Getting Started

try, catch, and rescue

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

Errors

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

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

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

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

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

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

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

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

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

```
iex> try do
...>   raise "oops"
...> rescue
```

```
...>   e in RuntimeError -> e
...> end
%RuntimeError{message: "oops"}
```

The example above rescues the runtime error and returns the exception itself, which is then printed in the `iex` session.

If you don’t have any use for the exception, you don’t have to pass a variable to `rescue`:

```
iex> try do
...>   raise "oops"
...> rescue
...>   RuntimeError -> "Error!"
...> end
"Error!"
```

In practice, Elixir developers rarely use the `try/rescue` construct. For example, many languages would force you to rescue an error when a file cannot be opened successfully. Elixir instead provides a `File.read/1` function which returns a tuple containing information about whether the file was opened successfully:

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

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

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

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

```
iex> File.read!("unknown")
** (File.Error) could not read file "unknown": no such file or directory
    (elixir) lib/file.ex:272: File.read!/1
```

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

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

Fail fast / Let it crash

One saying that is common in the Erlang community, as well as Elixir's, is "fail fast" / "let it crash". The idea behind let it crash is that, in case something *unexpected* happens, it is best to let the exception happen, without rescuing it.

It is important to emphasize the word *unexpected*. For example, imagine you are building a script to process files. Your script receives filenames as inputs. It is expected that users may make mistakes and provide unknown filenames. In this scenario, while you could use `File.read!/1` to read files and let it crash in case of invalid filenames, it probably makes more sense to use `File.read/1` and provide users of your script with a clear and precise feedback of what went wrong.

Other times, you may fully expect a certain file to exist, and in case it does not, it means something terribly wrong has happened elsewhere. In such cases, `File.read!/1` is all you need.

The second approach also works because, as discussed in the Processes chapter, all Elixir code runs inside processes that are isolated and don't share anything by default. Therefore, an unhandled exception in a process will never crash or corrupt the state of another process. This allows us to define supervisor processes, which are meant to observe when a process terminates unexpectedly, and start a new one in its place.

At the end of the day, “fail fast” / “let it crash” is a way of saying that, when something *unexpected* happens, it is best to start from scratch within a new process, freshly started by a supervisor, rather than blindly trying to rescue all possible error cases without the full context of when and how they can happen.

Reraise

While we generally avoid using `try/rescue` in Elixir, one situation where we may want to use such constructs is for observability/monitoring. Imagine you want to log that something went wrong, you could do:

```
try do
  ... some code ...
rescue
  e ->
    Logger.error(Exception.format(:error, e, __STACKTRACE__))
    reraise e, __STACKTRACE__
end
```

In the example above, we rescued the exception, logged it, and then re-raised it. We use the `__STACKTRACE__` construct both when formatting the exception and when re-raising. This ensures we reraise the exception as is, without changing value or its origin.

Generally speaking, we take errors in Elixir literally: they are reserved for unexpected and/or exceptional situations, never for controlling the flow of our code. In case you actually need flow control constructs, *throws* should be used. That’s what we are going to see next.

Throws

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

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

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

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

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

Exits

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

```
iex> spawn_link(fn -> exit(1) end)
** (EXIT from #PID<0.56.0>) shell process exited with reason: 1
```

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

`exit` can also be “caught” using `try/catch`:

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

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

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

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

After

Sometimes it's necessary to ensure that a resource is cleaned up after some action that could potentially raise an error. The `try/after` construct allows you to do that. For example, we can open a file and use an `after` clause to close it – even if something goes wrong:

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

The `after` clause will be executed regardless of whether or not the tried block succeeds. Note, however, that if a linked process exits, this process will exit and the `after` clause will not get run. Thus `after` provides only a soft guarantee. Luckily, files in Elixir are also linked to the current processes and therefore they will always get closed if the current process crashes, independent of the `after` clause. You will find the same to be true for other resources like ETS tables, sockets, ports and more.

Sometimes you may want to wrap the entire body of a function in a `try` construct, often to guarantee some code will be executed afterwards. In such cases, Elixir allows you to omit the `try` line:

```
iex> defmodule RunAfter do
...>   def without_even_trying do
...>     raise "oops"
...>   after
...>     IO.puts "cleaning up!"
...>   end
...> end
iex> RunAfter.without_even_trying
cleaning up!
** (RuntimeError) oops
```

Elixir will automatically wrap the function body in a `try` whenever one of `after`, `rescue` or `catch` is specified.

Else

If an `else` block is present, it will match on the results of the `try` block whenever the `try` block finishes without a throw or an error.

```
iex> x = 2
2
iex> try do
...>   1 / x
...> rescue
...>   ArithmeticError ->
```

```
...>     :infinity
...> else
...>   y when y < 1 and y > -1 ->
...>     :small
...>   _ ->
...>     :large
...> end
:small
```

Exceptions in the `else` block are not caught. If no pattern inside the `else` block matches, an exception will be raised; this exception is not caught by the current `try/catch/rescue/after` block.

Variables scope

Similar to `case`, `cond`, `if` and other constructs in Elixir, variables defined inside `try/catch/rescue/after` blocks do not leak to the outer context. In other words, this code is invalid:

```
iex> try do
...>   raise "fail"
...>   what_happened = :did_not_raise
...> rescue
...>   _ -> what_happened = :rescued
...> end
iex> what_happened
** (CompileError) undefined function: what_happened/0
```

Instead, you should return the value of the `try` expression:

```
iex> what_happened =
...>   try do
...>     raise "fail"
...>     :did_not_raise
...>   rescue
...>     _ -> :rescued
...>   end
iex> what_happened
:rescued
```

Furthermore, variables defined in the do-block of `try` are not available inside `rescue/after/else` either. This is because the `try` block may fail at any moment and therefore the variables may have never been bound in the first place. So this also isn't valid:

```
iex> try do
...>   raise "fail"
...>   another_what_happened = :did_not_raise
...> rescue
...>   _ -> another_what_happened
...> end
** (CompileError) undefined function: another_what_happened/0
```

This finishes our introduction to `try`, `catch`, and `rescue`. You will find they are used less frequently in Elixir than in other languages.