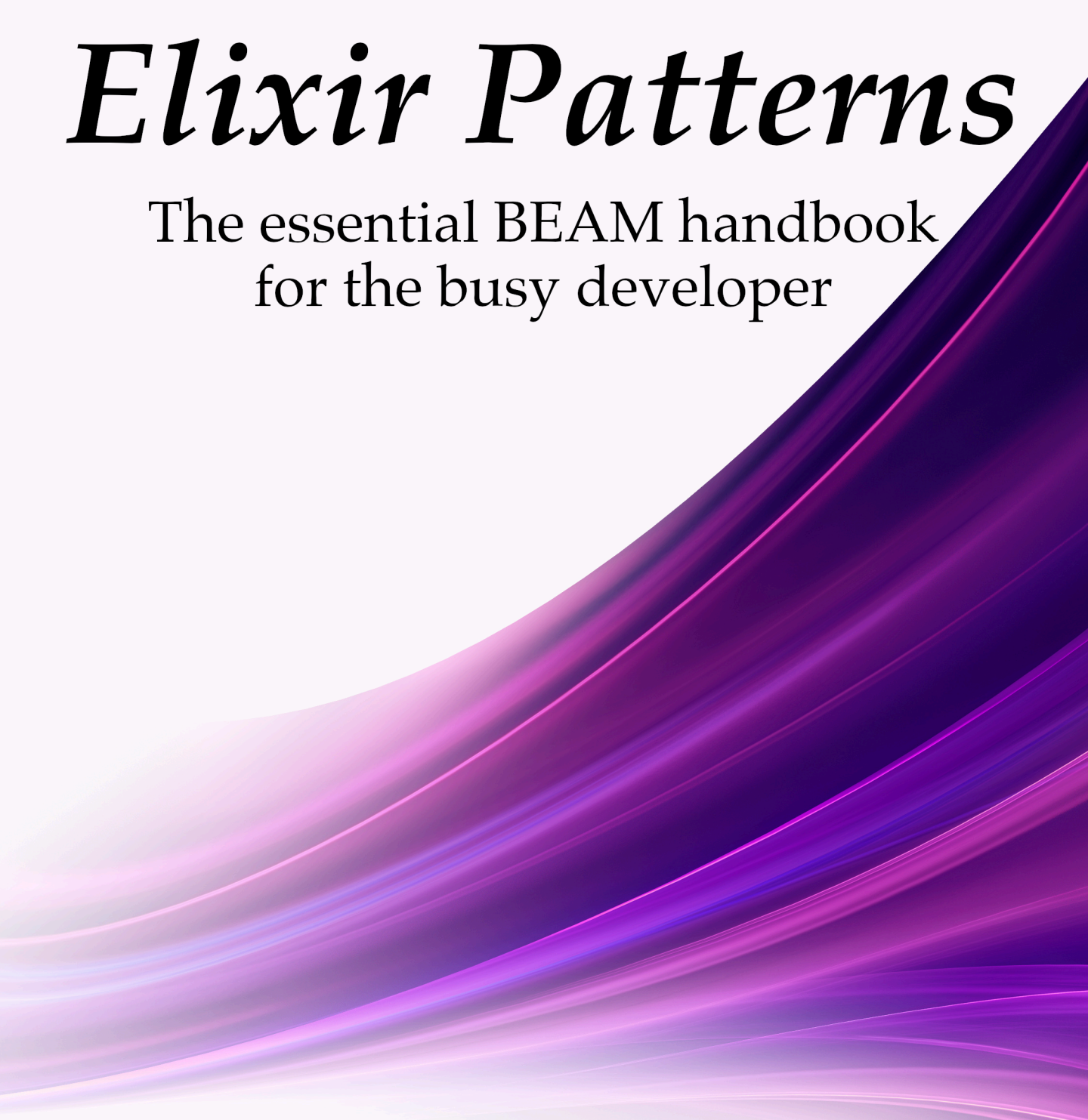
Elixir Patterns

The essential BEAM handbook
for the busy developer



Alexander Koutmos

Edited by Hugo Barauna

Elixir Patterns

The essential BEAM handbook for the busy developer

Alexander Koutmos, Hugo Barauna

Version 0.2.0, 2022-03-24

Table of Contents

Chapter 1. Introduction

Hello and welcome to Elixir Patterns! Before we begin, we would like to thank you for your support. Writing and publishing a book is a immense undertaking and it is great to know that you decided to read this particular book when there are so many great publications out there. Thank you! With that being said, we would like to discuss some of the motivations that led to this book as it will give you a sense for why it exists and how you can leverage it.

If you have ever programmed in an Object Oriented Programming language before, you have most likely heard of (or even used) some Object Oriented design patterns. Things like the Factory pattern, the Adapter pattern, the Singleton pattern, and the Builder pattern to name a few. While most of these patterns are not necessarily applicable to Functional Programming languages, the idea of having a "go-to" toolbox of patterns that you can leverage is an enticing idea. Often times, these design patterns are inherently abstract, and do not aim to leverage any of the unique properties of any specific run-time or language. As such, it can sometimes be difficult to know what patterns to use in certain circumstances.

Given that this book will be focusing on Elixir and the Erlang virtual machine, we will take a slightly different approach to design patterns. Specifically, this book aims to surface the powerful and unique characteristics of the Erlang virtual machine (or BEAM for short) and show you how you can go about solving every day problems in a simple yet scalable way.

Not only will you learn how to better leverage the tools that are at your disposal courtesy of Erlang and the BEAM, but you will also learn how to better utilize Functional Programming in order to achieve your goals in a clear and concise way.

1.1. What You Will Learn

In the first few chapters you will learn about the Erlang and Elixir standard libraries

and useful ways that they can be used. This will include covering some of the data structure modules that are available to you via Erlang and some other utility modules that can come in handy in certain situations. You'll then be introduced to a handful for Elixir modules, and you'll learn about some clever ways that they can be used. Some of these modules include the Task, Enum and Stream modules.

Once you have a good sense for the Elixir and Erlang fundamentals, you'll get into the meat and potatoes of the book and learn about processes, GenServers, Supervisors and how they work exactly. After you have a good grasp on how to write and incorporate GenServers into your application, you'll learn about some common patterns and when to reach for these patterns when you write your own applications.

After that, we'll go over how to package Elixir applications into a release and how to ensure that our GenServers and applications are configured properly. Specifically, we'll go over how we can configure our various resources depending on the running environment and how we can leverage the Adapter pattern to swap out module implementations as needed.

1.2. Who is This Book for

In order to get the most out of this book, it is recommended that you have some familiarity with Elixir and understand the basics of the BEAM. If you need a refresher on the Elixir programming language or are new to it, we would recommend taking a look at the Elixir Getting Started Guide (https://elixir-lang.org/getting-started/introduction.html) prior to diving into this book.

With that being said, we think we're ready to begin! Without further ado let's dive into the Erlang standard library and learn how we can leverage it even when programming with Elixir.

Chapter 2. Erlang Standard Library Part 1

2.1. Introduction

As an Elixir programmer, it is crucial that you understand how to use some of the underlying tools that are available to you via Erlang. While Elixir as a programming language is amazing, some of the Erlang utilities are only available to you via Erlang. It would be too much of a maintenance burden on the Elixir Core Team if they had to maintain wrappers for each Erlang module in Elixir just so that the interface to the Erlang modules feels more "Elixiry".

Luckily, Elixir was designed to work seamlessly with Erlang and so, it is very easy to consume both 1st party and 3rd party Erlang libraries. You'll start off by learning about the various data structures that are natively available to you via the Erlang standard library and then you'll experiment with a handful of other Erlang modules that can provide plenty of utility in your day-to-day Elixir work.

Let's jump right in!

2.2. What Are Immutable Data Structures

With Erlang, and by extension Elixir, being functional programming languages, we can expect that the data structures that we have access to natively on the BEAM are immutable and cannot be altered whenever they are passed to a function. As a construct, immutability provides some amazing guarantees (especially in a team environment). It means that we can rest easy knowing that the data that we pass from function to function is not being changed out from under us and unintended side effects are not something that we generally need to worry about.

What is a side-effect?

A side-effect is when some part of your program performs an operation that has a lasting effect outside the scope of your function. This can be anything from writing to the database, sending an email, altering the filesystem, or changing data outside the scope of the function. This means that calling the same function with the same input data can yield different results since these external dependencies may experience issues and raise non-deterministic errors.

As an example, let's take a look at a Javascript snippet and compare it to a similar Elixir snippet:

Listing 1. Javascript object mutability

```
$ node
> function do_work(data) {
... data.b = 42 ①
... }

> let my_data = { a: 10 } ②

> console.log(my_data)
{ a: 10 } ③

> do_work(my_data)

> console.log(my_data)
{ a: 10, b: 42 } ④
```

① A new `b` key is added to the provided object.

② The initial object only contains an `a` key.

③ Our data prior to calling the `do_work` function.

④ Our data after to calling the `do_work` function.

As you can see from the above snippet, after we call the `do_work` function, the object referenced by the `my_data` variable has been altered. While you get used to this after a while of programming in an OOP environment, every once and a while bugs crop up as a result of data being mutated by downstream functions and they can be very tricky to track down and debug. In a team environment, it requires due diligence by all members of the team to ensure that unintended side-effects do not make their way into the codebase.

Let's take a look at the equivalent Elixir version and see how it differs and why this is important in the context of data structures.

Listing 2. Elixir map immutability

```
$ iex
iex(1) > do_work = fn data ->
...(1) > Map.put(data, :b, 42) ①
...(1) > end
#Function<44.65746770/1 in :erl_eval.expr/5>

iex(2) > my_data = %{a: 10} ②
%{a: 10}

iex(3) > IO.inspect(my_data) ③
%{a: 10}
%{a: 10}

iex(4) > do_work.(my_data)
%{a: 10, b: 42}

iex(5) > IO.inspect(my_data) ④
%{a: 10}
%{a: 10}
```

① A new `b` key is added to the provided map.

② The initial map only contains an `a` key.

③ Our data prior to calling the `do_work` function.

④ Our data after to calling the `do_work` function.

As you can see from marker `4`, the contents of `my_data` remain unchanged. This is the beauty of immutability in action. The state of map bound to `my_data` remains unchanged and so we can confidently pass data from function to function without worrying about side effects. This in turn allows us to create programs composed primarily of pure functions (functions without side effects).

The relevance of this when it comes to data structures in Erlang and Elixir is that all of the data structures that we have access to must be immutable. Like many things in Software Engineering (and engineering in general), this does not come without its trade-offs. In a naive implementation of a run-time that supports immutable data structures, every time a piece of data is acted upon, a full copy of that data would have to be made as not to stomp on any prior versions of that data. As you can imagine, performing a full copy any time something changes would be prohibitively expensive, and any run-time that had immutable data implemented this way would be unusable in a real-world environment due to the performance overhead. Luckily there is a class of data structures that make this possible in an efficient manner and they are called persistent data structures.[1]

Persistent data structures are data structures that can be changed whilst always maintaining a history of all changes that occurred to the data (without a full copy). Under the hood, the BEAM leverages persistent data structures in order to provide immutability as a first-class citizen while not having to copy the entire data structure any time something changes (with the exceptions of when data is passed between processes or when data is extracted from native data stores like ETS). In fact, you can even see this in the Erlang source code when you look at the implementation of the Erlang map.[2] As you can see, maps are actually implemented as hash array mapped trie when the number of keys in the map is greater than 32 entries. This means that

looking for values in a large map (greater than 32 keys) has a complexity of `O(log n)`.

While this may have seemed like a bit of a tangent, the reason that it is important to know this is that supporting immutable data structures greatly impacts how the data is represented internally. In addition, the time complexity[3] for a lot of the Erlang data structures may not align with their mutable counterparts. In other words, a map in a lot of languages may have a time complexity of `O(1)`, while in Erlang a map is `O(N)` for small maps (as they are represented as a list), and `O(log n)` for large maps.

Now that you have a sense of the theory behind the internals of data structures and immutability on the BEAM, it's time to start playing with some of the data structures that we have access to, courtesy of the Erlang standard library. While some data structures (like maps, lists, tuples, etc) are accessible in Elixir without the use of a module, in this section we'll dive into some of the Erlang modules that are purpose-built for certain data structures. The first that we will take a look at is the queue data structure.

2.3. Using Queues in Erlang

Queues are one of the most basic (yet powerful) data structures in computer science and have applicability in a wide range of scenarios. Simply put, a queue defines a collection of items that are processed in a first-in-first-out fashion. In other words, an element `A` that was put into the queue before element `B`, will be processed prior to element `B`. Going forward, when talking about inserting/extracting elements from a queue it is customary to say "push" when elements are inserted into the queue, and "pop" when elements are extracted from the queue.

Where did the name Erlang come from?

Queues are such a fundamental data structure that they even have their own branch of research called Queuing theory[4]. Queueing theory is devoted to the study of queues and was actually created by a man named Agner Krarup Erlang[5]. As a side note, the Erlang programming language is actually named after Agner Krarup Erlang for his work in the telephony space.

While there is no `Queue` data structure in the Elixir standard library, we have all the Erlang data structures at our disposal directly in Elixir and so we can lean on the Erlang `:queue` module. Let's experiment with this module a bit and see how it works:

Listing 3. A simple :queue example

```
iex(1) > my_queue = :queue.new() ①
{[], []}

iex(2) > my_queue = :queue.in(1, my_queue) ②
{[1], []}

iex(3) > my_queue = :queue.in(2, my_queue)
{[2], [1]}

iex(4) > {{:value, my_value}, my_queue} = :queue.out(my_queue) ③
{{:value, 1}, {[3], [2]}}

iex(5) > {{:value, my_value}, my_queue} = :queue.out(my_queue)
{{:value, 2}, {[], [3]}}

iex(6) > {:empty, my_queue} = :queue.out(my_queue) ④
{:empty, {[], []}}
```

① We create a new queue data structure by calling `:queue.new/0`.

② We insert elements into the queue using `:queue.in/2`.

③ We pop elements from the queue using `:queue.out/1` (note the pattern matching to extract the value and the updated queue).

④ When the queue no longer has elements in it, we get an `:empty` atom back as opposed to a `{:value, ⋯}` tuple.

As we can see in the example above, we are able to insert elements into the queue and retrieve them in the desired first-in-first-out (or FIFO for short) order. It is important to note that if we don't reassign `my_queue` every time we alter the queue, we'll get the same result over and over again (courtesy of immutability). As a result, you'll need to ensure that when you are popping elements off the queue, you capture the new state of the queue. This can be a problem if you are iterating or reducing over a queue as you could be infinitely iterating over the queue. Here is an example of this in IEx so you can see this behavior in action:

Listing 4. How not to use a :queue

```
iex(1) > my_queue = :queue.new()
{[], []}

iex(2) > my_queue = :queue.in(1, my_queue)
{[1], []}

iex(3) > :queue.out(my_queue) ①
{{:value, 1}, {[], [2]}}

iex(4) > :queue.out(my_queue) ②
{{:value, 1}, {[], [2]}}
```

① We pop an element from the queue without reassigning the `my_queue` variable.

② Performing another pop on the `my_queue` queue returns the same result.

There are plenty of other functions in the Erlang `:queue` module and I suggest taking a

look at the official Erlang documentation to see what else you can do with queues[6]. One important thing to explicitly call out prior to moving out to digraphs, is that many of the queue operations that require traversing the entire queue are `O(N)` operations. In other words, the entire list must be traversed in order to execute the desired function. Depending on the size of your queue, this may or may not be a problem.

One example where this may be a problem is when you are trying to compute the length of a queue. In order to get the length of the queue, the Erlang `:queue.len/1` function must iterate over the entire queue. As you'll see in the later chapters, it is often useful to wrap an instance of a queue in a struct or a map and manually keep track of its length so that you can avoid the `O(N)` compute time.

2.4. The Many Set Implementations in Erlang

Sets are useful data structures for when you need to ensure that you only have one and only one of each item in the collection. In other words, there is no possibility of having duplicates in a set. If you attempt to insert a duplicate element into the set, you will receive the same set back. Erlang provides us with several different set implementations and it is important to know when each is applicable for the problem at hand. Let's take a look at the `:sets` module first.

2.4.1. sets Module

The Erlang `:sets` module allows you to add elements to the set if comparing them via `===` yields `false`. In addition, depending on how you initialize the set, you may have different underlying data structures that represent the set.

What's the difference between == and ===?

The Elixir Kernel module provides the basic comparison operators that you use in elixir including `==`, `!=`, `!==`, and `===`. The difference between the `==`/`!=` and `===`/`!===` is that the former performs a looser comparison as opposed to the latter which checks the value and the type. This is mostly relevant when comparing floats to integers as `42 == 42.0` will yield `true` while `42 === 42.0` will yield `false`.

Listing 5. Initializing a set

```
iex(1) > :sets.new(version: 2)
%{} ①

iex(2) > :sets.new()
{:set, 0, 16, 16, 8, 80, 48, ②
 {[], [], [], [], [], [], [], [], [], [], [], [], [], [], [], []},
 {{[], [], [], [], [], [], [], [], [], [], [], [], [], [], [], []}}}
```

① When initializing a set with the keyword list `version: 2` the internal representation of the set is actually a map where all the keys have a value of `[]` (this is only available if you are running OTP 24+ and is a more performant set representation).

② When initializing a set with no options, the internal representation of the set is a tuple composed of lists and counters.

Regardless of how the set is instantiated, the functions inside of the `:sets` module allow you to operate on the set just the same. In other words, whether or not you create a new set using `version: 2`, the `:sets` module will behave the same. If you are running OTP 24+, there is no reason not to use `version: 2`. Let's take a look at an example where we use the `:sets` module for something more real-world:

Listing 6. Unique email recipients

```
iex(1) > unique_emails = :sets.new(version: 2)
%{}

iex(2) > unique_emails = :sets.add_element("john@cool-app.com",
unique_emails)
%{"john@cool-app.com" => []} ①

iex(3) > unique_emails = :sets.add_element("jane@cool-app.com",
unique_emails)
%{"jane@cool-app.com" => [], "john@cool-app.com" => []}

iex(4) > unique_emails = :sets.add_element("jane@cool-app.com",
unique_emails)
%{"jane@cool-app.com" => [], "john@cool-app.com" => []} ②

iex(5) > :sets.to_list(unique_emails) ③
["jane@cool-app.com", "john@cool-app.com"]

iex(6) > :sets.is_element("alex@cool-app.com", unique_emails) ④
false
```

① You are able to add elements to the set via `:sets.add_element/2`.

② Adding the same element to the set does not alter the set.

③ You can convert the set into a list via `:sets.add_element/1`.

④ You can check to see if an element is in the set via `:sets.is_element/2`.

There are plenty of other functions in the `:sets` module and I suggest taking a look at the official Erlang docs[7] to see what other operations you can perform.

2.4.2. ordsets Module

The next Erlang module that we will look at is `:ordsets`. This module differs from the previously discussed `:sets` module in a couple of ways:

1. The internal representation of the set is an ordered list where the order is defined by the Erlang term order[8]
2. It checks for element equality via `==` as opposed to `===`

The APIs between the two modules are more or less the same and so the operations that you could perform on sets generated via `:sets` will be more or less the same using `:ordsets`. The only thing to keep in mind is that you cannot pass a set created via `:sets` to `:ordsets` and vice-versa. Let's take a look at an example using `:ordsets`:

Listing 7. Release contributors

```
iex(1) > set = :ordsets.new()
[] ①

iex(2) > set = :ordsets.add_element("Jane Smith", set)
["Jane Smith"]

iex(3) > set = :ordsets.add_element("Alex Koutmos", set)
["Alex Koutmos", "Jane Smith"]

iex(4) > set = :ordsets.add_element("Alex Koutmos", set)
["Alex Koutmos", "Jane Smith"] ②

iex(5) > :ordsets.to_list(set)
["Alex Koutmos", "Jane Smith"] ③
```

① You create a new set using `:ordsets.new/0`.

② Adding the same element to the set does not alter the set.

③ You can convert the set into a list via `:ordsets.to_list/1`.

As you can see, the elements in the set came back in the correct alphabetical order and were indeed unique. This differs from the `:sets` module since the call to `:sets.to_list/1` does not guarantee the order of the resulting set elements. With an understanding of `:sets` and `:ordsets`, let's take a look at one last set module - `:gb_sets`.

2.4.3. gb_sets Module

The `:gb_sets` module is yet another implementation of the set data structure, and like the previously mentioned modules, the user-facing APIs are more or less the same, with the differences being mostly in the underlying implementation of the data structure. Another thing to note is that `:gb_sets` behaves like `:ordsets` in that it compares elements via `==` and not `===`. In other words, if you need to differentiate between `42` and `42.0` in your set, this implementation is not the one for you. Under the hood, `:gb_sets` stores the set as a general balanced tree and so, operations on the set will be done in logarithmic time. Let's take a look at an example using `:gb_sets` to see how it compares to the other implementations:

Listing 8. Lottery numbers

```
iex(1) > set = :gb_sets.new()
{0, nil} ①

iex(2) > set = :gb_sets.add_element(42, set)
{1, {42, nil, nil}}

iex(3) > set = :gb_sets.add_element(2, set)
{2, {42, {2, nil, nil}, nil}}

iex(4) > set = :gb_sets.add_element(10, set)
{3, {42, {2, nil, {10, nil, nil}}, nil}}

iex(5) > set = :gb_sets.add_element(10, set)
{3, {42, {2, nil, {10, nil, nil}}, nil}} ②

iex(6) > :gb_sets.to_list(set) ③
[2, 10, 42]
```

① You create a new set using `:gb_sets.new/0`.

② Adding the same element to the set does not alter the set.

③ You can convert the set into a list via `:gb_sets.to_list/1`.

2.4.4. Which one should I use?

As you can see there are quite a few implementations of sets available to you on the BEAM, not to mention the Elixir `MapSet` version which we haven't even talked about yet (that will be in the next chapter). So the question becomes, which one should you use? The answer will largely depend on what kind of characteristics you are looking for. If you need to have a set that is ordered when it is converted to a list, I would suggest either `:ordsets` or `:gb_sets` (using the latter when the set is greater than 100 elements). If you the ordering of the elements is not important, the `:sets` module will

work just fine and will be the most performant so long as equality checks via `==` are sufficient.

With that being said, let's take a look at the array data structure that is available to us via the `:array` module.

2.5. Arrays…in Erlang?

While you wouldn't expect arrays to be available in a functional programming language, they are indeed something that you get out of the box with Erlang. While arrays in Erlang don't have the same characteristics that you would expect from arrays in object-oriented or mutable languages, they do offer the ability to randomly access elements (albeit not in constant time). For those unfamiliar with arrays, they allow you to get and set elements by a numerical index. They differ from linked lists in that you can access element 5 for example without having to start at the head of the list and traverse it until you get to the fifth element. Let's take a look at an example to see how it works:

Listing 9. Racing results

```
iex(1) > array = :array.new() ①
{:array, 0, 10, :undefined, 10}

iex(2) > array = :array.set(0, "Alex Koutmos", array) ②
{:array, 1, 10, :undefined,
 {"Alex Koutmos", :undefined, :undefined, :undefined, :undefined,
:undefined,
  :undefined, :undefined, :undefined, :undefined}}

iex(3) > array = :array.set(1, "Bob Smith", array)
{:array, 2, 10, :undefined,
 {"Alex Koutmos", "Bob Smith", :undefined, :undefined, :undefined,
:undefined,
  :undefined, :undefined, :undefined, :undefined}}

iex(4) > array = :array.set(2, "Jannet Angelo", array)
{:array, 3, 10, :undefined,
 {"Alex Koutmos", "Bob Smith", "Jannet Angelo", :undefined, :undefined,
  :undefined, :undefined, :undefined, :undefined, :undefined}}

iex(5) > :array.get(1, array) ③
"Bob Smith"

iex(6) > :array.get(2, array)
"Jannet Angelo"
```

① You create a new array by calling `:array.new()`.

② You set a value in the array by calling `:array.set/3` where the arguments are the index, the value and lastly the array that you are operating on (remember that arrays are immutable just like everything else so you'll need to reassign the result).

③ You can retrieve the element in the array at a particular index by calling `:array.get/2`.

You also have the option of converting an array to a list either by calling `:array.sparse_to_list/1` or `:array.to_list/1`. The difference between the two being that `:array.sparse_to_list/1` will skip over any undefined indices while `:array.to_list/1` will not.

If you find yourself needing to extract data in a way that makes heavy use of the element's index, arrays may be a good option for you as tuples (the underlying data structure representing the array) are more performant when it comes to random element access when compared to say lists.

Now that we have had a look at the various immutable data structures that are available to us via the Erlang standard library, it's time to take a look at the Erlang module to see what utilities are available to us as well.

2.6. Using the Erlang Module for Everyday Tasks

In addition to mutable and immutable data structures, the Erlang standard library offers a plethora of other modules for performing a wide variety of tasks. One of these modules that contains a nice collection of utility functions is the `:erlang` module[9]. We picked out a few functions that we have used in production that have proven to be very useful. We'll go over some simple usage examples and provide scenarios when these might be useful in production. Without further ado, let's jump right in!

2.6.1. binary_to_term and term_to_binary

Oftentimes, you'll want to serialize and deserialize Erlang and Elixir terms to a format that you can easily persist and read in at a later point in time. The `:erlang.binary_to_term/{1,2}` and `:erlang.term_to_binary/{1,2}` functions allow you to do just that.

What is a term?

A term in Elixir and Erlang is any piece of data like a number, map, list, tuple, etc. The `any()` typespec in Elixir is a way to denote that the argument that is being provided is an arbitrary term.

You can use these functions for some of the following use cases:

- When you need to perhaps continue work across application restarts and need to save state.
- When you need to send (trusted) data to other Erlang or Elixir applications.
- When you need to persist terms to the database.

Let's see how this works in practice so that it is more clear:

Listing 10. Serializing and deserializing terms

```
iex(1) > my_data = %{name: "John Smith", age: 42, favorite_lang: :elixir}
%{age: 42, favorite_lang: :elixir, name: "John Smith"}

iex(2) > base_64_serialized = my_data |> :erlang.term_to_binary() |> Base
.encode64() ①
"g3QAAAADZAADYWdlYSpkAA1mYXZvcml0ZV9sYW5nZAAGZWxpeGlyZAAEbmFtZW0AAAAKSm9ob
iBTbWl0aA=="

iex(3) > base_64_serialized |> Base.decode64!() |> :erlang.
binary_to_term([:safe]) ②
%{age: 42, favorite_lang: :elixir, name: "John Smith"}
```

① We take our `my_data` map and serialize it using `:erlang.term_to_binary/1` and then base64 encode it using `Base.encode64/1`.

② We can then deserialize the map by first decoding the base64 encoded string and then passing the result to `:erlang.binary_to_term/2`.

One important thing to note about decoding is that you need to make sure that the data you are decoding is coming from a trusted source. It is best not to decode serialized data that is coming from untrusted sources as you run the potential of crashing the BEAM if the payload attempts to create new atoms to overflow the atom table. If you do need to decode data that cannot necessarily be trusted, be sure to pass the `:safe` option as you did in the previous example.

2.6.2. md5

MD5, while not super useful in the context of cryptography, can be useful for a number of other things given how quickly the hashes can be created. With that being said, the `:erlang.md5/1` function can be useful if you need to keep track of files and to see if they have changed.

Listing 11. Computing MD5 hash

```
iex(1) > "./elixir_patterns.pdf" |>
...(1) > File.read!() |>
...(1) > :erlang.md5()
<<168, 142, 134, 106, 203, 208, 151, 185, 200, 125, 31, 103, 26, 184, 157,
110>>
```

As you can see, it is straightforward to compute the MD5 hash of any binary or iolist data. If you need to transfer this data or display it and cannot send the raw binary as it is returned from the `:erlang.md5/1` function, you can always use the Elixir `Base` module and encode it.

2.6.3. phash2

If you have ever needed to partition data or spread work across a group of processes/nodes, `:erlang.phash2/2` is a very handy utility to have in your back pocket. Given any term, and an upper range, `:erlang.phash/2` will deterministically produce an integer within that range (`0..(upper_range - 1)` to be exact). In other words, every

time you provide the term X to the function, you will get back the integer Y. This can be useful when partitioning data or work because then you can map any term back to the worker or partition that it belongs to. The Elixir Registry module for example makes use of this function when partitioning the ETS tables to look up registered processes[10].

Let's take a look at a short example so you can see how `:erlang.phash2/2` distributes the hashes for a simple term:

Listing 12. Partitioning data with phash2

```
iex(1) > 1..100_000 |>
...(1) > Enum.reduce(%{}, fn number, acc ->
...(1) >   index = :erlang.phash2("Some data - #{number}", 10) ①
...(1) >   Map.update(acc, index, 1, &(&1 + 1)) ②
...(1) > end)
%{
  0 => 9961,
  1 => 10092,
  2 => 10026,
  3 => 9929,
  4 => 10000,
  5 => 10112,
  6 => 10121,
  7 => 9844,
  8 => 10078,
  9 => 9837
}
```

① We use `:erlang.phash2/2` to generate a deterministic index between 0-9 (inclusive) for the provided term.

② We increment the key in the accumulator map so that we can see how `:erlang.phash2/2` distributes the hashed terms.

As you can see, the `:erlang.phash2/2` function does a good job of distributing the hash integer across the range provided even for minutely different terms. While the distribution is not 100% balanced, for all real-world scenarios this will be more than sufficient.

2.6.4. memory

In addition to providing a plethora of tools and utilities for you to use in your applications, Erlang also provides plenty of tools for introspecting the BEAM itself. One of those functions is the `:erlang.memory/0` function and it can give you a good overview as to how you are consuming memory on the BEAM:

Listing 13. Getting memory usage stats

```
iex(1) > :erlang.memory()
[
  total: 49432608,
  processes: 30567016,
  processes_used: 30566016,
  system: 18865592,
  atom: 442553,
  atom_used: 420529,
  binary: 606872,
  code: 8087272,
  ets: 590792
]
```

All the results are reported in bytes and you can see the breakdown between the different resources that the BEAM offers to you. This can be very handy when running a system in production and you need to keep an eye on your application's memory usage. Libraries like PromEx[11] make use of this call in order to capture memory usage over time for your application. Let's take a look at one more introspection function before moving on to the `:crypto` module.

2.6.5. system_info

The `:erlang.system_info/1` function allows you to fetch all sorts of information regarding the BEAM. You can see the current atom count, ETS table count, what are the system limits for the current invocation of the BEAM and so much more. You should definitely explore the Erlang docs to see what else you can do[12]. Here are some sample calls that you can play with:

Listing 14. Getting BEAM information

```
iex(1) > :erlang.system_info(:system_version)
'Erlang/OTP 24 [erts-12.2] [source] [64-bit] [smp:10:10] [ds:10:10:10]
[async-threads:1]\n'

iex(2) > :erlang.system_info(:atom_count) ①
13728

iex(3) > :erlang.system_info(:atom_limit)
1048576

iex(4) > :erlang.system_info(:ets_count)
24

iex(5) > :erlang.system_info(:schedulers)
10

iex(6) > :erlang.system_info(:emu_flavor)
:emu
```

As you can see, by passing different atoms to the `:erlang.system_info/1` function, you can retrieve information on that particular resource. Be sure to check out the Erlang docs for the full listing as there are many possible things that you can view. Another function to take a look at while visiting the Erlang docs is the `:erlang.statistics/1` function. It operates similarly to the `:erlang.system_info/1` function in that you pass

an atom to the function and you retrieve some data back regarding the specified resource.

Now that you have a good sense of the functions available to you in the `:erlang` module, let's continue our journey through the Erlang standard library by taking a look at the `:crypto` module.

2.7. What's Next?

While we covered quite a bit of the Erlang standard library in this chapter, there are still plenty of gems that are useful in your day-to-day programming that we did not cover here. We urge you to dive into the Erlang docs[13] and do some exploring. You may be surprised by what you find!

With a solid overview of the Erlang standard library, it's time to shift our focus to the Elixir standard library. Similar to the Erlang standard library, there is quite a bit in the Elixir standard library. So instead of going over every function in the Enum or Map module, we'll be looking at useful ways to compose functions from these modules in order to perform various tasks. With that being said, let's jump right to it!

[1] https://en.wikipedia.org/wiki/Persistent_data_structure
[2] https://github.com/erlang/otp/blob/maint-24/erts/emulator/beam/erl_map.c#L20
[3] https://en.wikipedia.org/wiki/Time_complexity
[4] https://en.wikipedia.org/wiki/Queueing_theory
[5] https://en.wikipedia.org/wiki/Agner_Krarup_Erlang
[6] https://www.erlang.org/doc/man/queue.html
[7] https://www.erlang.org/doc/man/sets.html
[8] https://www.erlang.org/doc/reference_manual/expressions.html#term-comparisons
[9] https://www.erlang.org/doc/man/erlang.html
[10] https://github.com/elixir-lang/elixir/blob/v1.13.2/lib/elixir/lib/registry.ex#L1325-L1327
[11] https://github.com/akoutmos/prom_ex/blob/1.6.0/lib/prom_ex/plugins/beam.ex#L419-L428
[12] https://www.erlang.org/doc/man/erlang.html#system_info-1
[13] https://www.erlang.org/doc

Chapter 3. Erlang Standard Library Part 2

3.1. Introduction

In the previous chapter, we took a look at the immutable data structures that are available to us in the Erlang standard library as well as some of the utility functions available via the `:erlang` module. This chapter will continue diving into the Erlang standard library and will focus on some of the mutable data structures that are in Erlang as well as some more advanced utility modules.

With that being said, let's take a look at some mutable data structures!

3.1.1. Mutable Erlang Data Structures

With Erlang being a functional programming language, you would not expect to find mutable data structures available to you in the standard library. Although Erlang is a functional programming language, that is very much by necessity as it facilitates many of the characteristics that are fundamental to the virtual machine (like shared-nothing concurrency and message passing). As such, it is also very practical and offers you escape hatches for when you need to do things in a not so FP kind of way. Generally speaking, you would reach for mutable data structures when you need to optimize a particular part of your system or application. With that being said, be careful when reaching for the following tools and be sure to understand the nuances that come along with using them.

3.2. Directed Graphs with the Digraph Module

The first data structure (that is implemented in a mutable fashion in Erlang) which you should be familiar with are graphs. In computer science, graphs are abstract data types that allow you to represent certain types of data where the elements in the

dataset have relationships with other pieces of data in the dataset[14]. Some common examples of real-world problems described as graphs include:

- Workflow engines
- Social networks
- Build systems
- Routing and mapping
- Recommendation engines

In each of these problems, you have a collection of vertices (also called nodes) connected to one another via edges. For example, in routing and mapping applications, cities and towns would be your vertices, and the roads that connect them would be your edges. Once you construct your graph with all of your vertices and edges, you can traverse it and ascertain all kinds of information (like the shortest path from city A to city B in a mapping application). Below is an example graph so you can see what a graph looks like when you visualize it:

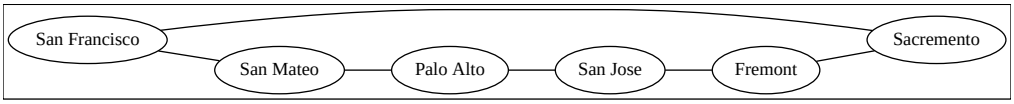


Figure 1. Bay Area cities graph

As you can see, the various cities in the Bay Area in California are the vertices, and the major highways that connect them are the edges. You are only able to travel from one vertex to the other along the defined edges, which dictates how you can traverse the graph.

If constructing and traversing graph data structures sounds complicated or something you would rather not do yourself, don't worry as the Erlang standard library has your back. Erlang provides two modules for constructing and working with graph data structures. Specifically, you have the `digraph` and `digraph_utils` modules for constructing and working with graphs.

One thing to note is that the `digraph` module operates on a class of graphs known as directed graphs. In a directed graph, you still have vertices and edges, with the only caveat that the edges have a specific direction. If you need to denote a bidirectional connection between two vertices, then you need only create two edges - one going in one direction and another going in the opposite direction. In addition, you can have directed graphs that are cyclical and acyclical. Cyclical graphs can have loops in them so that when you traverse them, you can end up on a previously encountered vertex. Whereas with acyclical graphs, you cannot end up on a previously encountered vertex due to the layout of the edges.

In order to get comfortable with the `digraph` module and the accompanying utility module, let's construct a directed graph that represents a series of steps that we need to perform when a new user signs up for our SaaS service. The workflow steps are connected in a directed graph like so:

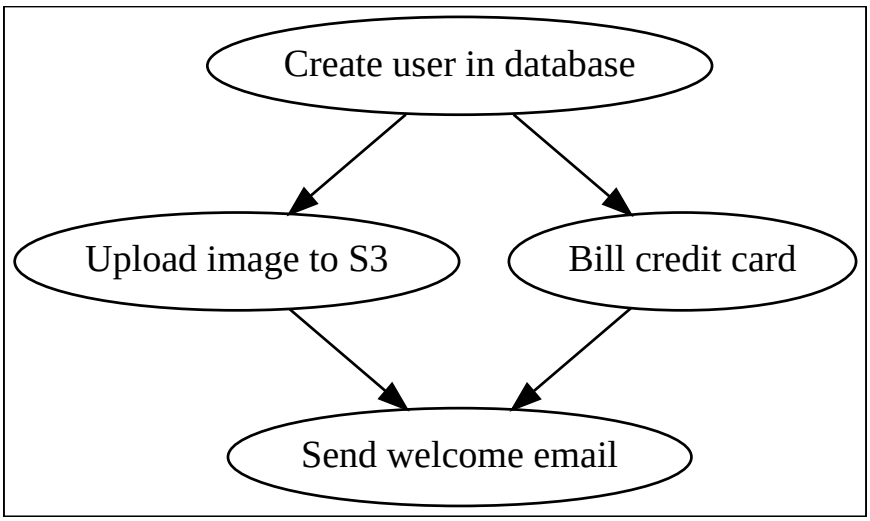


Figure 2. New user registration workflow

Let's start by creating a new directed graph instance and creating an anonymous helper function to simulate work:

Listing 15. Create a new directed graph

```
iex(1) > my_workflow = :digraph.new([:acyclic]) ①
{:digraph, #Reference<0.1888117864.2375680005.579>,
 #Reference<0.1888117864.2375680005.580>,
 #Reference<0.1888117864.2375680005.581>, false}

iex(2) > do_work = fn step -> ②
...(2) >   fn ->
...(2) >     IO.puts("Running the following step: " <> step)
...(2) >
...(2) >     # Simulate load
...(2) >     Process.sleep(500)
...(2) >   end
...(2) > end
#Function<44.65746770/1 in :erl_eval.expr/5>
```

① You create a new directed acyclic graph using `:digraph.new/1` (directed acyclic graphs are actually called DAGs for short as they are a very common type of graph).

② The `do_work/1` anonymous function returns another anonymous function that will be used to simulate work in the workflow.

Next we'll add some steps to the workflow and set up the relationships between the steps:

Listing 16. New user registration workflow setup

```
iex(3) > :digraph.add_vertex(my_workflow, :create_user, do_work.("Create
user in database")) ①
:create_user

iex(4) > :digraph.add_vertex(my_workflow, :upload_avatar, do_work.("Upload
image to S3"))
:upload_avatar

iex(5) > :digraph.add_vertex(my_workflow, :charge_card, do_work.("Bill
credit card"))
:charge_card

iex(6) > :digraph.add_vertex(my_workflow, :welcome_email, do_work.("Send
welcome email"))
:welcome_email

iex(7) > :digraph.add_edge(my_workflow, :create_user, :upload_avatar) ②
[:"$e" | 0]

iex(8) > :digraph.add_edge(my_workflow, :create_user, :charge_card)
[:"$e" | 1]

iex(9) > :digraph.add_edge(my_workflow, :upload_avatar, :welcome_email)
[:"$e" | 2]

iex(10) > :digraph.add_edge(my_workflow, :charge_card, :welcome_email)
[:"$e" | 3]
```

① You add vertices to the graph using `:digraph.add_vertex/3`.

② You add edges to the graph using `:digraph.add_edge/3`.

One thing you will notice with `:digraph` mutable data structure is that we never

reassign the output of any of the function calls to our `workflow` variable. The reason for this is that once we have our reference to the graph (returned to us from the `:digraph.new/1` function), we only need to pass the reference to the subsequent `:digraph` functions and they will perform the necessary side-effects on the instance of the graph. In other words, by passing the reference to the graph to the `:digraph.add_edge/3` function, it will mutate the underlying data structure without returning a new copy of the data structure.

Another important thing to note about the representation of graphs using `:digraph`, is that their state is persisted in ETS. We'll be learning more about ETS later on in the section, but for now, you should understand that this has the following implications:

- The graph can only be updated by the process that created it
- Given that ETS tables not automatically garbage collected, the only way to clean up a `:digraph` instance is to call `:digraph.delete/1`, of for the parent process to terminate

With a better understanding of the `:digraph` caveats, let's see what we can do with the graph that we constructed:

Listing 17. New user registration workflow usage

```
iex(11) > :digraph.info(my_workflow) ①
[cyclicity: :acyclic, memory: 1779, protection: :protected]

iex(12) > :digraph.source_vertices(my_workflow) ②
[:create_user]

iex(13) > :digraph.sink_vertices(my_workflow) ③
[:welcome_email]

iex(14) > :digraph_utils.is_acyclic(my_workflow) ④
true

iex(15) > my_workflow |> ⑤
...(15) > :digraph_utils.topsort() |>
...(15) > Enum.each(fn vertex ->
...(15) >   {_vertex, work_function} = :digraph.vertex(my_workflow,
vertex)
...(15) >   work_function.()
...(15) > end)
Running the following step: Create user in database
Running the following step: Bill credit card
Running the following step: Upload image to S3
Running the following step: Send welcome email
:ok

iex(16) > :digraph.delete(my_workflow) ⑥
true
```

① By calling `:digraph.info/1` we are able to see how much memory our graph is consuming as well as other details.

② You can list the source vertices (where you can enter the graph) using `:digraph.source_vertices/1`.

③ You can list the sink vertices (where you can exit the graph) using `:digraph.sink_vertices/1`.

④ You can make sure that the graph is not cyclical using `:digraph.is_acyclic/1`

⑤ Using the `:digraph_utils.topsort/1` function you can topologically sort the graph. This ensures that parent vertices are visited prior to child vertices such that all the dependencies of a vertex are evaluated prior to that vertex being visited.

⑥ Delete the graph and clean up the ETS tables that were used.

As you can see from the previous example, you can perform some rather complex operations using the `:digraph` module and can model all sorts of problems using graphs. Next, we'll take a look at the performance-oriented modules `:atomics` and `:counters`.

3.3. Fast Incrementers with Atomics and Counters

Similar to the `:digraph` module, the `:atomics` and `:counters` modules are not immutable data structures. In fact, they are not necessarily data structures so much as they are purpose-built data storage mechanisms. Specifically, they are intended for when you have a use case that needs to be **highly** optimized.

Specifically, the `:atomics` and `:counters` modules are purpose-built, hardware-accelerated modules used for adding and subtracting from arrays of numbers in the most performant way possible. As the name implies, the `:atomics` module allows you to perform these operations in an atomic fashion such that there will be no data inconsistencies even if there are a large number of changes happening concurrently. The `:counters` module on the other hand allows you to make the trade-off as to whether you want additional performance at the expense of read inconsistencies.

Let's play around with `:atomics` module first and see how it works in a high throughput environment:

Listing 18. Atomics example

```
iex (1) > existing_user_index = 1
1

iex (2) > new_user_index = 2
2

iex (3) > my_atomic = :atomics.new(2, signed: false) ①
#Reference<0.2046450538.1649803274.4215>

iex (4) > 1..100_000 |>
... (4) > Task.async_stream(
... (4) >   fn _ ->
... (4) >     user_type = Enum.random([:existing_user, :new_user])
... (4) >
... (4) >     case user_type do ②
... (4) >       :existing_user -> :atomics.add(my_atomic,
existing_user_index, 1)
... (4) >       :new_user -> :atomics.add(my_atomic, new_user_index, 1)
... (4) >     end
... (4) >   end,
... (4) >   max_concurrency: 500
... (4) > ) |>
... (4) > Stream.run() ③
:ok

iex (5) > num_existing_users = :atomics.get(my_atomic,
existing_user_index) ④
49760

iex (6) > num_new_users = :atomics.get(my_atomic, new_user_index) ⑤
50240
```

① Using `:atomic.new/2` we create a new atomic with values being unsigned. Index 1

will count new user interactions while index 2 in the atomics array will count existing user interactions in this example. This function call will return a reference to the counter that needs to be used when performing actions on the `:atomic` instance.

② Depending on the random `user_type` that is generated, the appropriate counter is incremented.

③ We run the `Stream` with 500 concurrent tasks in order to work through the 100,000 entries concurrently.

④ Using `:atomics.get/2` we can see how many simulated existing users were counted.

⑤ Using `:atomics.get/2` we can see how many simulated new users were counted.

In the above example, you created a new `:atomics` array of size two. The first position signifies the counts for existing user requests and the second position signifies the counts for new user requests. By leveraging `Task.async_stream/2`, you spawn a maximum of 500 processes in order to simulate concurrent load on the atomic counter. As you can see, in the end, the total count is equal to our expected number of requests.

Let's try the same thing, but with the `:counters` module:

Listing 19. Counters example

```
iex (1) > existing_user_index = 1
1

iex (2) > new_user_index = 2
2

iex (3) > my_counter = :counters.new(2, [:write_concurrency]) ①
{:write_concurrency, #Reference<0.1459955631.581566473.9100>}

iex (4) > 1..100_000 |>
... (4) > Task.async_stream(
... (4) >   fn _ ->
... (4) >     usertype = Enum.random([:existing_user, :new_user])
... (4) >
... (4) >     case user_type do
... (4) >       :existing_user -> :counters.add(my_counter,
existing_user_index, 1)
... (4) >       :new_user -> :counters.add(my_counter, new_user_index, 1)
... (4) >     end
... (4) >   end,
... (4) >   max_concurrency: 500
... (4) > ) |>
... (4) > Stream.run()
:ok

iex (5) > num_existing_users = :counters.get(my_counter,
existing_user_index)
49902 ②

iex (6) > num_new_users = :counters.get(my_counter, new_user_index)
50098
```

① We create a `:counters` instance in much the same way that we create an `:atomics`

instance. We also provide the `:write_concurrency` flag which provides better write performance at the trade-off that reads may be in accurate.

② Much like with the `:atomics` results, we see an almost 50/50 split between the two different types of simulated events that we are measuring.

As you can see, we are able to yield the same results with the `:counters` module as we did with the `:atomics` module. While these modules are not something that you should reach for on a day-to-day basis (given they are not immutable data structures), it is good to know they exist and are there if you need the extra performance. Next, we'll take a look at another use case specific data structure, `:persistent_term`.

3.4. Blazing Fast Data Reads with Persistent Term

Similar to the `:atomics` and `:counters` modules, the `:persistent_term` module is another mutable data storage option that is purpose-built for a particular use case. In this case it is read performance. This is an excellent option for when you have data that rarely changes but that is read often. As a warning, if the data in `:persistent_term` is changed, a global garbage collection must be run on all running processes in order to clean up any references to the persistent term entry. As such, make sure you have a good sense of the read/write characteristics of the data that you are storing in `:persistent_term` so that you are not inadvertently hindering your application's performance.

With that little warning out of the way, let's see how you can use `:persistent_term` to store your data. You can think of `:persistent_term` as a globally namespaced key-value store. In other words, if you know the key, you can access the data regardless of what process you are calling from. That's why it is good to namespace your keys like in the example below:

Listing 20. Simple persistent term example

```
iex(1) > :persistent_term.get() ①
[
  {{:logger_config, :"$primary_config$"}, 7},
  {{:logger_config, ...}, 7},
  {{...}, ...},
  {...},
  ...
]

iex(2) > :persistent_term.put({:my_app, :my_key}, %{some: "Data", that:
"Rarely", ever: "Changes"}) ②
:ok

iex(2) > {:my_app, :my_key} |>
...(2) > :persistent_term.get() |> ③
...(2) > Map.fetch(:some)
{:ok, "Data"}
```

① `:persistent_term` is used by Erlang and Elixir for certain things as soon as the BEAM starts up.

② You can use `:persistent_term.put/2` to insert data into persistent term storage.

③ You can retrieve data out of persistent term using `:persistent_term.get/1`.

As you can see, by using `:persistent_term.get/0`, you can get all of the data currently stored in `:persistent_term`. By using `:persistent_term.put/2` and `:persistent_term.get/1` you can write and read from the global data store any data that you need. In the later chapters, we'll see how we can pair `:persistent_term` with a GenServer in order to hydrate our read-only cache. The last mutable data storage mechanisms that we will be looking at are ETS and DETS. Let's dive right in!

3.5. Using ETS and DETS for General Purpose Data Storage

Erlang Term Storage, or ETS for short, is a highly performant key-value store that can be used for storing large amounts of data. It is able to fulfill these requirements due to the fact that it is a mutable key-value store. By making this trade-off, ETS is able to outperform the built-in map data structure given that maps are not truly key-value data structures. Specifically, maps as implemented in Erlang, are represented by trees under the hood in order to efficiently support immutability. While it may feel "wrong" at first to reach for a mutable data store in a functional programming language, remember that Erlang and Elixir are more about pragmatism than they are about implementing a purely functional programming language. As such, when the need arises and you need a highly performant data store, ETS may be just what you are looking for. In addition, if you ever need to persist the contents of an ETS table, you can leverage DETS and write the data to disk so that you can use it again at a later point in time.

Prior to storing your data in ETS, there are a couple of things that we should cover so that you are aware of the caveats associated with ETS. Firstly, ETS tables require an owning process to be running in order to remain active. As soon as that owning process terminates, the ETS table is reclaimed and the data is no longer accessible. If you need to terminate the owning process but want the table to stick around, you can use `:ets.give_away/3` function to do so. Another thing to keep in mind when using ETS is that the tables are only accessible from the node that they reside on. In other words, if you are using distributed Erlang and your nodes are connected, you cannot read from an ETS table on another node.

Reading data from remote ETS tables.

If your application is set up in such a way that there are ETS tables on remote Erlang nodes that need to be read from, be sure to check out the `:erpc` module.[15] With this module, you can perform remote procedure calls on a remote node and retrieve data out of the remote ETS table. While we won't be covering distributed Erlang applications in this book, be sure to check out the documentation and experiment with the distribution primitives provided to you via Erlang and OTP!

With those caveats out of the way, let's discuss how you can go about storing data in ETS. ETS supports 4 different types of tables. Those tables are `:set`, `:ordered_set`, `:bag`, and `:duplicate_bag`. These table types have the following properties:

- ETS tables of type `:set` only allow unique keys to exist in the table. If an object with a duplicate key is inserted, the previous object is overwritten. If you do not specify a table type, this is the default.
- ETS tables of type `:ordered_set` also only allow for unique keys to exist in the table. The difference between `:set` and `:ordered_set` is that the table is automatically sorted as the contents of the table change (the elements are sorted by Erlang term order).
- ETS tables of type `:bag` allow for multiple objects to be inserted with the same key, but there can only be one instance of each object per key.
- ETS tables of type `:duplicate_bag` are similar to tables of type `:bag` with the difference being that you can have duplicate objects per key in the table.

Let's take a couple of these table types for a test drive and see how they work. We'll start with the default table type `:set`.

3.5.1. :set ETS tables and Querying

In order to get comfortable with ETS, we'll start by creating a simple table with a few entries that we will, later on, retrieve in a few different ways. Let's start by creating the ETS table and inserting our data:

Listing 21. Creating an ETS table and inserting data

```
iex (1) > unique_ets_table = :ets.new(:my_table, [:set]) ①
#Reference<0.3125578499.571080714.81386>

iex (2) > user_1 = {1, %{first_name: "Alex", last_name: "Koutmos",
favorite_lang: :elixir}}
{1, %{favorite_lang: :elixir, first_name: "Alex", last_name: "Koutmos"}}
②

iex (3) > user_2 = {2, %{first_name: "Hugo", last_name: "Barauna",
favorite_lang: :elixir}}
{2, %{favorite_lang: :elixir, first_name: "Hugo", last_name: "Barauna"}}

iex (4) > user_3 = {3, %{first_name: "Joe", last_name: "Smith",
favorite_lang: :go}}
{3, %{favorite_lang: :go, first_name: "Joe", last_name: "Smith"}}

iex (5) > :ets.insert(unique_ets_table, user_1) ③
true

iex (6) > :ets.insert(unique_ets_table, user_2)
true

iex (7) > :ets.insert(unique_ets_table, user_3)
true
```

① Using `:ets.new/2` you can create a new ETS table. The `unique_ets_table` in this case will contain the reference to the ETS table that was just created. Since the `:set`

table type is the default, this line could have also been written as `:ets.new(:my_table, [])` and the same table would have been created. There are other options that can be passed to `:ets.new/2`, but we'll get into those later on.

② When you insert data into the ETS table you need to format it as a tuple where the first value is the key, and the second value is the object you want to store in ETS.

③ Using the `:ets.insert/2` function, you can insert your data into the ETS table

With our data now in the ETS table, let's perform some simple queries and extract some of the data:

Listing 22. Simple ETS queries

```
iex (8) > :ets.lookup(unique_ets_table, 3) ①
[{3, %{favorite_lang: :go, first_name: "Joe", last_name: "Smith"}}]

iex (9) > :ets.lookup(unique_ets_table, 2)
[{2, %{favorite_lang: :elixir, first_name: "Hugo", last_name: "Barauna"}}]

iex (10) > :ets.lookup(unique_ets_table, 100) ②
[]
```

① Using `:ets.lookup/2` you can retrieve objects via their key.

② If the key does not exist in the ETS table, then an empty list is returned.

As you can see, we are able to fetch data out of the ETS table given that we provide a valid ID to `:ets.lookup/2`. When we provide an invalid key we get back an empty list. This is all well and good if we know the key of the data that we are looking for beforehand...but what if we want to perform some more open-ended queries against our ETS table? In order to do that, we'll need to reach for something called Match Specifications.[16]

In short, MatchSpecs can be used to fetch data out of ETS sort of like how SQL can be used to query and fetch data out of a database. The syntax may feel a bit awkward at

first, but it will make sense once we unpack it. Let's try querying the previously inserted dataset for all users whose favorite programming language is Elixir:

Listing 23. Advanced ETS queries

```
iex (11) > unique_ets_table |>
... (11) > :ets.select([
... (11) >   {
... (11) >     {:"$1", %{first_name: :"$2", last_name: :"$3",
favorite_lang: :elixir}}, ①
... (11) >     [], ②
... (11) >     [{{:"$1", :"$2", :"$3"}}] ③
... (11) >   }
... (11) > ])
[
  {1, "Alex", "Koutmos"},
  {2, "Hugo", "Barauna"}
]

iex (12) > unique_ets_table |>
... (12) > :ets.select_count([
... (12) >   {
... (12) >     {:"$1", %{favorite_lang: :"$2"}},
... (12) >     [{:==, :"$2", :elixir}], ④
... (12) >     [true] ⑤
... (12) >   }
... (12) > ])
2

iex (13) > unique_ets_table
... (13) > |> :ets.select([
... (13) >   {
... (13) >     {:"$1", %{first_name: :"$2", last_name: :"$3",
favorite_lang: :elixir}},
... (13) >     [],
```

```
... (13) >     [%{id: :"$1", first_name: :"$2", last_name: :"$3"}] ⑥
... (13) >   }
... (13) > ])
[
  %{first_name: "Alex", id: 1, last_name: "Koutmos"},
  %{first_name: "Hugo", id: 2, last_name: "Barauna"}
]

iex (13) > :ets.delete(unique_ets_table) ⑦
```

① The three-element tuple that is wrapped in a list and passed to the `:ets.select/2` function represents a single MatchSpec. The first element in the match specification tuple defines the match for the data in the ETS table. As you can see, it mirrors the shape of the data stored in ETS as it is a two-element tuple where the first element is the ID of the object, and the second element is the object itself. The `:"$X"` entries denote variable matches that can be used in subsequent clauses of the MatchSpec. This of this entry like the head of a pattern matched function.

② The second element in the MatchSpec tuple defines the guards that should be applied to the query. In this case, there are none so we pass an empty list. Think of this entry as the `when` clause of a function.

③ The last element in the tuple defines what the returned data should look like.

④ In this guard clause, we want all of the ETS entries where the `favorite_lang` of the entry `== :elixir`.

⑤ In order to count objects with a `:select_count/2`, we need the MatchSpec to return just `true`.

⑥ We can also return the selected data in a map by structuring the return data in a map and including the desired matched fields.

⑦ After we are all done with the ETS table, we can manually delete it via the `:ets.delete/1` function call, or we can terminate the process that owns the table.

As you can see, we can perform some rather SQL-like queries against our ETS data

and retrieve back only what we need. While the fields we are filtering and querying on are not indexed like in an SQL database, the matches that are performed against the ETS table are much more performant than if you extract everything from ETS and then use `Enum.filter/2` after the fact. In addition, any time that you read out of ETS, you are doing a complete copy of the extracted data into the calling process' memory heap. As a result, extracting only what you need can definitely boost performance.

If you ever struggle to write your MatchSpec statements, there is a helper function in ETS that can help guide you. Remember how we said that the three clauses of the MatchSpec tuple can be thought of as different parts of a function definition? Well, you'll be happy to hear that there is an ETS helper function that can help you transform function definitions to MatchSpec clauses. The `:ets.fun2ms/1` function can take anonymous function declarations and turn them into MatchSpec clauses. Let's revisit some of the queries that we put together for the previous example, but have `:ets.fun2ms/1` generated the MatchSpec clauses instead:

Listing 24. Creating MatchSpec clauses

```
iex (1) > :ets.fun2ms(fn {id, %{first_name: first_name, last_name:
last_name}} ->
... (1) >   {id, first_name, last_name}
... (1) > end)
[{
  {:"$1", %{first_name: :"$2", last_name: :"$3"}},
  [],
  [{{:"$1", :"$2", :"$3"}}]
}]

iex (2) > :ets.fun2ms(fn {_id, %{favorite_lang: lang}} when lang ==
:elixir ->
... (2) >   true
... (2) > end)
[{
  {:"$1", %{favorite_lang: :"$2"}},
  [{:==, :"$2", :elixir}],
  [true]
}]
```

As you can see, by using the `:ets.fun2ms/1` function, we can turn anonymous functions into MatchSpec clauses! While this function does not implement all of the functionality available to you via MatchSpecs, it can definitely provide a convenient springboard to help you write some of your queries. With that said, let's take a look at some of the other ETS table types and also talk about table permissions!

3.5.2. :bag ETS Tables and Permissions

In the previous section we talked about how to use an ETS table of type `:set` and how to perform some interesting queries against the data that is stored in those tables. In this section, we'll be leaning on the `:bag` table type and will also be configuring the permissions of our ETS table to see how the different modes behave in different

circumstances.

Before we dive into the examples, let's quickly look at the different types of access modes that ETS tables can enforce: - `:public` - With a public ETS table, any process can read or write to the table in addition to the owning process. - `:protected` - With a protected ETS table, any process can read from the table, but only the owning process can write to it. This is actually the default access mode for ETS tables that do not explicitly define an access mode. - `:private` - With a private ETS table, only the owning process can read and write to the table.

With that quick summary of the access modes, let's start by creating a new ETS table that will contain some event data that can be written from any process:

Listing 25. Public ETS example

```
iex (1) > my_metrics_table = :ets.new(:my_metrics_table, [:bag, :public])
①
#Reference<0.1927145192.4284874762.137966>

iex (2) > Task.async(fn -> ②
... (2) >   :ets.insert(my_metrics_table, {:auth_attempt, %{user: "Alex",
ts: NaiveDateTime.utc_now()}})
... (2) >   :ets.insert(my_metrics_table, {:auth_attempt, %{user: "Hugo",
ts: NaiveDateTime.utc_now()}})
... (2) >   :ets.insert(my_metrics_table, {:new_user_created, %{user:
"Jane", ts: NaiveDateTime.utc_now()}})
... (2) > end)
%Task{
  owner: #PID<0.109.0>,
  pid: #PID<0.118.0>,
  ref: #Reference<0.1927145192.4284809226.138129>
}

iex (3) > :ets.tab2list(my_metrics_table) ③
[
  new_user_created: %{ts: ~N[2022-03-07 16:26:06.363282], user: "Jane"},
  auth_attempt: %{ts: ~N[2022-03-07 16:26:06.357023], user: "Alex"},
  auth_attempt: %{ts: ~N[2022-03-07 16:26:06.363268], user: "Hugo"}
]
```

① We create a new ETS table of type `:bag` and with an access mode of `:public`.

② Using `Task.async/1` we are able to write to that table from a completely different process (since the IEx session owns the ETS table, we need to use the `Task` module to simulate a different process writing to the table).

③ By dumping the contents of the ETS table using `:ets.tab2list/1` we can see that our data is in the table and that the `:bag` table type allowed us to have different objects

with the same key in the same table.

Let's take this same exact example, but instead try and write to a private table from a `Task` process:

Listing 26. Private ETS example

```
iex (1) > my_metrics_table = :ets.new(:my_metrics_table, [:bag, :private])
①
#Reference<0.281610941.3755343882.136861>

iex (2) > Task.async(fn -> ②
... (2) >  :ets.insert(my_metrics_table, {:auth_attempt, %{user: "Alex",
ts: NaiveDateTime.utc_now()}})
... (2) >  :ets.insert(my_metrics_table, {:auth_attempt, %{user: "Hugo",
ts: NaiveDateTime.utc_now()}})
... (2) >  :ets.insert(my_metrics_table, {:new_user_created, %{user:
"Jane", ts: NaiveDateTime.utc_now()}})
... (2) > end)
%Task{ ③
  owner: #PID<0.109.0>,
  pid: #PID<0.116.0>,
  ref: #Reference<0.281610941.3755278346.136915>
}

12:11:47.979 [error] Task #PID<0.116.0> started from #PID<0.109.0>
terminating ④
** (ArgumentError) errors were found at the given arguments:

  * 1st argument: the table identifier refers to an ETS table with
insufficient access rights
    ...
```

① We create an ETS table of type `:bag` and with an access mode of `:private`.

② We try to insert elements in the table just as before.

③ The `Task` instance is created.

④ The attempted write to the ETS table raised an error and also crashed the IEx session from where the `Task` was spawned (the reason the IEx session crashed is that a `Task` is automatically linked to their calling process).

As you can see, ETS tables are very versatile and allow you to perform a wide range of operations right from the runtime as opposed to needing external dependencies like Redis, MySQL, or PostgreSQL. We'll be revisiting ETS in the later chapters as we learn how we can pair ETS and GenServers together to create some useful and self-contained utilities. The last thing that we will be going over in this section is how we can persist the contents of our ETS tables across BEAM restarts using DETS. Let's jump right in!

3.5.3. Persisting ETS Tables to Disk with :dets

Given that ETS is strictly a memory-only datastore, it can perhaps be the wrong place to store certain kinds of data if you need a storage mechanism that is not volatile. In scenarios like this, you can reach for DETS! If you take a look at the DETS documentation[17], you'll notice that many of the functions that we used in the previous ETS examples also exist in DETS. This is because the two expose a very similar API as they operate in much the same way, with the major difference being that ETS is memory-based while DETS is disk-based. It is also important to note that since DETS relies on the file system, it is much slower than ETS. Depending on your use case this may be an important thing to keep in mind.

We won't cover access models and table types for DETS as they operate the same way as they do with ETS, so instead, we'll look at backing up an ETS table to DETS and see what that flow looks like:

Listing 27. Backing up ETS to DETS

```
iex (1) > {:ok, my_dets_table} = :dets.open_file('my_dets_table.dets',
[type: :bag])
{:ok, 'my_dets_table.dets'} ①

iex (2) > my_ets_table = :ets.new(:my_metrics_table, [:bag])
#Reference<0.471729446.2426011657.37812> ②

iex (4) > :ets.insert(my_ets_table, {:auth_attempt, %{user: "Alex", ts:
NaiveDateTime.utc_now()}})
true

iex (5) > :ets.insert(my_ets_table, {:auth_attempt, %{user: "Hugo", ts:
NaiveDateTime.utc_now()}})
true

iex (6) > :ets.insert(my_ets_table, {:new_user_created, %{user: "Jane",
ts: NaiveDateTime.utc_now()}})
true

iex (8) > :ets.to_dets(my_ets_table, my_dets_table) ③
'my_dets_table.dets'

iex (9) > :dets.sync(my_dets_table) ④
:ok
```

① We create a new DETS file using `:dets.open_file/2` (if the provided file already exists it opens it instead of creating it).

② We create an ETS table of the same `:bag` type as the DETS table.

③ We dump the contents of the ETS table to the DETS table instance.

④ We run `:dets.sync/1` to flush any pending operations to the DETS file.

After running the above code in IEx, you can then press `control+c` to exit IEx. You can then run `ls` in your current working directory and you'll notice that there is a `my_dets_table.dets` file on the file system. Let's start a new IEx session and see how we can use that file to reload our ETS table state:

Listing 28. Restoring ETS from DETS

```
iex (1) > {:ok, my_dets_table} = :dets.open_file('my_dets_table.dets',
[type: :bag])
{:ok, 'my_dets_table.dets'} ①

iex (2) > my_ets_table = :ets.new(:my_metrics_table, [:bag])
#Reference<0.3687902597.1621753858.95423>

iex (4) > :ets.tab2list(my_ets_table) ②
[]

iex (5) > :dets.to_ets(my_dets_table, my_ets_table)
#Reference<0.3687902597.1621753858.95423> ③

iex (6) > :ets.tab2list(my_ets_table)
[
  new_user_created: %{ts: ~N[2022-03-07 18:42:11.473131], user: "Jane"},
  auth_attempt: %{ts: ~N[2022-03-07 18:42:11.470036], user: "Alex"},
  auth_attempt: %{ts: ~N[2022-03-07 18:42:11.472759], user: "Hugo"}
]
```

① We open the existing DETS file using the same `:dets.open_file/2` call as before.

② Our ETS table currently has no data in it after we create it.

③ By calling `:dets.to_ets/2` with the DETS and ETS instances as arguments, we are able to restore all of the ETS data from the previous IEx session.

As you can see, ETS and DETS play together quite nicely and are able to support a wide range of use cases given their different specializations. Next, we'll be taking a

look at the Erlang `:crypto` module so we can learn how to leverage the various cryptographic tools available to us in the standard library.

3.6. Keeping Things Secret with the Crypto Module

Cryptography in and of itself is a very complex and mathematically involved field. While we won't be getting into the weeds as to how the various cryptography algorithms work, it is important to know what the various tools are in the cryptography toolbox and when to reach for them. In general, cryptography is all about secure communication between two or more parties.

The Erlang `:crypto` module helps us ensure secure communication by providing us with tools for computing hashes from data, for validating message authentication codes (MACs), and for encrypting data both symmetrically and asymmetrically. If you are specifically interested in asymmetric cryptography, be sure to also check out the Erlang `:public_key` module[18].

Before diving into the code, let's unpack what these four groups of tools are and how we can leverage them in our day to day programming endeavors.

1. Hash functions - Simply put, hash functions can deterministically compute an output (i.e the same) of a fixed length, regardless of the size of the input. In other words, given a hash function `X`, and input data `Y` you will always get an output of `Z` (or if you prefer it as an equation `X(Y) = Z`). This also means that you cannot reliably reconstruct the input from the output given that hash functions are one-way functions and reduce an arbitrary input space into a finite output space. The fact that these functions work one way is why hash functions are used to store passwords. In the case of a database leak, the attacker would need to attempt a large number of permutations in order to find some value that would yield the same output as to log in to someone else's account.
2. Message Authentication Codes - Message authentication codes, or MACs for short

allow message senders and recipients to verify that the messages that are shared are both authentic and have not been tampered with. Given a shared secret key between the sender and receiver, the two parties can pass a message payload through a MAC function and generate an authentication code that can be used to verify the message once it is received. One example where this is particularly useful is when your application supports webhook functionality. The best way to ensure that the payload that you received has not been tampered with and is indeed authentic is to have your sending application provide to you their result of the MAC function and you can compare that to what you compute on your side. If the two values match then you know that the inbound message can be safely processed.

3. Symmetric Encryption - Symmetric encryption is probably the category of cryptographic tools that people most associate with cryptography. With symmetric cryptography, a message is encrypted with a secret key, at which point it is no longer discernible what the original message was. In order to derive the original message, the same secret key must be applied to the encrypted message through a function that decrypts the encrypted message. If you are encrypting data at rest in a database, this is generally how it is done. It is encrypted via a symmetric encryption algorithm and then written to the database.
4. Asymmetric Encryption - Asymmetric encryption is slightly more complicated than symmetric encryption but equally important. Whereas symmetric encryption relies on the same key to encrypt/decrypt a piece of data, asymmetric encryption relies on a pair of keys to encrypt and decrypt messages. One of the keys is known as the public key which can be freely distributed to anyone without compromising security. The other key is the private key and this key (as the name implies) should be kept secret and not distributed. Using the public key, message senders can encrypt messages and send them to their intended recipients. Only the person who has the private key can decrypt the message. This way, even if the sender's machine is compromised, all the attacker would be able to get access to is the public key which is not useful in decrypting messages anyways. If you have ever accessed a website over HTTPS, then you are using asymmetric encryption under the hood to ensure that your communication with the server is secure.

With an understanding of what these different types of tools are and how they work, let's taken them for a test drive using the `:crypto` module.

3.6.1. Hash Function

As mentioned earlier, hash functions are one-way functions that produce the same fixed-length hash for the same input. In the example below we pass a few different binaries to the hash `:crypto.hash/2` function and apply different hash algorithms each time:

Listing 29. Computing the hash from data

```
iex(1) > :crypto.hash(:blake2s, "This is some data") ①
...(1) > |> Base.encode16()
"B060EC95F0BEF0F967671AB6A47196685CC241BF3CD329A8DB3A09E253C7CCA9"

iex(2) > :crypto.hash(:blake2s, "This is some other data") ②
...(2) > |> Base.encode16()
"170959DFB421768A00EFDC6C70580C5B473E54691A6E5EF32711D4D102CE8AE8"

iex(3) > :crypto.hash(:sha256, "This is some other data") ③
...(3) > |> Base.encode16()
"07F6BFDB1BC57D898DD8A9022BF01BB581529323071E21337628C3EF6AB29BD1"
```

① Generate the hash of the data using the `:blake2s` algorithm

② Using the same algorithm but a different payload, a different hash is generated

③ In this case we generate the hash of the data using `:sha256`

As you can see, by using the `:crypto.hash/2` function we can generate the hash for a piece of data. In the previous examples, we leveraged the `:blake2s` and `:sha256` algorithms to generate the hash on some strings. Next, let's take a look at some different ways to use MACs.

3.6.2. Message Authentication Codes

MACs allow us to validate that a message came from a known source and that the message has not been tampered with. They are able to do this given that the sender and receiver share a secret key and when the key is applied to the message, an equal hash should be generated. Let's take this for a test drive for a better understanding:

Listing 30. MAC helper functions

```
iex(1) > generate_hmac = fn secret_key, payload -> ①
...(1) > :hmac
...(1) > |> :crypto.mac(:sha256, secret_key, payload)
...(1) > |> Base.encode64()
...(1) > end
#Function<43.65746770/2 in :erl_eval.expr/5>

iex(2) > validate_hmac = fn your_key, payload, expected_hash -> ②
...(2) > :hmac
...(2) > |> :crypto.mac(:sha256, your_key, payload)
...(2) > |> Base.encode64()
...(2) > |> Kernel.==(expected_hash)
...(2) > end
#Function<42.65746770/3 in :erl_eval.expr/5>
```

① The `generate_hmac` helper function will generate a MAC hash using the SHA256 algorithm

② The `validate_hmac` helper function will check to see if the secret key provided yields the expected MAC hash

Let's leverage these helper functions by running some data through them and checking to see what happens when we attempt to validate the data payload:

Listing 31. Validate data integrity with a MAC

```
iex(3) > payload = :erlang.term_to_binary(%{some: "Data", i: "Need"}) |>
Base.encode64()
"g3QAAAACZAABaW0AAAAETmVlZGQABHNvbWVtAAAABERhdGE=" ①

iex(4) > secret_key = "this_is_a_secret_and_secure_key" ②
"this_is_a_secret_and_secure_key"

iex(5) > correct_hmac_hash = generate_hmac.(secret_key, payload)
"gp1QB0rWy4m5DDoDBqZU4hwyVhBdwMXc0gnGELup1Ow="

iex(6) > validate_hmac.("INVALID KEY", payload, correct_hmac_hash) ③
false

iex(7) > validate_hmac.(secret_key, payload, correct_hmac_hash) ④
true
```

① Here we serialize some data using `:erlang.term_to_binary/1`

② We set the correct secret key to `this_is_a_secret_and_secure_key`

③ Given an invalid key, the `validate_hmac/3` function returns false

④ Given a valid key, the `validate_hmac/3` function return true

At first glance, it may seem as though this type of cryptography tooling does not provide a lot of utility when all the data you are working with is local to the machine. Once you consider transmitting data on the public internet, then this becomes far more useful. For example, APIs as a service such as Stripe[19] and Twilio[20] leverage MACs in order to give you the assurance that the data you are operating on is authentic and has not been tampered with. Now that you have a good understanding of how MACs work, let's take a look at symmetric encryption.

3.6.3. Symmetric Encryption

Symmetric encryption is usually what comes to mind when people talk about cryptography. With symmetric encryption, a secret key is required both to encrypt and decrypt messages. Full disk encryption usually relies on symmetric encryption to secure data at rest on the disk and when it comes time to read data off the disk, you must provide the same secret key. Let's see how we can do this using the Erlang `:crypto` module. First we'll create a couple of helper functions in order to make the code more concise:

Listing 32. Helper functions to encrypt and decrypt

```
iex(1) > encrypt =
...(1) >   fn message, key ->
...(1) >     opts = [encrypt: true, padding: :zero] ①
...(1) >     :crypto.crypto_one_time(:aes_256_ecb, key, message, opts)
...(1) >   end
#Function<43.65746770/2 in :erl_eval.expr/5>

iex(2) > decrypt =
...(2) >   fn payload, key ->
...(2) >     opts = [encrypt: false] ②
...(2) >
...(2) >     :aes_256_ecb
...(2) >     |> :crypto.crypto_one_time(key, payload, opts)
...(2) >     |> String.trim(<<0>>) ③
...(2) >   end
#Function<43.65746770/2 in :erl_eval.expr/5>
```

① When we encrypt data, we need to be sure that we pass the `encrypt: true` option. In addition, we need to pad the message using the `padding: :zero` option to ensure that our data does not get trimmed because it is not a block size aligned payload.

② When we decrypt the payload, we need to pass the `encrypt: false` value to signal to the `:crypto.crypto_one_time/4` function that we want to decrypt.

③ Given that we padded the encrypted payload, we also need to trim any trailing null bytes.

With our helper functions in place, we can start to leverage them and see what we get back:

Listing 33. Symmetrically encrypting and decrypting data

```
iex(3) > message = "This is a very very important message. Keep it
secret...keep safe"
"This is a very very important message. Keep it secret...keep safe" ①

iex(4) > secret_key = :crypto.strong_rand_bytes(32) ②
<<97, 176, 26, 172, 231, ...>>

iex(5) > encrypted_message = encrypt.(message, secret_key) ③
<<219, 216, 102, 130, 75, ...>>

iex(6) > try do
...(6) >   decrypt.(encrypted_message, "INVALID_KEY") ④
...(6) > rescue
...(6) >   error -> error
...(6) > end
%ErlangError{original: {:badarg, {'api_ng.c', 244}, 'Bad key size'}}

iex(7) > decrypt.(encrypted_message, :crypto.strong_rand_bytes(32)) ⑤
<<194, 121, 32, 55, 79, 163, ...>>

iex(8) > decrypt.(encrypted_message, secret_key) ⑥
"This is a very very important message. Keep it secret...keep safe"
```

① This is the information that we would like to keep secret

② Using `:crypto.strong_rand_bytes/1` we are able to generate a 32 byte secret key

③ Using our secret key and the original message we encrypt it

④ Providing a key of the wrong length yields an error

⑤ providing a key of the correct size but incorrect value yields random data

⑥ Using the encrypted data and the secret key, we can derive the original message

As you can see, it is relatively straightforward to encrypt and decrypt data using the `:crypto.crypto_one_time/4` function. One thing that should be noted is the value of the last argument to the function. Be sure that you flip the `:encrypt` boolean accordingly and also set the `:padding` option or else your data will be truncated to fit in the cipher's block (if you are using a block cipher like `:aes_256_ecb`). Aside from that, just make sure that your key is the correct size and that you do not lose the key as you won't be able to recover your data.

Asymmetric encryption is a bit more complicated and out of scope for this book. If you are interested in the topic though, I would suggest looking at the Erlang Getting Started guide for public key encryption[21].

3.7. What's Next?

While we covered quite a bit of the Erlang standard library in this chapter, there are still plenty of gems that are useful in your day-to-day programming that we did not cover here. We urge you to dive into the Erlang docs[22] and do some exploring. You may be surprised by what you find!

With a solid overview of the Erlang standard library, it's time to shift our focus to the Elixir standard library. Similar to the Erlang standard library, there is quite a bit in the Elixir standard library. So instead of going over every function in the Enum or Map module, we'll be looking at useful ways to compose functions from these modules in order to perform various tasks. With that being said, let's jump right to it!

[14] https://en.wikipedia.org/wiki/Graph_(abstract_data_type)

[15] https://www.erlang.org/doc/man/erpc.html

[16] https://www.erlang.org/doc/apps/erts/match_spec.html

[17] https://www.erlang.org/doc/man/dets.html

[18] https://www.erlang.org/doc/man/public_key.html
[19] https://stripe.com/docs/webhooks/signatures
[20] https://www.twilio.com/docs/usage/security
[21] https://www.erlang.org/doc/apps/public_key/using_public_key.html
[22] https://www.erlang.org/doc