Elixir Cookbook

This book is a set of recipes grouped by topic that acts as a good reference to get ideas from or to quickly search for a solution to a problem. You will begin by launching an IEx session and using it to test some ideas. Next, you will perform various operations like loading and compiling modules, inspecting your system, generating a supervised app, and so on. Furthermore, you will be introduced to immutability, working with data structures, performing pattern matching, and using stream modules to generate infinite data sequences. You will learn about everything from joining strings to determining the word frequency in text. With respect to modules and functions, you will also discover how to load code from other modules and use guards and pattern matching in functions.

What this book will do for you...

- Utilize Erlang code seamlessly from Elixir, leveraging a huge amount of battle-tested libraries
- Use the Mix tool to generate applications and manage dependencies
- Create modules and functions and use them from different machines, taking advantage of Elixir's integrated distribution mechanism
- Implement OTP behaviors in Elixir
- Package and deploy applications on running systems
- Work with the Phoenix framework to generate a basic web application
- Interact with external programs and APIs
- Serve static files and implement websockets

$ 44.99 US
£ 27.99 UK

Prices do not include local sales tax or VAT

Inside this Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

Free Sample
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Command Line"
- A synopsis of the book’s content
- More information on Elixir Cookbook

About the Author

Paulo A Pereira is a journalist and senior software engineer with a background in Grails and Rails. He fell in love with Elixir and has a passion for exploring new technologies and keeping himself up to date with the industry's developments.

Paulo previously worked as a consultant and lead developer for Mediadigital, implementing Grails and Rails solutions, and he is currently working at Onfido Background Checks, a London-based tech start-up that is proving to be a key player in the background checking industry.

I would like to thank my wife and daughter for their unconditional support. I would also like to thank Wilson for his help and guidance, and José, a true inspiration in his approach to work and life in general.

Finally, I would like to thank all of the reviewers for their valuable comments and the entire Packt Publishing team for their support, especially Fahad, whose kind and steady guidance helped me keep myself on track.
Elixir Cookbook

More than ever, programmers need tools and languages that enable them to develop applications that take full advantage of all the resources available in a system. A few years ago, programs began to speed up just because CPUs were getting progressively faster. However, the "speed limit" has now been hit, and processors are no longer getting faster.

Instead, we are getting more cores available per chip. Today, the challenge is how to take advantage of all that extra power. Elixir helps us do this!

Elixir is a dynamic, functional programming language created by José Valim. It is compatible with the Erlang virtual machine and ecosystem. It focuses on scalability and fault tolerance. With its concurrency model and its ability to handle distribution seamlessly, it makes the task of implementing resilient and efficient systems easier, even fun!

In this cookbook, you will find recipes covering some of the language tooling and concepts. You will find out that no special powers are needed to write concurrent programs or code that can be executed by other machines. You will find out that all you need is an expressive and powerful language, such as Elixir.

What This Book Covers

*Chapter 1, Command Line*, introduces Interactive Elixir (IEx), which is a command line tool that allows us to execute and evaluate code. This chapter also introduces Mix, which is an Elixir tool to create and manage projects.

*Chapter 2, Data Types and Structures*, focuses on some concepts of the language: immutability, pattern matching, and lazy evaluation.

*Chapter 3, Strings and Binaries*, shows us how to manipulate strings in Elixir.

*Chapter 4, Modules and Functions*, focuses on the building blocks of Elixir applications, from module directives to pattern matching in function definitions.

*Chapter 5, Processes and Nodes*, shows you that spawning multiple processes to perform asynchronous computations or connecting multiple machines and executing code on any of them is not as hard as it seems. Elixir makes the task easier, and we explore specific examples.
Chapter 6, **OTP – Open Telecom Platform**, talks about OTP, which is a systematization of common programming concepts. It allows us to develop large-scale systems on a solid foundation. In this chapter, we will explore some of its constructs.

Chapter 7, **Cowboy and Phoenix**, is all about the Web! It discusses a range of topics, from serving static files to implementing websockets, or using a fully-featured web framework.

Chapter 8, **Interactions**, interacts with our host operating system and talks about external systems such as Postgresql or Redis. We will also build a Twitter feed parser.

Appendix, **Installation and Further Reading**, covers references for installing Elixir, Redis, and PostgreSQL, as well as for further reading.
This chapter will cover the following recipes:

- **Interactive Elixir (IEx):**
  - Using the terminal to prototype and test ideas
  - Loading and compiling modules
  - Getting help and accessing documentation within IEx
  - Using Erlang from Elixir
  - Inspecting your system in IEx
  - Inspecting your system with Observer

- **Mix:**
  - Creating a simple application
  - Managing dependencies
  - Generating a supervised application
  - Generating umbrella applications
  - Managing application configuration
  - Creating custom Mix tasks

### Introduction

The command line is the preferred way to create and interact with Elixir applications, inspect running systems, and prototype ideas.
Interactive Elixir (IEx) allows immediate evaluation of any expression, and it is also possible to define modules directly without saving them previously on a file. Similar tools exist in other programming languages; Ruby's IRB or Clojure's REPL are some examples.

Mix is a build tool that provides several tasks to create, compile, and test projects, and handle dependencies. It is also possible to define custom tasks with Mix. In the Creating custom Mix tasks recipe, we will create a task to display the memory usage. It is common for some applications to define their own tasks. Phoenix framework (which will be covered in Chapter 7, Cowboy and Phoenix) is just one example of this.

Using the terminal to prototype and test ideas

The Elixir default installation provides an REPL (short for read-eval-print-loop) named IEx. IEx is a programming environment that takes user expressions, evaluates them, and returns the result to the user. This allows the user to test code and even create entire modules, without having to compile a source file.

To start prototyping or testing some ideas, all we need to do is use IEx via our command line.

Getting ready

To get started, we need to have Elixir installed. Instructions on how to install Elixir can be found at http://elixir-lang.org/install.html. This page covers installation on OSX, Unix and Unix-like systems, and Windows. It also gives some instructions on how to install Erlang, which is the only prerequisite to run Elixir.

How to do it...

To prototype and test the ideas using IEx, follow these steps:

1. Start IEx by typing `iex` in your command line.
2. Type some expressions and have them evaluated:
   ```elixir
   iex(1)> a = 2 + 2
   4
   iex(2)> b = a * a
   16
   iex(3)> a + b
   ```
3. Define an anonymous function to add two numbers:
   ```elixir```
   iex(5)> sum = fn(a, b) -> a + b end
   ```
   Function<12.90072148/2 in :erl_eval.expr/5>
   ```
   4. Invoke the function to add two numbers:
   ```elixir```
   iex(6)> sum.(1,2)
   3
   ```
   5. Quit from IEx by pressing Ctrl + C twice.

**How it works...**

IEx evaluates expressions as they are typed, allowing us to get instant feedback. This allows and encourages experimenting ideas without the overhead of editing a source file and compiling it in order to see any changes made.

In this recipe, we used the `=` operator. Unlike other languages, `=` is not an assignment operator but a pattern matching operator. We will get into more detail on pattern matching in the Using pattern matching and Pattern matching an HTTPoison response recipes in Chapter 2, Data Types and Structures.

In step 3, we used a dot (.) in the `sum` function right before the arguments, like this: `sum.(1,2)`. The dot is used to call the anonymous function.

**There's more...**

It is possible to define modules inside an IEx session.

**Loading and compiling modules**

It is possible to load code from source files into an IEx session. Multiple modules may be loaded and used, allowing us to incorporate existing code into our prototyping or idea testing session.
Getting ready

In this recipe, we will be importing two files that define the Greeter and Echoer modules into our iEx session.

In the following lines, we will list the contents of these modules:

```elixir
code/greeter.ex
defmodule Greeter do
  def greet(name \ "you") do
    "Hello #{name} !"
  end
end

code/echoer.ex

defmodule Echoer do
  def echo(msg) do
    IO.puts "#{msg} ... #{msg} ...... #{msg}"
  end
end
```

Downloading the example code

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

How to do it...

We will follow these steps to load and compile the modules:

1. Start iEx:
   ```elixir```
2. Load the Greeter module defined in greeter.ex:
   ```elixir```

```elixir
iex
iex(1)> c("greeter.ex")
[Greeter]
```
3. Load the `Echoer` module defined in `echoer.ex`:
   ```elixir```
   iex(2)> c("echoer.ex")
   [Echoer]
   ```

4. Use the `greet` function defined in the `Greeter` module:
   ```elixir```
   iex(3)> Greeter.greet("Me")
   "Hello Me !"
   ```

5. Use the `echo` function defined in the `Echoer` module:
   ```elixir```
   iex(4)> Echoer.echo("hello")
   hello ... hello ...... hello
   :ok
   ```

6. Combine the functions defined in both modules:
   ```elixir```
   iex(7)> Greeter.greet("Me") |> Echoer.echo
   Hello Me ! ... Hello Me ! ...... Hello Me !
   :ok
   ```

Some functions may have default values. They are denoted by the use
of `\`. In the `Greeter` module, the `greet` function is defined as
def `greet(name \ "you")`, which means that if we omit the argument
passed to the function, it will default to `you`.

---

**How it works...**

When `c("file_name.ex")` is invoked from IEx, the file is loaded and compiled (a
corresponding file with the `.beam` extension will be created).

The module (or modules) defined on each imported file become available. It is possible to
invoke functions on these modules using the `ModuleName.function_name(args)` syntax.

If a `module_name.beam` file exists for a given module, then every time you import that
module into an IEx session, you will see the following warning:

```elixir```
module_name.ex:1: warning: redefining module ModuleName
```

The warning means that a new compiled `.beam` file is being created, potentially redefining
the module. If no changes were made to the source code, the code will be the same,
although the warning is still issued.

In step 6, the pipe operator (|>) is used to simplify the code. This operator means that the
output of the left operation will be fed as the first argument to the right operation.
This is equivalent to writing the following:

```erl
Echoer.echo(Greeter.greet("Me"))
```

**There's more...**

In steps 2 and 3, the `greeter.ex` and `echoer.ex` files are imported without indicating the path because they are under the same directory from where the IEx session was started.

It is possible to use relative or full paths when loading files:

- We can use relative paths like this:
  ```erl
  iex(1)> c("./greeter.ex")
  ```

- We can use full paths like this:
  ```erl
  iex(2)> c("/home/user/echoer.ex")
  ```

Note that the `c` IEx function accepts a string as an argument.

**Getting help and accessing documentation within IEx**

Documentation is a first-class citizen in the Elixir ecosystem, so it comes as no surprise that IEx provides convenient ways to access documentation and get help without the need to leave an IEx session.

This recipe exemplifies the use of the defined help functions.

**How to do it...**

We will follow these steps to get help and access documentation in an IEx session:

1. Enter `h` inside a running IEx session to see the help options related to the use of IEx helpers, as shown in this screenshot:
Welcome to Interactive Elixir. You are currently seeing the documentation for the module IEx.Helpers which provides many helpers to make Elixir’s shell more joyful to work with.

This message was triggered by invoking the helper h(), usually referred to as h/0 (since it expects 0 arguments).

There are many other helpers available:

- c/2 - compiles a file at the given path
- cd/1 - changes the current directory
- clear/0 - clears the screen
- flush/0 - flushes all messages sent to the shell
- h/0 - prints this help message
- h/1 - prints help for the given module, function or macro
- l/1 - loads the given module’s beam code and purges the current version
- ls/0 - lists the contents of the current directory
- ls/1 - lists the contents of the specified directory
- pwd/0 - prints the current working directory
- r/1 - recompiles and reloads the given module’s source file
- respawn/0 - respawn the current shell
- r/1 - prints spec information
- t/1 - prints type information
- w/0 - prints the history of commands evaluated in the session
- w/1 - retrieves the nth value from the history
- import_file/1 - evaluates the given file in the shell’s context

Help for functions in this module can be consulted directly from the command line, as an example, try:

```elixir
iex> h(c/2)
```

You can also retrieve the documentation for any module or function. Try these:

```elixir
iex> h(Enum)
iex> h(Enum.reverse/1)
```

To learn more about IEx as a whole, just type h(IEx).

iex(3)> _
2. If we wish, for instance, to get information regarding the \( c/2 \) function, we type \( h(c/2) \), as shown in the following screenshot:

3. Accessing a module documentation is done by invoking \( h(ModuleName) \). In the next screenshot, we access documentation related to \( Enum \):
4. Getting information about a specific function inside a module is also possible by invoking `h(ModuleName.function_name)`. The following screenshot shows the documentation for `Enum.map`:

![Screenshot of Enum.map documentation]

**How it works...**

When we define modules, it is possible to use the `@moduledoc` and `@doc` annotations to define documentation related to the whole module or to a specific function in that module.

IEx parses the documentation defined with these annotations and makes it available in a convenient way so that there’s no need to leave the session when help or some more information is needed.

IEx itself has several helper functions defined (refer to the first screenshot of this recipe), and among them, we find `h/0` and `h/1`.

- It is common to refer to functions by their name followed by `/` and a number indicating the number of arguments that function takes. Therefore, `h/0` is a function named `h` that takes 0 arguments, and `h/1` is the same `h` function but with 1 argument.

**There's more...**

There are several defined functions that allow accessing information on function specifications and types (if defined). To learn more, you can use `s/1` and `t/1`.

As an example, to get information on the types defined for the `Enum` module, we would use `t(Enum)`, and to get information on the specifications, we would use `s(Enum)`. 

---

Chapter 1
Using Erlang from Elixir

Elixir code runs in the Erlang VM. The ability to invoke Erlang code from within Elixir allows the use of resources from the entire Erlang ecosystem, and since Elixir code is compiled to the same byte code as Erlang code, there is no performance penalty.

It is also possible to include in an Elixir application the Erlang code that is already compiled.

If you take a closer look, the files we compile in IEx sessions have the .beam extension, and that's exactly the same format Erlang's compiled code gets transformed into.

Getting ready

To use Erlang code in Elixir, we start a new IEx session.

How to do it...

These are some examples of how to invoke Erlang code from Elixir:

1. Erlang's Application module defines a function named `which_applications`. This function returns a list of applications being executed in an Erlang VM. This is the way to use this function from Elixir:

   ```elixir
   iex(1)> :application.which_applications
   ```

   The Erlang code would be `application:which_applications()`.  

2. To get information on any Erlang module, there is a function named `module_info`. To know more about the `erlang` module, we enter this:

   ```elixir
   iex(2)> :erlang.module_info
   ```

   The Erlang code would be `erlang:module_info()`.  

How it works...

In Elixir, Erlang modules are represented as atoms. Functions in these modules are invoked in the same way as any Elixir function.
In Elixir, the atom type is used as an identifier for a given value. In Ruby, the equivalent of the atom is known as the symbol.

There's more...

Existing Erlang libraries can be included in Elixir applications, widening the available options. It is also possible to choose an Erlang implementation of a module over Elixir’s.

The Elixir standard library has a List module defined. The Erlang counterpart is lists.

If we wish to get the last element of a list, we could use both modules:

- We can use Elixir's List module like this:
  ```elixir```
  ```
  List.last([1,2,3])
  ```

- We can use Erlang's lists module in this manner:
  ```elixir```
  ```
  :lists.last([1,2,3])
  ```

The Erlang code for this operation is `lists:last([1,2,3]).`

Inspecting your system in IEx

Sometimes, we need to take a look at what is going on in a running VM. It is useful to see which applications are open and any information about memory usage.

We will use some Erlang modules to inspect a VM instance.

Getting ready

Start a new IEx session.

How to do it...

We will follow these steps to get information on our running system:

1. To get the currently running applications, type this:
   ```elixir```
   ```
   iex(1)> :application.which_applications
   [  
   { :logger, 'logger', '0.15.1' },
   ```
The list that is returned contains three-element tuples. The first element is an atom identifying the application, the second element is the application description, and the third is the application version.

2. We get information on the memory usage by running the following commands:
   ```elixir
   iex(2)> :erlang.memory
   [total: 15474240, processes: 4958016, processes_used: 4957056, system: 10516224,
    atom: 256313, atom_used: 234423, binary: 15352, code: 6071692, ets: 399560]
   ``

3. It is also possible to get memory usage for atoms, ets tables, binaries, and so on:
   ```elixir
   iex(3)> :erlang.memory(:atom)
   256313
   ```

### How it works...

As we saw in the previous recipe, *Using Erlang from Elixir*, it is possible to seamlessly call Erlang code from Elixir. Even though there is no specific Elixir code to perform these inspections, it is easy to get these abilities via Erlang libraries.

### See also

- When a GUI environment is available, there's a tool called Observer that helps to get information on an Erlang VM. Take a look at the next recipe, *Inspecting your system with Observer*.

### Inspecting your system with Observer

The command line isn't the only way to get information on an Erlang VM. There is a GUI tool named Observer that allows access to information in a more convenient way.
If a GUI-enabled system is available, Observer allows us to open multiple windows with information on the whole system's statistics or even an individual process of that running system.

**Getting ready**

Start an IEx session.

**How to do it...**

To use the Observer GUI application, we will follow these steps:

1. Start the Observer application:
   ```elixir
   iex(1)> :observer.start
   :ok
   ```

2. A new window with a tabbed interface will open, and the first information displayed shows CPU information, memory usage, system information, and statistics, as shown in the following screenshot:

![Observer GUI Screenshot](image-url)
3. Select the **Load Charts** tab to see graphical representation of memory usage, IO, and scheduler utilization over time, as shown here:

![Graphical representation of memory usage, IO, and scheduler utilization over time](image)

4. Under the **Applications** tab, by selecting the **kernel** application, it is possible to see a representation of an application process's hierarchy, as shown in this screenshot:

![Representation of an application process's hierarchy](image)
5. Double-click on any of the nodes, for example, `code_server`. A new window will be opened with information for the specific process, as shown in the following screenshot:

![Process Information](image)

**Creating a simple application**

In this recipe, we will be using Mix to create a new application.

**How to do it...**

To create a new Elixir application, follow these steps:

1. In a command-line session, enter `mix help` to see a list of available tasks:
   ```
   > mix help
   ```
Here is what the screen will look like:

```
> mix new simple_app
```

What happens next is shown in the following screenshot:

```
> mix new simple_app
   * creating README.md
   * creating .gitignore
   * creating mix.exs
   * creating config
   * creating config/config.exs
   * creating lib
   * creating lib/simple_app.exs
   * creating test
   * creating test/test_helper.exs
   * creating test/test/simple_app_test.exs

Your mix project was created successfully. You can use mix to compile it, test it, and more:

  cd simple_app
  mix test

Run 'mix help' for more commands.
```
3. Inside the *simple_app* directory, the generated application is ready to be started. Run `iex -S mix` to start the application and verify that everything is working:

```elixir
> iex -S mix
```

Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe] [kernel-poll:false] [dtrace]
Interactive Elixir (0.15.1) - press Ctrl+C to exit (type h() ENTER for help)
```
```
iex(1)> 
```
```
4. Nothing happened. So is it working? The absence of messages in the IEx session is a good thing. This generated application behaves more like a library; there's no main function like in Java or C. To be sure that the application is responding, edit the `lib/simple_app.ex` file by inserting the following code:

```elixir
defmodule SimpleApp do
  def greet do
    IO.puts "Hello from Simple App!"
  end
end
```

5. Restart the application by pressing `Ctrl + C` twice and entering `iex -S mix` again.

6. In the IEx session, enter `SimpleApp.greet`.

7. You will see the following output from the application:

```
iex(1)> SimpleApp.greet
Hello from Simple App!
:ok
```
```
iex(2)> 
```

The Elixir application is ready to be used either on your local machine or, if a node is started, it could even be used from another machine.

### How it works...

The Elixir installation provides a command-line tool called Mix. Mix is a build tool. With this tool, it is possible to invoke several tasks to create applications, manage their dependencies, run them, and more.

Mix even allows the creation of custom tasks.

### See also

- To generate an OTP application with a supervisor, see the *Generating a supervised application* recipe.
Managing dependencies

One of the advantages of OTP (more information on OTP may be found in Chapter 6, OTP – Open Telecom Platform) is modularity, and it is very common to have several applications running as dependencies of one application. An application is a way to achieve modularity; in this context, we call an application something that is known in other programming languages as a library. In this recipe, we will integrate an HTTP client with a new application. We will be using the Hex package manager (http://hex.pm).

Getting ready

1. Generate a new application with mix new manage_deps:
   ```
   > mix new manage_deps
   ```
   The output is shown in the following screenshot:

   ![Screenshot of mix new manage_deps output]

   Your mix project was created successfully.
   You can use mix to compile it, test it, and more:
   ```
   > cd manage_deps
   > mix test
   Run `mix help` for more commands.
   ```


3. We will choose HTTPoison (https://hex.pm/packages/httpoison).
How to do it...

To add a dependency to our application, we will follow these steps:

1. **Inside the `manage_deps` application, open `mix.exs` and edit the file to include HTTPoison as a dependency:**
   
   ```elixir
   defp deps do
     [{:httpoison, "~> 0.4"}]
   end
   ```

2. **HTTPoison must be started with our system. Add this to the started applications list by including it inside the `application` function:**
   
   ```elixir
def application do
   [applications: [:logger, :httpoison]]
end
   ```

3. **Save `mix.exs` and run `mix deps.get` to fetch the declared dependencies, as shown in this screenshot:**

```bash
> mix deps.get
* Getting httpoison (package)
Checking package (http://s3.hex.pm/tarballs/httpoison-0.4.2.tar)
Using locally cached package
Unpacked package tarball (/Users/paulopereira/.hex/packages/httpoison-0.4.2.tar)
* Getting hackney (package)
Checking package (http://s3.hex.pm/tarballs/hackney-0.13.1.tar)
Using locally cached package
Unpacked package tarball (/Users/paulopereira/.hex/packages/hackney-0.13.1.tar)
* Getting idna (package)
Checking package (http://s3.hex.pm/tarballs/idna-1.8.1.tar)
Using locally cached package
Unpacked package tarball (/Users/paulopereira/.hex/packages/idna-1.8.1.tar)
```
4. Compile the dependencies by executing `mix deps.compile`, as shown in the following screenshot:

```
> mix deps.compile
  == idna (compile)
  Compiled src/punycode.erl
  Compiled src/idna_unicode.erl
  Compiled src/idna.erl
  Compiled src/idna_ucs.erl
  Compiled src/idna_unicode_data.erl
  == hackney (compile)
  Compiled src/hackney_connect/hackney_pool_handler.erl
  Compiled src/hackney_lib/hackney_multiport.erl
  Compiled src/hackney_lib/hackney_url.erl
  Compiled src/hackney_lib/hackney_http.erl
  Compiled src/hackney_lib/hackney_headers.erl
  Compiled src/hackney_lib/hackney_cookies.erl
  Compiled src/hackney_lib/hackney_date.erl
  Compiled src/hackney_connect/hackney_tcp_transport.erl
  Compiled src/hackney_connect/hackney_ssl_transport.erl
  Compiled src/hackney_lib/hackney_urli.erl
  Compiled src/hackney_connect/hackney_socket.erl
  Compiled src/hackney_connect/hackney_http_connect.erl
  Compiled src/hackney_connect/hackney_socket_transport.erl
  Compiled src/hackney_connect/hackney_util.erl
  Compiled src/hackney_connect/hackney_connect.erl
  Compiled src/hackney_connect/hackney_stream.erl
  Compiled src/hackney_client/hackney_response.erl
  Compiled src/hackney_client/hackney_manager.erl
  Compiled src/hackney_client/hackney_idna.erl
  Compiled src/hackney_client/hackney_request.erl
  Compiled src/hackney_app/hackney_http.erl
  Compiled src/hackney_app/hackney_http_adapter.erl
  Compiled src/hackney_app/hackney_http_base.erl
  Compiled src/hackney_app/hackney_http_server.erl
  Compiled src/hackney_lib/hackney_mime.types.erl
  == httpoison
  Compiled lib/httpoison/base.ex
  Compiled lib/httpoison.ex
  Generated httpoison.app
```

Sometimes, some of the dependencies are Erlang projects, so you may get a prompt asking you to install rebar (rebar is a tool similar to Mix used in Erlang projects). Once you accept to download it, it will be available in your system and you won't have to worry about it anymore.

5. Start your application with `iex -S mix`. 
6. Inside the IEx session, check whether HTTPoison is running:
```elixir
iex(1)> :application.which_applications
[{:manage_deps, 'manage_deps', '0.0.1'},
 {:httpoison, 'Yet Another HTTP client for Elixir powered by
 hackney\n', '0.4.2'}, {:hackney, 'simple HTTP client', '0.13.1'}(…)]
```
7. Get Google's main page using HTTPoison:
```elixir
iex(5)> HTTPoison.get("http://www.google.com")
%HTTPoison.Response{body: "<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY><n<h1>302 Moved</h1><
<A HREF="http://www.google.pt/?gfe_rd=cr&amp;ei=WFAOVLvQFJSs8wfehYJg">here</A>.</BODY></HTML></n",
headers: %{"Alternate-Protocol" => "80:quic", "Cache-Control" => "private",
"Content-Length" => "256", "Content-Type" => "text/html;
charset=UTF-8",
"Date" => "Tue, 09 Sep 2014 00:56:56 GMT",
"Location" => "http://www.google.pt/?gfe_rd=cr&
ei=WFAOVLvQFJSs8wfehYJg",
"Server" => "GFE/2.0"}, status_code: 302}
```

**How it works...**

Dependencies are preferably added using hex.pm (https://hex.pm/).

If an application doesn't yet exist in Hex, it is also possible to use a GitHub
repository as a source.

To fetch a dependency from GitHub, instead of declaring the dependency
with the {:httpoison, "~> 0.4"} format, the following format is used:

{httpoison, github: "edgurgel/httpoison "}

The local filesystem may also be configured as a source for dependencies,
as follows:

{:httpotion, path: "path/to/httpotion"}

Once the dependencies are declared inside the mix.exs file, there are Mix tasks to get,
compile, and clean them. The dependencies are then fetched, and if these dependencies
have more dependencies on themselves, Mix is smart enough to fetch them.

When compiling dependencies, Mix is also capable of figuring out whether any dependent
application has its own dependencies and whether they need to be compiled.
Starting IEx with the `--s` Mix loads the Mix environment inside IEx, and the application becomes accessible.

As shown in the *Inspecting your system* recipe, it is possible to get a list of running applications and check whether our dependency (and its own dependencies) are running. In the particular case of HTTPoison, automatic start is ensured by adding the atom representing the application name to the list under `applications`([applications: [:logger, :httpoison]]).

### See also
- The documentation on Hex usage available at [https://hex.pm/docs/usage](https://hex.pm/docs/usage).

### Generating a supervised application

An application may be generated with a supervision tree to monitor processes. The supervision tree must be started and stopped with the application, and to do so, an application module callback must also be implemented. Mix provides a simple way to generate this type of application.

### How to do it...

To generate an application with a supervision tree and an application module callback, we run `mix new supervised_app --sup` in the command line. This is shown in the following screenshot:

```
$ mix new supervised_app --sup
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/supervised_app.ex
* creating test
* creating test/test_helper.exs
* creating test/supervised_app_test.exs

Your mix project was created successfully.
You can use mix to compile it, test it, and more:

cd supervised_app
mix test

Run `mix help` for more commands.
```


Chapter 1

How it works...

When mix new task is invoked with the --sup option, although the generated application appears to be identical to the application created in the Creating a simple application recipe, a few things change, which are as follows:

```elixir
supervised_app/mix.exs
def application do
  [applications: [:logger],
   mod: {SupervisedApp, []}]
end
```

An application module callback is added like this:

```elixir
supervised_app/lib/supervised_app.ex

defmodule SupervisedApp do
  use Application
  def start(_type, _args) do
    import Supervisor.Spec, warn: false
    children = [
      # Define workers and child supervisors to be supervised
      # worker(SupervisedApp.Worker, [arg1, arg2, arg3])
    ]
    opts = [strategy: :one_for_one, name: SupervisedApp.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

The Application module behavior is declared, and a start function must be defined to comply with this behavior. Inside the start function, a list of children (usually worker processes) is declared, and so are the supervision options (opts). The supervisor is then started, passing the list of processes to be supervised and the options.

See also

Generating umbrella applications

The "Erlang way" is to name each self-contained unit of code as an app. Sometimes, an app may be what is referred to as a library in other languages. This is a great way to achieve code reusability and modularity, but sometimes, it is convenient to treat all the apps in a project as a single entity, committing them as a whole to version control, to allow running tests, and so on. Think of an umbrella application as a container used to hold one or more applications and to make them behave as a single application.

This recipe shows how to create umbrella applications with Mix.

How to do it...

1. Generate an umbrella application to contain other applications:
   ```bash
   mix new --umbrella container
   ```
   What happens next is shown in the following screenshot:

   ```bash
   > mix new --umbrella container
   * creating .gitignore
   * creating README.md
   * creating mix.exs
   * creating apps
   * creating config
   * creating config/config.exs
   Your umbrella project was created successfully.
   Inside your project, you will find an apps/ directory where you can create and host many apps:
   ```
   ```bash
   cd container
   cd apps
   mix new my_app
   ```
   Commands like 'mix compile' and 'mix test' when executed in the umbrella project root will automatically run for each application in the apps/ directory.

2. Generate application_one and application_two inside the container/apps directory:
   ```bash
   > cd container/apps
   > mix new application_one
   > mix new application_two
   ```
3. Modify the tests in the applications as follows:
   - Change the test in container/apps/application_one/application_one_test.exs like this:
     test "the truth on application one" do
       IO.puts "Running Application One tests"
       assert 1 + 1 == 2
     end
   - Change the test in container/apps/application_two/application_two_test.exs as shown here:
     test "the truth on application two" do
       IO.puts "Running Application Two tests"
       assert 2 - 1 == 1
     end

4. Run the tests in all applications (inside the container directory):
   > mix test

   The result of these tests is shown here:

   ```
   > mix test
   => application_two
   Running Application Two tests
   .
   Finished in 0.04 seconds (0.04s on load, 0.00s on tests) 1 tests, 0 failures
   Rerun with --seed 41707 to replicate
   => application_one
   Running Application One tests
   .
   Finished in 0.00 seconds 1 tests, 0 failures
   ```

5. Now run the tests individually. Firstly, run them for application_one as follows:
   > cd apps/application_one
   > mix test
The outcome of these tests is shown in the following screenshot:

```
> mix test
Running Application One tests
.
Finished in 0.04 seconds (0.04s on load, 0.00s on tests)
1 tests, 0 failures
```

For `application_two`, run them like this:

```bash
> cd ../application_two
> mix test
```

The result of these tests is shown in this screenshot:

```
> mix test
Running Application Two tests
.
Finished in 0.04 seconds (0.04s on load, 0.00s on tests)
1 tests, 0 failures
```

**How it works...**

By generating this structure of the application with subprojects under the `apps` directory, Elixir makes dependency management, compilation, and testing easier. It is possible to perform these tasks at the umbrella application level, affecting all the subprojects, or at each subproject level, allowing a high level of granularity.

**See also**

- The Elixir Getting Started guide on dependencies and umbrella projects is available at http://elixir-lang.org/getting_started/mix_otp/7.html. It says the following:

  *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.

Managing application configuration

Mix tasks run in a specific environment. The predefined environments are production, development, and test (prod, dev, and test). The default environment is dev. In this recipe, we will configure an application with different values for each environment. Invoking the same function will result in a different output based on the configuration.

How to do it...

To manage an application configuration, we follow these steps:

1. Create a new application:
   
   ```bash
   > mix new config_example
   ```

2. Go to the generated application directory and open config/config.exs.

3. Replace all of the file's content with the following code:

   ```elixir
   use Mix.Config
   
   config :config_example,
    message_one: "This is a shared message!"

   import_config "#{Mix.env}.exs"
   ```

4. Create three more files under the config directory with the following code:
   
   - In config/dev.exs, add the following:
     
     ```elixir
     use Mix.Config
     
     config :config_example,
      message_two: "I'm a development environment message!"
     ```
   
   - In config/prod.exs, add this code:
     
     ```elixir
     use Mix.Config
     
     config :config_example,
      message_two: "I'm a production environment message!"
     ```
   
   - In config/test.exs, add the following:
     
     ```elixir
     use Mix.Config
     ```
5. Define two module attributes in `lib/config_example.ex` to hold the values of `message_one` and `message_two`, as follows:

```elixir
@message_one Application.get_env(:config_example, :message_one)
@message_two Application.get_env(:config_example, :message_two)
```

6. Create a `show_messages` function in `lib/config_example.ex`, like this:

```elixir
def show_messages do
  IO.puts "Message one is: #{@message_one}"
  IO.puts "Message two is: #{@message_two}"
end
```

7. Start the application in the three different environments and see the output of the `show_messages` function:

- For the development environment, start the application as follows:
  ```bash
  > MIX_ENV=dev iex -S mix
  iex(1)> ConfigExample.show_messages
  Message one is: This is a shared message!
  Message two is: I'm a development environment message!
  :ok
  iex(2)>
  ```

- For the production environment, start the application like this:
  ```bash
  > MIX_ENV=prod iex -S mix
  iex(1)> ConfigExample.show_messages
  Message one is: This is a shared message!
  Message two is: I'm a production environment message!
  :ok
  iex(2)>
  ```

- For the test environment, start the application as follows:
  ```bash
  > MIX_ENV=test iex -S mix
  iex(1)> ConfigExample.show_messages
  Message one is: This is a shared message!
  Message two is: I'm a test environment message!
  :ok
  iex(2)>
  ```
How it works...

When we include the last line in `config.exs (import_config "#{Mix.env}.exs")`, the Mix configuration is loaded from the files, in this case with the Mix environment as its name and `.exs` as its extension.

The configuration from the imported files will override any existing configuration (with the same key) in the `config.exs` file. In fact, Configuration values are merged recursively. See the example at https://github.com/alco/mix-config-example.

To access configuration values, we use `Application.get_env(:app, :key)`.

Creating custom Mix tasks

Sometimes, the existing Mix tasks just aren't enough. Fortunately, Mix allows the creation of customized tasks that integrate as if they were shipped with Mix itself. In this recipe, we will create a custom Mix task that will print the Erlang VM memory status.

How to do it...

The steps required to create a custom task are as follows:

1. Create a new file, `meminfo.ex`, that defines the `Meminfo` module inside Mix.
   Tasks:
   ```elixir```
   defmodule Mix.Tasks.Meminfo do
     use Mix.Task
   end
   ```

2. Add the new task description to be displayed when `mix help` is invoked:
   ```elixir```
   @shortdoc "Get Erlang VM memory usage information"
   ```

3. Add the new task module documentation:
   ```elixir```
   @moduledoc ""
   A mix custom task that outputs some information regarding
   the Erlang VM memory usage
   ""
   ```

4. Create a run/1 function:
   ```elixir```
   def run(_) do
     meminfo = :erlang.memory
     IO.puts ""
     Total       #{meminfo[:total]}
   end
   ```
5. Compile the code using the Elixir compiler, $\texttt{elixirc}$:

$\texttt{elixirc}$ \texttt{meminfo.ex}

No message should appear but a file named \texttt{Elixir.Mix.Tasks.Meminfo.beam} is created.

6. Run $\texttt{mix}$ \texttt{help} to see the new task listed and its short description:

$\texttt{> mix}$ \texttt{help}

\texttt{mix} # Run the default task (current: mix run)
\texttt{mix archive} # List all archives
(...)
\texttt{mix meminfo} # Get Erlang VM memory usage information
\texttt{mix new} # Create a new Elixir project
\texttt{mix run} # Run the given file or expression
\texttt{mix test} # Run a project's tests
\texttt{iex -S mix} # Start IEx and run the default task

7. Execute the custom task:

$\texttt{> mix meminfo}$

\begin{verbatim}
Total            17692216
Processes        4778984
Processes (used) 4777656
System           12913232
Atom             339441
Atom (used)      321302
Binary           14152
Code             8136817
ETS              452832
\end{verbatim}
How it works...

Mix tasks are just modules that are declared as `Mix.Tasks.<MODULENAME>` with a run function defined.

In `meminfo.ex`, we use the `Mix.Task` module by declaring `use Mix.Task`. The `use` directive allows us to use a given module in the current context.

The `@shortdoc` attribute allows us to define a short description to display when some help on Mix or the `mix.task` is displayed.

The `run/1` function is the place where all of the task's work is done. In this particular case, we use an Erlang function to return a keyword list with several entries, and print them for the user in a formatted way.
Where to buy this book

You can buy Elixir Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.