do: a = 1
  end
end

defmodule HygieneTest do
  def go do
    require Hygiene
    a = 13
    Hygiene.no_interference
    a
  end
end

HygieneTest.go
# => 13
```

In the example above, even though the macro injects `a = 1`, it does not affect the variable `a` defined by the `go` function. If a macro wants to explicitly affect the context, it can use `var!`:

```
defmodule Hygiene do
  defmacro interference do
    quote do: var!(a) = 1
  end
end

defmodule HygieneTest do
  def go do
    require Hygiene
    a = 13
    Hygiene.interference
    a
  end
end

HygieneTest.go
# => 1
```

Variable hygiene only works because Elixir annotates variables with their context. For example, a variable `x` defined on line 3 of a module would be represented as:

```
{:x, [line: 3], nil}
```

However, a quoted variable is represented as:

```
defmodule Sample do
  def quoted do
    quote do: x
  end
end

Sample.quoted #=> {:x, [line: 3], Sample}
```

Notice that the third element in the quoted variable is the atom `Sample`, instead of `nil`, which marks the variable as coming from the `Sample` module. Therefore, Elixir considers these two variables as coming from different contexts and handles them accordingly.

Elixir provides similar mechanisms for imports and aliases too. This guarantees that a macro will behave as specified by its source module rather than conflicting with the target module where the macro is expanded. Hygiene can be bypassed under specific situations by using macros like `var!/2` and `alias!/2`, although one must be careful when using those as they directly change the user environment.

Sometimes variable names might be dynamically created. In such cases, `Macro.var/2` can be used to define new variables:

```
defmodule Sample do
  defmacro initialize_to_char_count(variables) do
    Enum.map variables, fn(name) ->
      var = Macro.var(name, nil)
      length = name |> Atom.to_string |> String.length
      quote do
        unquote(var) = unquote(length)
      end
    end
  end

  def run do
    initialize_to_char_count [:red, :green, :yellow]
```

```
    [red, green, yellow]
  end
end

> Sample.run #=> [3, 5, 6]
```

Take note of the second argument to `Macro.var/2`. This is the context being used and will determine hygiene as described in the next section.

2.3 The environment

When calling `Macro.expand_once/2` earlier in this chapter, we used the special form `__ENV__`.

`__ENV__` returns an instance of the `Macro.Env` struct which contains useful information about the compilation environment, including the current module, file and line, all variables defined in the current scope, as well as imports, requires and so on:

```
iex> __ENV__.module
nil
iex> __ENV__.file
"iex"
iex> __ENV__.requires
[IEx.Helpers, Kernel, Kernel.Typespec]
iex> require Integer
nil
iex> __ENV__.requires
[IEx.Helpers, Integer, Kernel, Kernel.Typespec]
```

Many of the functions in the `Macro` module expect an environment. You can read more about these functions in *the docs for ``Macro`* module </docs/stable/elixir/Macro.html>`__ and learn more about the compilation environment in the *docs for ``Macro.Env`* </docs/stable/elixir/Macro.Env.html>`__.

2.4 Private macros

Elixir also supports private macros via `defmacrop`. As private functions, these macros are only available inside the module that defines them, and only at compilation time.

It is important that a macro is defined before its usage. Failing to define a macro before its invocation will raise an error at runtime, since the macro won't be expanded and will be translated to a function call:

```
iex> defmodule Sample do
...>  def four, do: two + two
...>  defmacrop two, do: 2
...> end
** (CompileError) iex:2: function two/0 undefined
```

2.5 Write macros responsibly

Macros are a powerful construct and Elixir provides many mechanisms to ensure they are used responsibly.

- Macros are hygienic: by default, variables defined inside a macro are not going to affect the user code. Furthermore, function calls and aliases available in the macro context are not going to leak into the user context.
- Macros are lexical: it is impossible to inject code or macros globally. In order to use a macro, you need to explicitly `require` or `import` the module that defines the macro.

- Macros are explicit: it is impossible to run a macro without explicitly invoking it. For example, some languages allow developers to completely rewrite functions behind the scenes, often via parse transforms or via some reflection mechanisms. In Elixir, a macro must be explicitly invoked in the caller.
- Macros' language is clear: many languages provide syntax shortcuts for `quote` and `unquote`. In Elixir, we preferred to have them explicitly spelled out, in order to clearly delimit the boundaries of a macro definition and its quoted expressions.

Even if Elixir attempts its best to provide a safe environment, the major responsibility still falls on the developers. That's why the first rule of the macro club is **write macros responsibly**. Macros are harder to write than ordinary Elixir functions and it's considered to be bad style to use them when they're not necessary. Elixir already provides elegant mechanisms to write your every day code and macros should be saved as a last resort.

If you ever need to resort to macros, remember that macros are not your API. Keep your macro definitions short, including their quoted contents. For example, instead of writing a macro like this:

```
defmodule MyModule do
  defmacro my_macro(a, b, c) do
    quote do
      do_this(unquote(a))
      ...
      do_that(unquote(b))
      ...
      and_that(unquote(c))
    end
  end
end
```

write:

```
defmodule MyModule do
  defmacro my_macro(a, b, c) do
    quote do
      # Keep what you need to do here to a minimum
      # and move everything else to a function
      do_this_that_and_that(unquote(a), unquote(b), unquote(c))
    end
  end

  def do_this_that_and_that(a, b, c) do
    do_this(a)
    ...
    do_that(b)
    ...
    and_that(c)
  end
end
```

This makes your code clearer and easier to test and maintain, as you can invoke `do_this_that_and_that/3` directly. It also helps you design an actual API for developers that does not rely on macros.

With those lessons, we finish our introduction to macros. The next chapter is a brief discussion on DSLs that shows how we can mix macros and module attributes to annotate and extend modules and functions.

3 Domain Specific Languages

{% include toc.html %}

Domain Specific Languages allow developers to tailor their application to a particular domain. There are many language features that, when used in combination, can aid developers to write Domain Specific Languages. In this chapter we will focus on how macros and module attributes can be used together to create domain specific modules that are focused on solving one particular problem. As an example, we will write a very simple module to define and run tests.

The goal is to build a module named `TestCase` that allows us to write the following:

```
defmodule MyTest do
  use TestCase

  test "arithmetic operations" do
    4 = 2 + 2
  end

  test "list operations" do
    [1, 2, 3] = [1, 2] ++ [3]
  end
end

MyTest.run
```

In the example above, by using `TestCase`, we can write tests using the `test` macro, which defines a function named `run` to automatically run all tests for us. Our prototype will simply rely on the match operator (=) as a mechanism to do assertions.

3.1 The `test` macro

Let's start by creating a module that simply defines and imports the `test` macro when used:

```
defmodule TestCase do
  # Callback invoked by `use`.
  #
  # For now it simply returns a quoted expression that
  # imports the module itself into the user code.
  @doc false
  defmacro __using__(_opts) do
    quote do
      import TestCase
    end
  end

  @doc """
  Defines a test case with the given description.

  ## Examples

      test "arithmetic operations" do
        4 = 2 + 2
      end

  """
  defmacro test(description, do: block) do
    function_name = String.to_atom("test " <> description)
    quote do
      def unquote(function_name)(), do: unquote(block)
    end
  end
```

```
end
```

Assuming we defined `TestCase` in a file named `tests.exs`, we can open it up by running `iex tests.exs` and define our first tests:

```
iex> defmodule MyTest do
...>   use TestCase
...>
...>   test "hello" do
...>     "hello" = "world"
...>   end
...> end
```

For now we don't have a mechanism to run tests, but we know that a function named "test hello" was defined behind the scenes. When we invoke it, it should fail:

```
iex> MyTest."test hello"()
** (MatchError) no match of right hand side value: "world"
```

3.2 Storing information with attributes

In order to finish our `TestCase` implementation, we need to be able to access all defined test cases. One way of doing this is by retrieving the tests at runtime via `__MODULE__.__info__(:functions)`, which returns a list of all functions in a given module. However, considering that we may want to store more information about each test besides the test name, a more flexible approach is required.

When discussing module attributes in earlier chapters, we mentioned how they can be used as temporary storage. That's exactly the property we will apply in this section.

In the `__using__/1` implementation, we will initialize a module attribute named `@tests` to an empty list, then store the name of each defined test in this attribute so the tests can be invoked from the `run` function.

Here is the updated code for the `TestCase` module:

```
defmodule TestCase do
  @doc false
  defmacro __using__(_opts) do
    quote do
      import TestCase

      # Initialize @tests to an empty list
      @tests []

      # Invoke TestCase.__before_compile__/1 before the module is compiled
      @before_compile TestCase
    end
  end

  @doc """
  Defines a test case with the given description.

  ## Examples

      test "arithmetic operations" do
        4 = 2 + 2
      end
```

```
  """
  defmacro test(description, do: block) do
    function_name = String.to_atom("test " <> description)
    quote do
      # Prepend the newly defined test to the list of tests
      @tests [unquote(function_name)|@tests]
      def unquote(function_name)(), do: unquote(block)
    end
  end

  # This will be invoked right before the target module is compiled
  # giving us the perfect opportunity to inject the `run/0` function
  @doc false
  defmacro __before_compile__(env) do
    quote do
      def run do
        Enum.each @tests, fn name ->
          IO.puts "Running #{name}"
          apply(__MODULE__, name, [])
        end
      end
    end
  end
end
```

By starting a new IEx session, we can now define our tests and run them:

```
iex> defmodule MyTest do
...>   use TestCase
...>
...>   test "hello" do
...>     "hello" = "world"
...>   end
...> end
iex> MyTest.run
Running test hello
** (MatchError) no match of right hand side value: "world"
```

Although we have overlooked some details, this is the main idea behind creating domain specific modules in Elixir. Macros enable us to return quoted expressions that are executed in the caller, which we can then use to transform code and store relevant information in the target module via module attributes. Finally, callbacks such as `@before_compile` allow us to inject code into the module when its definition is complete.

Besides `@before_compile`, there are other useful module attributes like `@on_definition` and `@after_compile`, which you can read more about in *the docs for the ‘‘Module‘* module </docs/stable/elixir/Module.html>‘__. You can also find useful information about macros and the compilation environment in the documentation for the `Macro` module </docs/stable/elixir/Macro.html>‘__ and `Macro.Env` </docs/stable/elixir/Macro.Env.html>‘__.

CHAPTER 2

Technical guides

Scoping Rules in Elixir (and Erlang)

For everyday use it is sufficient to understand the basics of scoping rules in Elixir: that there's the top level scope and function clause scope, and that named functions have their own peculiar differences from the more conventional anonymous functions.

But there are, in fact, quite a few rules you need to know to get a complete picture of the way scopes work in Elixir. In this technical article we will take a close look at all of the scoping rules and learn in what ways they differ from Erlang.

Types of Scope

In Elixir there are two types of scope:

- the top level scope
- function clause scope

There are a number of constructs that create new scope:

- modules and module-like structures: `defmodule`, `defprotocol`, `defimpl`
- functions: `fn`, `def`, `defp`
- comprehensions: `for`
- `try` block bodies

Most of the time user code in Elixir is structured in the following way. At the top level we define modules. Each module contains a number of attributes and function clauses. Inside a function clause there can be arbitrary number of expressions including control flow constructs like `case`, `if`, or `try`:

```
abc = "abc"             T ----------------------+
                                                |
defmodule M do              M ----------------+ |
  @doc "factorial"                            | |
  @limit 13                                   | |
                                              | |
  def foo(n) do                F ----------+  | |
    x = case n do                           |  | |  # T: top level scope
      0 -> 1                                |  | |
      i when i > 0 -> n * foo(n - 1)        |  | |  # M: module's scope
      _ -> :undef                           |  | |
    end                                     |  | |  # F: function clause scope
                                            |  | |
    for x <- [1,2,3] do             C ---+  |  | |  # C: comprehension's scope
      -x                                |  |  | |
    end                                 -----+  | |
                                                |  |
  end                                ------------+ |
                                                   |
end                                 ----------------+
                                  ------------------------+
```

Another way to visualise that structure, schematically:

```
# Figure 1

+-----------------------------------------------------------+
| Top level                                                 |
|                                                           |
|  +-------------------------+     +-------------------------+ |
|  | Module                  |     | Module                  | |
|  |                         |     |                         | |
|  | +---------------------+ |     | +---------------------+ | |
|  | | Function clause     | |     | | Function clause     | | |
|  | |                     | |     | |                     | | |
|  | | +-----------------+ | |     | | +-----------------+ | | |
|  | | | Comprehension   | | |     | | | Comprehension   | | | |
|  | | +-----------------+ | |     | | +-----------------+ | | |
|  | | +-----------------+ | | ... | | +-----------------+ | | |
```

```
|   | | | Anon. function | | |      | | | Anon. function | | | |
|   | | +----------------+ | |      | | +----------------+ | | |
|   | | +----------------+ | |      | | +----------------+ | | |
|   | | | Try block      | | |      | | | Try block      | | | |
|   | | +----------------+ | |      | | +----------------+ | | |
|   | +--------------------+ |      | +--------------------+ | |
|   +------------------------+      +------------------------+ |
|                                                              |
+--------------------------------------------------------------+
```

When working in the interactive shell, the scope hierarchy is usually flat ("function clause" in the graphic below now refers to anonymous functions instead of named functions):

```
# Figure 2

+-----------------------+
| Top level             |
|                       |
|  +-----------------+  |
|  | Module          |  |
|  +-----------------+  |
|  +-----------------+  |
|  | Function clause |  |
|  +-----------------+  |
|  +-----------------+  |
|  | Comprehension   |  |
|  +-----------------+  |
|  +-----------------+  |
|  | Anon. function  |  |
|  +-----------------+  |
|  +-----------------+  |
|  | Try block       |  |
|  +-----------------+  |
|                       |
+-----------------------+
```

Those are the two most commonly seen structures for code organisation in Elixir.

In the general case, however, all scopes are arbitrarily nestable: we could imagine a `case` expression inside a comprehension or a top-level `if` expression defining different modules depending on some condition. For example:

```
f = fn x ->
  case x do
    1 ->
      defmodule M do
        def say do
          "one"
        end
      end
    2 ->
     defmodule N do
        def say do
          "two"
        end
      end
  end
end
```

```
# no module has been defined yet
M.say       #=> undefined function: M.say/0
N.say       #=> undefined function: N.say/0

# define M
f.(1)
M.say       #=> "one"
N.say       #=> undefined function: N.say/0

# define N
f.(2)
M.say       #=> "one"
N.say       #=> "two"
```

In order to understand how the example above works, you should be aware of the fact the a module definition creates the module as its side-effect, so the module itself will be available globally. Only the name of the module is affected by the nesting of the `defmodule` call as we'll see later in this article.

Elixir Scopes Are Lexical

This means that it is possible to determine the scope of every identifier only by looking at the source code.

All variable bindings introduced in a scope are available until the end of that scope. Elixir has a few special forms that treat scopes a little differently (namely `require`, `import`, and `alias`). We will examine them at the end of this article.

Scope Nesting and Shadowing

According to the rules of lexical scope, any variables defined in the surrounding scope are accessible in all other scopes it contains.

In **Figure 1** above, any variable defined in the top level scope will be accessible in the module's scope and any scope nested inside it, and so on.

There is an exception to this rule which applies only to named functions: any variable coming from the surrounding scope has to be unquoted inside a function clause body.

Any variable in a nested scope whose name coincides with a variable from the surrounding scope will shadow that outer variable. In other words, the variable inside the nested scope temporarily hides the variable from the surrounding scope, but does not affect it in any way.

The Top Level Scope

The top level scope includes every variable and identifier defined outside of any other scope.

```
x         #=> undefined function: x/0

x = 1
x         #=> 1

f = fn -> x end
f.()      #=> 1
```

Named functions cannot be defined at the top level because a named function always belongs within a module. However, named functions can be imported into any lexical scope (including the top level scope) like this:

```
import String, only: [reverse: 1]

reverse "Hello"  #=> "olleH"
```

In fact, all functions and macros from the `Kernel` module are autoimported in the top level scope by the compiler.

Function Clause Scope

Each function clause defines a new lexical scope: any new variable bound inside it will not be available outside of that clause:

```
defmodule M do
  def foo(x), do: -x

  # this 'x' is completely independent from the one in 'foo/1'
  def bar(x), do: 2*x

  x = 1

  # shadowing in action: the 'x' in the argument list creates a variable
  # local to the function clause's body and has nothing to do with the
  # previously defined 'x'
  f = fn(x) ->
    x = x + 1
  end

  y = f.(x)
  IO.puts "The correct answer is #{y} == #{f.(x)}"
  # output: The correct answer is 2 == 2

  # in this case the argument 'y' shadows the named function 'y/0'
  def y(y), do: y*2

  # here the reference to 'y' inside the function
  # body is actually a recursive call to 'y/0'
  def y, do: y*2
end

M.foo 3       #=> -3
M.bar 4       #=> 8

M.y -2        #=> -4
M.y           #=> infinite loop
```

Apart from named functions, a new function clause scope is created for each module-like block, anonymous function, `try` block body, or comprehension body (see below).

```
f = fn(x) ->
  a = x - 1
end

a             #=> undefined function: a/0

g = fn(f) ->
```

```
  g = f
end

f             #=> (still the anonymous function defined above)
g             #=> (the anonymous function we've just defined)
```

Named Functions And Modules

As mentioned before, named function have a couple of peculiarities.

First, defining a named function does not introduce a new binding into the current scope:

```
defmodule M do
  def foo, do: "hi"

  foo()  # will cause CompileError: undefined function foo/0
end
```

Second, named functions cannot directly access surrounding scope, one has to use `unquote` to achieve that:

```
defmodule M do
  a = 1

  # 'a' inside unquote() unambiguously refers to 'a' defined
  # in the module's scope
  def a, do: unquote(a)

  # 'a' inside the body unambiguously refers to the function 'a/0'
  def a(b), do: a + b
end

M.a           #=> 1
M.a 3         #=> 4
```

Module scope works just like function clause scope: any variables defined between `defmodule` (or `defprotocol`, etc.) and its corresponding `end` will not be accessible outside of the module, but they will be available in the nested scopes of that module as per usual (modulo the unquoting caveat of named functions mentioned above).

It is important to understand that a module's scope exists as long as it is being compiled. In other words, variables are not "compiled into" the module. The `Module.function` syntax is only applicable to named functions and that's another thing that makes such functions special:

```
defmodule M do
  x = "hello"

  def hi, do: unquote(x)
end

M.hi          #=> "hello"
M.x           #=> undefined function: x/0
```

You may be wondering how local function calls work when named functions don't produce name bindings and don't have direct access to the surrounding scope. The answer to this lies in the following rule followed by Elixir when trying to resolve an identifier to its value:

> Any unbound identifier is treated as a local function call.

Let's see how this works in code:

```
defmodule P do
  def f, do: "I am P's f"
  def g, do: f
end

defmodule Q do
  def f, do: "I am Q's f"
  def g, do: f
end

# both P's 'g' and Q's 'g' refer to their local buddy named 'f'
P.g           #=> "I am P's f"
Q.g           #=> "I am Q's f"

# let's make 'f' local in the top level scope
f             #=> undefined function: f/0
import P
f             #=> "I am P's f"
```

One more note about module naming and nested modules: modules are always defined at the top level, no matter in what scope the actual call to `defmodule` is located. This means that as long the VM can find the .beam file with the module's code at run time, it does not matter in which scope you reference that module's name.

What the scoping does affect is the name the module will get:

```
defmodule P do
  # The actual module name will be P.Q, but it is implicitly aliased to Q
  # in P's scope
  defmodule Q do
    def q(false), do: "sorry"
    def q(true) do
      # The actual module name will be P.Q.M
      defmodule M do
        def say, do: "hi"
      end
    end
  end

  # Q is resolved to P.Q
  def foo do
    Q.q false
  end

  # At run time, this has the same exact implementation as foo
  def bar do
    P.Q.q false
  end
end

P.foo         #=> "sorry"
P.bar         #=> "sorry"
P.Q.q false   #=> "sorry"

# the module hasn't been defined yet
P.Q.M.say     #=> undefined function: P.Q.M.say/0

# after this call the P.Q.M module will become available
```

```
P.Q.q true
P.Q.M.say     #=> "hi"
```

Case-like Clauses

Control flow constructs `case`, `receive`, and `cond` share a common trait:

- any variable introduced in a clause pattern/condition will be accessible only within that clause's body
- any variable introduced inside some (but not all) clause bodies will become available in the surrounding scope (possibly with the default `nil` value)

Here are some examples of those rules in action:

```
case x do
  # both 'result' and 'a' are visible only within this clause's body
  {:ok, result}=a -> IO.inspect(result); a

  # 'error' is actually bound in the surrounding scope; its value will be nil
  # if 'x' does not match :error
  :error -> error = true

  # ordinary shadowing: this 'x' is visible only within the clause's body and
  # it doesn't affect the 'x' from the surrounding scope
  [x] -> IO.inspect(x)
end

result  #=> undefined function: result/0
a       #=> undefined function: a/0

error   #=> true if x == :error, otherwise nil
```

Note: due to a bug in the 0.12.x series, `cond`'s conditions actually leak bindings to the surrounding scope. This should be fixed in 0.13.1.

```
cond do
  a0 = false -> a = a0
  b = 1      -> b
  c = 2      -> c = 2
  true       -> d = 3
end

a       #=> false (bound to false inside the 1st condition's body)
b       #=> undefined function: b/0
c       #=> nil (the 2nd condition is truthy, so `c = 2` was not evaluated)
d       #=> nil (the body with `d = 3` was not evaluated,
        #        so 'd' also leaks with the default value)
```

```
if x = 3 do
  case y = :ok do
    :ok -> :ok
    :error -> a = "it's an error"
  end
else
  z = 11
end
```

```
x       #=> 3
y       #=> :ok
a       #=> nil
z       #=> nil
```

Try Blocks

The `try` block works similar to `case` and `receive`, but it creates new scope, so it never leaks variable bindings to the surrounding scope.

```
try do
  # all of the variables defined here are local to this block
  # (like in a function clause scope)
  a = 1
  b = a + 1
  c = d
rescue
  # these work like bindings in `case` patterns
  x in [RuntimeError] -> y = x
  x -> z = x
end

# none of the variables have leaked
a       #=> undefined function: a/0
b       #=> undefined function: b/0
c       #=> undefined function: c/0
d       #=> undefined function: d/0
x       #=> undefined function: x/0
y       #=> undefined function: y/0
z       #=> undefined function: z/0
```

Comprehensions

Comprehensions consist of two parts: the generator and the body.

Variables introduced in the generator part will only be visible within the body.

```
for a = x <- [1, 2, 3, 4], do: b = {a, x}
#=> [{1, 1}, {2, 2}, {3, 3}, {4, 4}]

a       #=> undefined function: a/0
x       #=> undefined function: x/0
```

The comprehension body itself works like function clause scope:

```
for x <- ["abc", "def"] do
  # import takes effect only within the comprehension's body
  import String, only: [reverse: 1]
  b = reverse x
end
#=> ["cba", "fed"]

b
#=> undefined function: b/0
```

```
reverse "hello"
#=> undefined function: reverse/1
```

`require`, `import`, and `alias`

All of the rules described so far apply to variable bindings. When it comes to one of these three special forms, their effect persists until the end of the `do` block they are called in. Effectively, those forms see a slightly different scope division in which control flow constructs create a new lexical scope:

```
# top level scope

defmodule M do
  # new scope
  import String, only: [reverse: 1]

  def foo do
    # new scope
    import String, only: [strip: 1]

    IO.puts reverse("abc")    # ok: inherited from the surrounding scope

    if true do
      # new scope
      import String, only: [downcase: 1]
    else
      # new scope
      import String, only: [upcase: 1]
    end

    " hello "
    |> strip      # ok: made local in the current scope with 'import'
    |> downcase   # error: no local function downcase/1
    |> upcase     # ditto
  end

  def bar do
    # new scope

    IO.puts reverse("abc")    # ok: inherited from the surrounding scope
    strip(" hello ")          # error: no local function strip/1
  end
end
```

Differences from Erlang

Most of the scoping rules described here have been inherited from Erlang.

One notable difference is that modules simply contain forms and function clauses, they don't have scope nor allow arbitrary expressions like modules in Elixir do.

There are two differences in the way case clause scope works in Erlang:

1. both bindings introduced in the pattern and in the body of a clause modify the surrounding scope
2. those variables that are bound in some (but not all) of the clauses will remain unbound in the surrounding scope (instead of getting the `nil` value like they do in Elixir); they are also called *unsafe* variables

```
case 1 of
  1=A -> B = A;
  _   -> C = 1
end.

A.  %=> 1
B.  %=> 1
C.  %=> variable 'C' is unbound
```

There is an `if` construct in Erlang that looks similar to `cond`, but works differently. It only allows guard expressions as conditions and those do not let you introduce variable bindings. Variables bound in clause bodies leak to the surrounding scope the same way they do in `case`.

```
X = 1,
if
  X -> A = X;
  true -> B = X
end.

A.  %=> variable 'A' is unbound
B.  %=> 1

%%%

Y = true,
if
  Y -> P = Y;
  true -> Q = Y
end.

P.  %=> true
Q.  %=> variable 'Q' is unbound
```

Refer to this page for more information about Erlang control flow constructs.

An assorted list of resources that describe various aspects of Erlang's scoping rules:

- Matching, Guards and Scope of Variables from Erlang's Getting Started guide.
- Scope of variables in the Erlang course.
- Static rules of variable scoping in Erlang paper
- case expression scope question on Erlang's mailing list