Course Content

udemy.com/elixir-scalable-and-efficient-application-development/learn/v4/t/lecture/6958426

Behaviours

Behaviours are not exclusive to Elixir; in fact, this is how Erlang implements the `Gen*` patterns. Behaviours are similar to interfaces in **object-oriented** (**OO**) languages. They specify a collection of functions that the module will expose. All the `GenServer` modules will have the `handle_call`, `handle_cast`, and `handle_info` functions.

The spelling of behaviour in the core Elixir and Erlang API uses the British English spelling. However, Elixir's parser will allow either spelling.

Similar to interfaces in OO languages, the behaviour definition also helps the developers and users of behaviours know which functions **need** to be implemented for the proper usage of their modules. It's hard to create a useful `GenServer` module without your own `handle_call` and `handle_cast`.

Defining behaviours

Behaviours in Elixir are defined using regular modules and the `defcallback` macro.

A short introduction to typespecs

Elixir is a dynamic language, allowing for the types to be inferred at runtime. This is especially useful during development and exploration. However, to achieve this flexibility, compile-time type checks are left at the door. This means we can write functions like the following lines of code:

```
1. iex(1)> defmodule Foo do
2. ...(1)>   def square(x), do: x * x
3. ...(1)> end
```

And, this can be used as usual:

```
1. iex(2)> Foo.square(4)
2. 16
3. iex(3)> Foo.square(4.4)
4. 19.360000000000003
```

However, the compiler and runtime parser are unable to prevent invalid uses because there is no type check:

```
1. iex(4)> Foo.square(:foo)
2. ** (ArithmeticError) bad argument in arithmetic expression
3.     iex:2: Foo.square/1
4. iex(5)> Foo.square("foo")
5. ** (ArithmeticError) bad argument in arithmetic expression
6.     iex:2: Foo.square/1
```

Typespecs are used as a form of self-documenting attributes to the code. They can help prevent developers misusing certain interfaces. Unfortunately, they do not provide compile-time errors:

```
iex(6)> defmodule Foo do
...(6)>   @spec square(number) :: number
...(6)>   def square(x), do: x * x
...(6)> end
iex(7)> Foo.square(4)
16
iex(8)> Foo.square(4.4)
19.360000000000003
iex(9)> Foo.square(:foo)
** (ArithmeticError) bad argument in arithmetic expression
    iex:2: Foo.square/1
iex(10)> Foo.square("foo")
** (ArithmeticError) bad argument in arithmetic expression
    iex:2: Foo.square/1
```

Fortunately, there are tools such as Dialyzer that can check and validate the code annotated with typespecs. Dialyzer is a static-analysis tool for Erlang, and since Elixir is compiled to BEAM code, the bytecode of the Erlang VM, full support of Dialyzer is given essentially for free.

The easiest way to use Dialyzer with Elixir projects is to use the Dialyxir Mix plugin.

The basic syntax for function typespecs is shown as follows:

```
@spec function_name(types_of_parameters) :: return_type
```

Typespecs are implemented as module attributes; thus, the `@spec` directive is declaring an attribute to the current module. The `function_name` declarative specifies **which** function the typespec is for, and the `function_name` declarative should match the actual function name. The parameter and return types are the main components of typespecs. The `::` symbol denotes the transition from function header and return type.

From the preceding example, we had the following typespec:

```
@spec square(number) :: number
```

Here, `square` is the function name, and `number` is the parameter and return type of the function. Another example of a typespec can be found from Elixir's standard library function, `round/1` :

```
@spec round(number) :: integer
```

This states that the `round` function takes anything that is a number and returns integers, where `number` is defined as either `float` or `integer` .

Typespecs are also used to define custom types. For example, let's presume we didn't have the `number` type from before; how would it be defined knowing what we know now? Short of knowing the full syntax, we are guessing something along the lines of:

```
number :: integer | float
```

This would be on point. The only missing part is the annotation:

```
@type number :: integer | float
```

If you're familiar with BNF notation for defining languages, typespecs will feel similar.

Using – implementing – behaviours

Behaviour definitions provide implementors with compile-time checks for correctness and the syntax follows very similarly from typespecs.

The actual syntax uses the `defcallback` macro from the `Behaviour` module.

We will cover macros and their entirety later in this chapter.

Let's walk through a short example of a configuration file parser, where the configuration file could be in several different formats.

The `ConfigParser` behaviour could be defined with the following module:

```
defmodule ConfigParser do
  use Behaviour
  defcallback parse(String.t) :: any
  defcallback extensions() :: [String.t]
end
```

The behaviour is defined similar to how any other Elixir module is. It even uses the familiar use `some_other_module` directive at the beginning to import the `Behaviour` functions and macros. However, instead of defining actual functions, it defines two functions that will be exported by the implementors of this behaviour.

The first function, `parse/1`, takes `String` as a parameter and returns an `any` type, where `any` is a catch-all for types; the returned type can be any valid Elixir type. The second function, `extensions/0`, returns a list of strings.

Next, a supported configuration file format could be JSON, and thus, we would need a configuration parser that implements the behaviour. Again, this is simply another Elixir module:

```
defmodule JsonConfigParser do
  @behaviour ConfigParser
  def parse(str), do: str
  def extensions(), do: ["json"]
end
```

Similarly, a module is defined, but instead of `use`, it specifies a module attribute `@behaviour`. The module then goes to define implementations for the exported functions of the `ConfigParser` behaviour. Since the actual parsing of JSON data isn't really the important detail, it is being omitted.

If we go ahead and save these modules as `config_parser.ex` and `json_config_parser.ex`, respectively, we can compile them and test them out:

```
$ elixirc config_parser.ex
$ elixirc json_config_parser.ex
```

If no output is given, the modules have compiled successfully and without errors or warnings. To test it out, go ahead and launch IEx and import the latter module:

```
iex(1)> import JsonConfigParser
nil
iex(2)> JsonConfigParser.parse("foobar")
"foobar"
iex(3)> JsonConfigParser.extensions
["json"]
```

Now, let's say we would like to add support for YAML configuration files as well. We can add another module (and update the code that dispatches the parsers):

```
defmodule YamlConfigParser do
  @behaviour ConfigParser
  def prase(str), do: str
  def extensions(), do: ["yaml", "yml"]
end
```

And compile it using the following command:

```
$ elixirc yaml_config_parser.ex
yaml_config_parser.ex:1: warning: undefined behaviour function parse/1 (for
behaviour ConfigParser)
```

Oops! It seems we mistyped the `prase/1` function and Elixir is complaining that we did not define the `parse/1` function. Correcting the issue, we will see that the warning goes away:

```
$ elixirc yaml_config_parser.ex
yaml_config_parser.ex:1: warning: redefining module YamlConfigParser
```

Although it still complains, this warning is safe to ignore. In the previous run, the module still compiled into proper BEAM bytecode, however, Elixir complained about the missing definition for the behaviour.

We can similarly load up the `YamlConfigParser` module in `iex` and see that it works (the same) as expected:

```
iex(1)> import YamlConfigParser
nil
iex(2)> YamlConfigParser.parse("yaml!")
"yaml!"
iex(3)> YamlConfigParser.extensions
["yaml", "yml"]
```

Protocols

Protocols are Elixir's way of adding a splash of polymorphism to OTP. Protocols, in a sense, provides a means for generics in code. A specific function can be declared.

However, the definition of such a function may be very specific to the types or records it is given. Thus, via a protocol, the function can be extended to new types that weren't defined, let alone considered, at the time of the original protocol.

Protocols, in this context, are not related to "network protocols". This is an unfortunate abuse of terms.

Protocols are defined similarly to how modules are defined, except, unlike the behaviours from the previous section, a different directive is used to define protocols.

That is, instead of using `defmodule` to define a protocol, we use `defprotocol`.

Let's define a relatively simple protocol for testing **falsy** values.

We will define falsy to be, well, `false`, `nil`, the empty list, `[]`, `0`, and so on. Another semantic we need to choose `is_falsy?/1` return `true` for falsy things, or `false`. For this, since we are testing if something is falsy it should return `true` it is falsy. Another option would be to return `:yes` or `:no`.

The protocol itself can be defined as follows:

```
1. defprotocol Falsy do
2.   def is_falsy?(data)
3. end
```

For the implementations, another macro, `defimpl`, is used. The implementation will take the protocol being defined and the current type for the implementation.

Here is the implementation for atoms:

```
1. defimpl Falsy, for: Atom do
2.   def is_falsy?(false), do: true
3.   def is_falsy?(nil), do: true
4.   def is_falsy?(_), do: false
5. end
```

As mentioned in the preceding code, `false` and `nil` would be falsy. All other atoms are not.

And another implementation for integers:

```
1. defimpl Falsy, for: Integer do
2.   def is_falsy?(0), do: true
3.   def is_falsy?(_), do: false
4. end
```

Again, as decided in the preceding code, `0` is a falsy value, but all other integers are not.

There are a few more types to define:

```
defimpl Falsy, for: List do
  def is_falsy?([]), do: true
  def is_falsy?(_), do: false
end
defimpl Falsy, for: Map do
  def is_falsy?(map), do: map_size(map) == 0
end
```

For lists, we match against the empty list for falsy values, leaving the rest to be truthy.

Maps, on the other hand, are slightly different because `%{}` can't be used for pattern matching since it would match **every** map given. Thus, if the size of the map is 0, the map is falsy.

Now that we have defined falsy for a few types, we can go ahead and try the different types:

```
iex(1)> import_file "falsy.ex"
{:module, Falsy.Map,
 <<70, 79, 82, 49, 0, 0, 6, 136, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 0, 174,
131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99, 115,
95, 118, 49, 108, 0, 0, 0, 2, 104, 2, ...>>,
 {:__impl__, 1}}
iex(2)> Falsy.is_falsy?(false)
true
iex(3)> Falsy.is_falsy?(nil)
true
iex(4)> Falsy.is_falsy?(:yes)
false
```

As expected for the atoms, `false` and `nil` are falsy (`true` results), but `:yes` is not (the `false` result):

```
iex(5)> Falsy.is_falsy?(0)
true
iex(6)> Falsy.is_falsy?(42)
false
iex(7)> Falsy.is_falsy?([])
true
iex(8)> Falsy.is_falsy?([1, 2, 3, 4])
false
iex(9)> Falsy.is_falsy?(Map.new())
true
iex(10)> Falsy.is_falsy?(%{:a => 1})
false
```

Similarly, integers, lists, and maps are handled correctly.

However, what happens when a float value is passed, or any other type that is not explicitly defined?

```
iex(11)> Falsy.is_falsy?(0.0)
** (Protocol.UndefinedError) protocol Falsy not implemented for 0.0
    iex:1: Falsy.impl_for!/1
    iex:2: Falsy.is_falsy?/1
```

As might be expected, Elixir complains that there is no implementation for the given type. As a protocol developer, the decision can be made to leave **unknowns** unimplemented, to be handled downstream, or to provide a default. There are certainly pros and cons to both approaches, and this is a decision that will have to be made. The do nothing approach is to leave it undefined and force the decision on the consumer of the protocol.

Built-in protocols

There are several protocols built into the core of Elixir. These protocols, in fact, enable a large amount of some of the code we have already written.

Take `Enum.map/2` or `Enum.reduce`, for just two examples. Without the `Enumerable` protocol, these two functions would have a very difficult time being **actually** useful:

```
1. iex(1)> Enum.map [1, 2, 3, 4], fn(x) -> x * x end
2. [1, 4, 9, 16]
```

Or the reducer:

```
1. iex(2)> Enum.reducer(1..10, 0, fn(x, acc) -> x + acc end)
2. 55
```

It is a similar case with the `Strings.Chars` protocol. Implementing this protocol for data types is essentially equivalent to implementing a `to_string` function typical of other languages.

```
1. iex(3)> to_string :hello
2. "hello"
```

Occasionally, the `String.Chars` protocol isn't sufficient to print complex datatypes:

```
1. iex(4)> tuple = {1, 2, 3}
2. {1, 2, 3}
```

```
1. iex(5)> "tuple #{tuple}"
2. ** (Protocol.UndefinedError) protocol String.Chars not implemented for {1, 2,
   3}
3.     (elixir) lib/string/chars.ex:3: String.Chars.impl_for!/1
4.     (elixir) lib/string/chars.ex:17: String.Chars.to_string/1
```

In these cases, there is the `Inspect` protocol. The `Inspect` protocol enables a transformation of any datatype into a textual form. Thus, to print tuple from before, we can add a call to inspect:

```
1. iex(6)> "tuple #{inspect tuple}"
2. "tuple {1, 2, 3}"
```

IEx uses `inspect/1` and the `Inspect` protocol for printing results to the console. Notice, however, the printed output prefixed with `#` is no longer a valid Elixir input—the previous examples could be piped back into Elixir or Elixir's interpreter and would be valid Elixir code. For example, using `inspect` on a function reference yields invalid Elixir code:

```
1. iex(7)> inspect &(&1*&1)
2. "#Function<6.54118792/1 in :erl_eval.expr/5>"
```

There are some more functions of Elixir that allow us to get the code representation of our code, and they are known as **Abstract Syntax Tree** (**AST**), and Elixir has great support for working with them.

Course Content

udemy.com/elixir-scalable-and-efficient-application-development/learn/v4/t/lecture/7102836

Abstract syntax trees are the representation of our language compilers (or interpreters) used when parsing and translating written code into a new form, bytecode (for example, Elixir, Erlang, Python, and Java), machine code (for example, C/C++ and assembler), or another language (for example, Less, CoffeeScript, and so on). This, typically internal representation is where the majority of the language expression is broken down into its components for translation or evaluation.

Fortunately, for us, José Valim, and those before him with Erlang decided that the AST should be available to the programmer as first-class datatypes. That is, we can view, evaluate, and manipulate the AST of our code at compile time. This access is what enables metaprogramming. With access to the AST and the ability to manipulate it to suit our needs, we are able to write code that writes code.

To access the AST of any expression, we can use `quote`. It takes the expression or block provided and returns the AST Elixir representation. Let's examine a few examples of using quote:

```
1. iex(1)> quote do: 2 + 5
2. {:+, [context: Elixir, import: Kernel], [2, 5]}
```

The structure of Elixir's AST is always a tuple of three elements:

- The first element is an atom or another tuple
- The second element of the tuple is the context or available binds within the quoted expression
- The third element is the argument list to the function

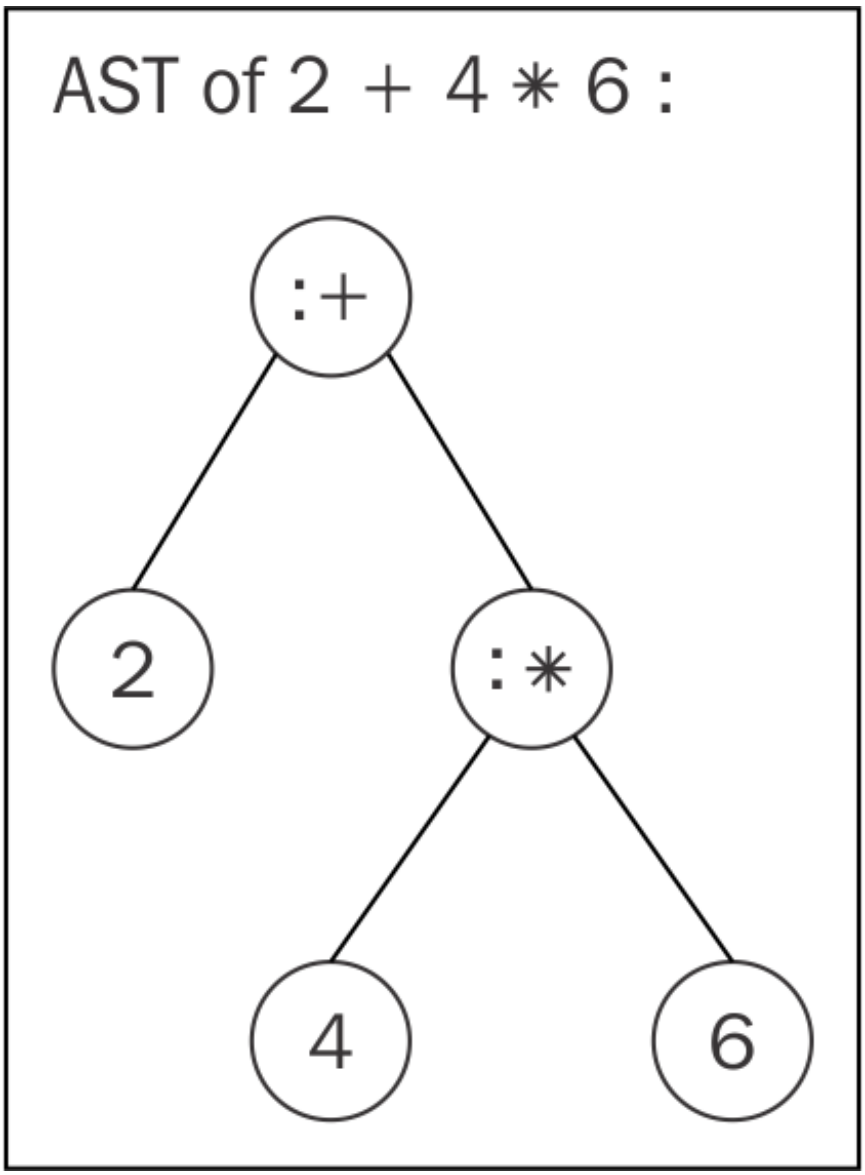
In this example, our first element, `:+`, is the atom representation of the function to be evaluated; the second element of the tuple—the first list—is the imported environmental contexts; and finally, the third element of the tuple—the second list—is the arguments to the function.

Let's try a slightly more complicated example, the one that involves order of operations:

```
1. iex(2)> quote do: 2 + 4 * 6
2. {:+, [context: Elixir, import: Kernel],
3.  [2, {:*, [context: Elixir, import: Kernel], [4, 6]}]}
```

Here, the first element of the larger tuple is `:+` because it will be the last function called in the chain. The second argument to `:+` will be the result of evaluating `:*`.

Let's see the tree as an actual tree:



Notice, that **:+** is the root of the tree with children **2** and **:***. Most ASTs will be represented with prefix notation because it resolves the ambiguous case for operation order that is usually inherent with infix notations.

Let's examine some more complicated ASTs:

```
1. iex(3)> quote do: fn(x) -> x * x end
2. {:fn, [],
3.  [{:->, [],
4.    [[{:x, [], Elixir}],
5.     {:*, [context: Elixir, import: Kernel],
6.      [{:x, [], Elixir}, {:x, [], Elixir}]}]}]}
```

Here, the returned tuples represent the creation of an anonymous function.

Remember, **all** Elixir code can be represented using Elixir data structures.

This, of course, requires that there be a few core data structures to bootstrap the language together. Those types are atoms, numbers, strings, lists (including keyword lists), and 2-pair tuples. Attempting to quote these types will result in themselves being returned:

```
iex(1)> quote do: :atom
:atom
iex(2)> quote do: 42
42
iex(3)> quote do: 19.9991
19.9991
iex(4)> quote do: "Hello, 世界»
«Hello, 世界»
iex(5)> quote do: [1, 2, 3, 4]
[1, 2, 3, 4]
iex(6)> quote do: [a: 1, b: 2]
[a: 1, b: 2]
iex(7)> quote do: {1, 2}
{1, 2}
```

The rest of the language can be realized with the previous data structures and a few special forms. This is where the macro system comes into play.

Our first macros

Now that we have the basics of abstract syntax trees, let's dive into macros, the heart of metaprogramming.

Macros, in the context of Elixir, are a means of deferring the evaluation of certain code. That is, instead immediately expanding an expression, say, when a value is passed, the expression will be passed in its quoted form to the macro. The macro would then be able to decide what it should do with the expressions passed.

This may become more clear when comparing macros to functions. For example, let's attempt to (re)create the `if-else` construct using a function:

```
defmodule MyIf do
  def if(condition, clauses) do
    do_clause = Keyword.get(clauses, :do, nil)
    else_clause = Keyword.get(clauses, :else, nil)
    case condition do
      val when val in [false, nil] ->
        else_clause
      _ -> do_clause
    end
  end
end
```

After loading into `iex`, using `MyIf` may look something similar to the following lines of code:

```
iex(1)> c "myif.exs"
[MyIf]
iex(2)> MyIf.if 1 == 2, do: (IO.puts "1 == 2"), else: (IO.puts "1 != 2")
1 == 2
1 != 2
:ok
```

Since we have defined a function, the arguments passed to the function are evaluated before being passed to the function. Therefore, the value of both `do_clause` and `else_clause` are already set to `:ok`. Similarly, the `condition` value is already bound to `false`. Since `condition` is bound to `false`, the function returns the value for `else_clause`. This results in both `IO.puts/1` calls being executing and the function returning `:ok`. This is exact opposite of what an `if-else` expression should do. Let's fix it using a macro:

```
1.  defmodule MyIf do
2.    defmacro if(condition, clauses) do
3.      do_clause = Keyword.get(clauses, :do, nil)
4.      else_clause = Keyword.get(clauses, :else, nil)
5.      quote do
6.        case unquote(condition) do
7.          val when val in [false, nil] ->
8.            unquote(else_clause)
9.          _ -> unquote(do_clause)
10.       end
11.     end
12.   end
13. end
```

Similar to how the function version is written, the macro extracts the `:do` and `:else` clauses (using `nil` if not found). Next, we return the result of quoting the `case` expression. Inside, the `case` expression, the passed condition is unquoted as to bind the passed value of the condition **into** the quoted block. Similarly, the `:do` and `:else` clauses are unquoted into the expression.

Then, using our macro may look similar to the following code:

```
1.  defmodule Test do
2.    require MyIf
3.    MyIf.if 1 == 2 do
4.      IO.puts "1 == 2"
5.    else
6.      IO.puts "1 != 2"
7.    end
8.  end
```

Save both modules into the same file and load them into an `iex` session:

```
1.  iex(1)> c "myif_macro.exs"
2.  1 != 2
3.  [Test, MyIf]
```

As we expected, only the `else` clause was executed.

The Elixir compiler needs to know **beforehand** the list of available macros as to know how to expand them. This is what the `require` directive at the beginning of the `Test` module does. It informs the Elixir compiler that the `MyIf` module is required to compile this module.

The result of a macro is an AST. The AST is then injected into module that uses it. That is, macros inject code.

Invoking `MyIf.if` inside the `Test` module yields changes in the `Test` module's AST that is then piped into the Elixir compiler.

We can inspect it by making a small change to the `MyIf.if` macro:

```
defmodule MyIf do
  defmacro if(condition, clauses) do
    do_clause = Keyword.get(clauses, :do, nil)
    else_clause = Keyword.get(clauses, :else, nil)
    ast = quote do
      case unquote(condition) do
        val when val in [false, nil] ->
          unquote(else_clause)
        _ -> unquote(do_clause)
      end
    end
    IO.puts Macro.to_string ast
    ast
  end
end
```

Instead of directly returning the result of the `quote` special form, we bind it to the variable, `ast`, and print it using `IO.puts/1` and `Macro.to_string/1`. Finally, the `ast` variable is returned to the caller so that it's properly consumed.

This way, any time the `MyIf.if` macro is used, the resulting AST will be printed to standard out before being injected.

Note that `Macro.to_string/1` does not unquote any of the quoted expressions.

Let's look at another example for Elixir, `square`. The macro will take a single argument and return the square of the argument. The argument will most typically be a number, but could also be an expression of sorts, or be part of a bigger expression.

This macro is often shown in C/C++, as the following `#define` expression:

```
#define square(x)   x*x
```

Don't worry if you don't know C or C++, we are only going to show basic arithmetic expressions of the language.

However, as some simple examples of this usage will show, this is entirely a non-starter for real use. For example, we may try something like this:

```
2/square(10)
```

And, this would expand to `2 / 10 * 10`, which is completely incorrect, and we may be tempted to resolve the issue by surrounding the expression in parentheses:

```
#define square(x) (x * x)
```

But this will still fail in the case of passing an expression to the macro:

```
square(1 + 1)
```

This would be expanded to `(1 + 1 * 1 + 1)`.

Again, we may be tempted to solve this problem by adding more parentheses:

```
#define square(x) ((x) * (x))
```

But this fails again with something like this:

```
x = 1
square(++x)
```

And, this would be expanded to `((++x) * (++x))`.

But what is the result of that expression? It's certainly no longer the square of `x` because `x` is inherently changed within the expression (read nasty variable mutation). Again, there are ways to solve this problem within the language, but more and even more subtle edge cases arise and this rabbit hole path of patching and finding more issues continues.

In Elixir, none of the preceding issues are a concern with even the naive version of the `square` macro:

```
defmodule Math do
  defmacro square(x) do
    quote do
      unquote(x) * unquote(x)
    end
  end
end
```

Loading this module into an `iex` session, we can see that all of the preceding mentioned cases do not result in issues.

```
iex(1)> c "math.exs"
[Math]
iex(2)> import Math
nil
iex(3)> square(10)
100
iex(4)> 2 / square(10)
0.02
iex(5)> square(1 + 1)
4
iex(6)> x = 2
iex(6)> square(x + 1)
9
```

The last example isn't exactly a direct mapping into the C/C++ version since the ++ operator of C languages doesn't exist (because it requires state and variable mutation).

All as we would expect an **actual** `square` function to behave. Now for a more interesting edge case, what if the expression involves sending a message to a process or printing text to the screen when expanded?

```
iex(6)> square((fn() -> IO.puts :square; 16 end).())
square
square
256
```

We still get the correct answer, as we might expect, however, `:square` is printed twice.

Looking back at the `square` macro, we define:

```
defmacro square(x) do
  quote do
    unquote(x) * unquote(x)
  end
end
```

We use `unquote/1` twice. Thus, whatever expression for `x` gets evaluated twice. How can we solve this problem?

We could do it ourselves with something similar to the following code:

```
defmacro square(x) do
  quote do
    x_eval = unquote(x)
    x_eval * x_eval
end
end
```

But this doesn't feel quite right.

We need another facility of Elixir's macro system, `bind_quoted`. Using `bind_quoted` is equivalent to doing the preceding variable assignment inside the `quote` block. It evaluates (read `unqote`) the provided arguments and holds onto their return values for use inside the quote block.

Using `bind_quoted` inside our `Math.square` macro looks something similar to the following code:

```
defmodule Math do
  defmacro square(x) do
    quote bind_quoted: [x: x] do
      x * x
    end
  end
end
```

Reloading this into an `iex` session, we can see this fixes our issue of invoking the `IO.puts` twice:

```
iex(1)> c "math.exs"
[Math]
iex(2)> import Math
nil
iex(3)> square((fn() -> IO.puts :square; 16 end).())
square
256
```

Another tool that we can use for inspecting that the `bind_quoted` option does what we expect is the `Macro.expand_once/2` function. This function takes an AST and the environment context and walks the given AST, expanding and unwrapping the top-level macro operations.

Thus, using this function on our current implementation of `Math.square` gives the following output:

```
iex(4)> Macro.expand_once(quote do square(5) end, __ENV__)
{:__block__, [],
 [{:=, [], [{:x, [counter: 4], Math}, 4]},
  {:*, [context: Math, import: Kernel],
   [{:x, [counter: 4], Math}, {:x, [counter: 4], Math}]}]}
```

The `__ENV__` macro is a module level macro similar to `__MODULE__`, which we've used earlier. Information provided by `__ENV__`, `__MODULE__`, and others is available from `Macro.Env`.

The expanded AST we receive from `Macro.expand_once` gives us a block that first assigns `4` to `x`, then returns the result of `*` with `x` and `x`, and this is exactly the expansion we expected.

Using `Macro.expand_once` on our more complicated variant (with `IO.puts`) results in a similarly structured AST, except with a lot more steps:

```
iex(5)> Macro.expand_once(
...(5)>     quote do square((fn() -> IO.puts :square; 16).()),
...(5)>     __ENV__)
{:__block__, [],
 [{:=, [],
   [{:x, [counter: 8], Math},
    { {:., [],
      [{:fn, [],
        [{:->, [],
          [[],
           {:__block__, [],
            [{ {:., [],
               [{:__aliases__, [alias: false, counter: 8], [:IO]}, :puts]}, [],
              [:square]}, 16]}]}]}]}, [], []}]},
  {:*, [context: Math, import: Kernel],
   [{:x, [counter: 8], Math}, {:x, [counter: 8], Math}]}]}
```

Similar to the earlier example, the expanded AST yielded back is a block that first assigns the result of invoking the anonymous function to `x` that is denoted by the tuple with `:=` and the nested tuple and evaluation of the anonymous function is denoted by the tuple of " `:.` ". Finally, after `x` is assigned, the result would be the evaluation of `:*`, which includes parameters to `x` and `x` again.

The `Macro.expand_once/2` function is indispensable when trying to develop, debug, or simply understand the expansion and injection of code.

However, as you might be wondering, what about the binding of `x`? Does this effect the outer scope? Fortunately, the answer is that it doesn't. Attempting to use the bound `x` variable **outside** of the injected code will result in a compiler error or will return the value bound previously. That is, Elixir macros are hygienic.

Course Content

The injected code of the macro cannot safely assume that certain variables will be available for it to consume. For example, let's look at a macro definition that attempts to access some variables of the caller context:

```
defmodule ContextInfo do
  defmacro grab_caller_context do
    quote do
      IO.puts x
end
  end
end
```

Load up this module in `iex`:

```
iex(1)> c "context.exs"
[ContextInfo]
iex(2)> import ContextInfo
nil
iex(3)> x = 42
42
iex(4)> grab_caller_context
** (CompileError) iex:4: undefined function x/0
    expanding macro: ContextInfo.grab_caller_context/0
    iex:4: (file)
```

After importing and binding a variable, invoking the macro yields a compiler error, as we would expect, because the macro cannot implicitly access the caller's context.

Similarly, the macro cannot safely inject code that changes the context or environment of the caller. Let's add another macro to the `ContextInfo` module that attempts to change the value of `x`:

```
defmacro inject_context_change do
  quote do
    x = 0
    IO.puts x
  end
end
```

Loading it again into `iex`, we see something like the following output:

```
iex(1)> c "context.exs"
[ContectInfo]
iex(2)> import ContextInfo
nil
iex(3)> x = 42
42
iex(4)> inject_context_change
0
:ok
iex(5)> x
42
```

Again, the result of calling the macro does not change the context of the caller.

A more explicit example of macro context versus caller context is shown with the following two modules:

```
defmodule MacroContext do
  defmacro info do
    IO.puts "Macro context: (#{__MODULE__})"
    quote do
      IO.puts "Caller context: (#{__MODULE__})"
      def some_info do
        IO.puts """
        I am #{__MODULE__} and I come with the following:
        #{inspect __info__(:functions)}
        """
      end
    end
  end
end
defmodule MyModule do
  require MacroContext
  MacroContext.info
end
```

Loading the two modules into `iex`, we see the following output:

```
iex(1)> c "context2.exs"
Macro context: (Elixir.MacroContext)
Caller context: (Elixir.MyModule)
[MyModule, MacroContext]
```

Let's invoke the injected function:

```
iex(2)> MyModule.some_info
I am Elixir.MyModule and I come with the following:
[some_info: 0]
:ok
```

The first module defines a macro that before doing anything else, prints the current module name via the `__MODULE__` macro. Inside the `quote` block, we print the caller's module name, again, via the `__MODULE__` macro. Then, we inject a function that prints the module name and the available functions of the module.

The output of compiling the modules yields the output of the macro context (`IO.puts "Macro context ..."`) and the output of the caller context (`IO.puts "Caller context ..."`). Then, invoking the injected `some_info/1` function yields the rest.

How does the `__MODULE__` macro show the correct module name for each context? This, it turns out, is the result of unhygienic macros.

Unhygienic macros – overriding context

Occasionally, part of writing good macros, overriding the context is required analogously to how dealing with state in programming is what results in useful programs.

To accomplish overriding the context, we need yet another macro facility from Elixir — `var!` . The `var!` macro is actually itself very versatile; it allows us to access members of the caller context **without** them being passed to the macro and allows the macro to modify the context of the caller.

Going back to some of our previous examples, let's try adding in `var!` :

```
1. defmodule ContextInfo do
2.   defmacro grab_caller_context do
3.     quote do
4.       IO.puts var!(x)
5.     end
6.   end
7. end
```

After loading and compiling this new version of `ContextInfo` into an `iex` session, we see we can access the `x` variable of the caller context:

```
1. iex(1)> c "context.exs"
2. [ContextInfo]
3. iex(2)> require ContextInfo
4. nil
5. iex(3)> x = 42
6. 42
7. iex(4)> ContextInfo.grab_caller_context
8. 42
9. :ok
```

As expected, the macro was able to print the value of the bound variable, even though the variable was bound in the caller's context and not the macro.

Similarly, updating the `inject_context_change` macro from previous examples:

```
1. defmodule ContextInfo do
2.   defmacro inject_context_change do
3.     quote do
4.       var!(x) = 0
5.     end
6.   end
7. end
```

And loading the updated module into `iex` :

```
iex(1)> c "context.exs"
[ContextInfo]
iex(2)> require ContextInfo
nil
iex(3)> x = 42
42
iex(4)> ContextInfo.inject_context_change
0
iex(5)> x
0
```

The result of this modified macro is the ability to bind a new value for a variable bound in the calling context.

That is, using the `var!` macro allows macros to extract and inject values from or into the context of the caller, enabling unhygienic things to occur.

As a note of caution, this is, by definition, non-functional and introduces complexities into the macros (which are already complicated enough). However, like most things with programming and decisions, the choice to use an unhygienic macro must be weighed properly and not necessarily rejected outright. Is the macro's ability to change the caller's context absolutely necessary for the macro to work? Does it make sense? Is using a macro in this way even warranted to begin with? Consider these questions carefully.

Macro examples

Let's go through some more in-depth examples of using macros to accomplish some pretty cool things, things that are generally fairly difficult to accomplish in other languages.

Debugging and tracing

We'll start with a debugging/tracer module that will enable us to automatically trace the methods of our library.

Of course, this is highly unnecessary since the Erlang VM itself is capable of adding this functionality for us without requiring anything from the developer.

As part of this example, we're going to dive into `use` and `__using__`. `use` as it turns out, is a relatively simple function that invokes the `__using__` macro of the module passed. This, in turn, injects the code of the `__using__` macro.

For example, let's say we've defined a basic module, `UsingTest`, as the following code:

```
defmodule UsingTest do
  defmacro __using__(_opts) do
    quote do
      IO.puts "I'm the __using__/1 of #{unquote(__MODULE__)}"
    end
  end
end
```

If we then define another, very simple module, say, `MyUsingTest`:

```
1. defmodule MyUsingTest do
2.   use UsingTest
3. end
```

Compiling them together should result in some output like the following printed to the screen:

```
1. iex(1)> c "usingtest.exs"
2. I'm the __using__ of Elixir.UsingTest
3. [TestMyUsing, UsingTest]
```

Since our goal is to add some tracing information to the invocation of the functions in our libraries, we need to redefine the `def` macro. That is, we will have a module, say, `Tracer`, with the following code:

```
1. defmodule Tracer do
2.   defmacro def(definition={name, _, args}, do: content) do
3.     quote do
4.       Kernel.def(unquote(definition)) do
5.         unquote(content)
6.       end
7.     end
8.   end
9.   defmacro __using__(_) do
10.    quote do
11.      import Kernel, except: [def: 2]
12.      import unquote(__MODULE__), only: [def: 2]
13.    end
14.  end
15. end
```

The first macro `def/2` defines our new version of `def` that will be used by the modules using use `Tracer`.

However, all we have done is added a level of indirection to the definition of functions, we haven't done any tracing. Let's add some calls that perform these actions.

First, we are going to want a helper function for printing the arguments of the function. It will be a simple transform of the arguments to something printable:

```
1. def dump_args(args) do
2.   args |> Enum.map(&inspect/1) |> Enum.join(", ")
3. end
```

That is, loop the given arguments and map them through `inspect/1` function, then join them all together with commas.

Finally, we need to modify the `def` macro to use our new helper function and print the function name, arguments, and result:

```
  defmacro def(definition={name, _, args}, do: content) do
    quote do
      Kernel.def(unquote(definition)) do
        IO.puts :stderr,
                ">>> Calling #{unquote(name)} with #
{Tracer.dump_args(unquote(args))}"
        result = unquote(content)
        IO.puts :stderr, "<<< Result: #{Macro.to_string result}"
        result
      end
    end
  end
```

In this version, we inject a call to `IO.puts/2` to print the called function, its arguments, and its result.

The whole completed `Tracer` module looks similar to the following code:

```
defmodule Tracer do
  def dump_args(args) do
    args |> Enum.map(&inspect/1) |> Enum.join(", ")
  end
  defmacro def(definition={name, _, args}, do: content) do
    quote do
      Kernel.def(unquote(definition)) do
        IO.puts :stderr,
                ">>> Calling #{unquote(name)} with #
{Tracer.dump_args(unquote(args))}"
        result = unquote(content)
        IO.puts :stderr, "<<< Result: #{Macro.to_string result}"
        result
      end
    end
  end
  defmacro __using__(_) do
    quote do
      import Kernel, except: [def: 2]
      import unquote(__MODULE__), only: [def: 2]
    end
  end
end
```

Now, in another module, define a simple `sort` module and include a line at the top for using the `Tracer` module:

```
defmodule Quicksort do
  use Tracer
  def sort(list), do: _sort(list)
  defp _sort([]), do: []
  defp _sort(l = [h|_]) do
    (l |> Enum.filter(&(&1 < h)) |> _sort)
    ++ [h] ++
    (l |> Enum.filter(&(&1 > h)) |> _sort)
  end
end
```

We've probably seen this module a few times; the major difference is that we've only added the `use` expression at the top. This simple change allows us to see the function call of the top-level, `sort`:

```
% elixirc tracer.exs% elixirc quicksort.exs
Loading the module into iex:
iex(1)> import Quicksort
nil
iex(2)> 1..10 |> Enum.reverse |> Quicksort.sort
>>> Calling sort with [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
<<< Result: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Here, we see the output of the macro injecting its `IO.puts` calls for our function and we also see the result of the function itself.

Static data to functions

Since macros are expanded at compile-time, some interesting things can be done during this injection. This way, creating functions from near static data is not only a possibility, but something that is used regularly to support MIME type translations, or Unicode character data support.

For example, Elixir itself uses Unicode code points in a text file to create the functions related to `String.upcase/1`, `String.downcase/1`, and others for complete Unicode support. If a new code point is added tomorrow, the code itself does not have to change; the code point needs only to be added to the source text file, and the project recompiled.

Let's look at how Elixir defines the `String.Unicode` module, and specifically, how the `String.downcase/1` function is implemented.

Under the Elixir source tree, there is a folder for Unicode data points. Also, in this folder, is an Elixir source file, `unicode.ex`, which defines the `String.Unicode` module.

At the beginning of this module are a few list comprehensions and some uses of `Enum.reduce/3`:

```
data_path = Path.join(__DIR__, "UnicodeData.txt")
{codes, whitespace} = Enum.reduce File.stream!(data_path), {[], []}, fn(line,
{cacc, wacc}) ->
  [codepoint, _name, _category,
   _class, bidi, _decomposition,
   _numeric_1, _numeric_2, _numeric_3,
   _bidi_mirror, _unicode_1, _iso,
   upper, lower, title] = :binary.split(line, ";", [:global])
title = :binary.part(title, 0, byte_size(title) - 1)
  cond do
    upper != "" or lower != "" or title != "" ->
      {[{to_binary.(codepoint),
         to_binary.(upper),
         to_binary.(lower),
         to_binary.(title)} | cacc],
       wacc}
    bidi in ["B", "S", "WS"] ->
      {cacc, [to_binary.(codepoint) | wacc]}
    true ->
      {cacc, wacc}
  end
end
```

This reads the file, `UnicodeData.txt`, and streams each line through the provided anonymous function, in turn, breaking down each Unicode code point into a list of tuples.

Later, a list comprehension is used to define the `downcase/1` function:

```
def downcase(string), do: downcase(string, "")
for {codepoint, _upper, lower, _title} <- codes, lower && lower != codepoint do
  defp downcase(unquote(codepoint) <> rest, acc) do
    downcase(rest, acc <> unquote(lower))
  end
end
defp downcase(<<char, rest :: binary>>, acc) do
  downcase(rest, <<acc :: binary, char>>)
end
defp downcase("", acc), do: acc
```

The first line defines the public interface of the `String.Unicode.downcase/1` function, which returns the result of the privately-defined `downcase/2`.

Next, a list comprehension is used to unpack the list of codes and extract the lower case character of each. This is then used to generate a large number of the `downcase/2` functions that are capable of being individually matched.

If there's no match, it will match against the next version, `<<char, rest :: binary>>, acc`. Finally, the last function of the four provides the return value when processing the provided binary is complete. That is, it matches against the empty string, `""`, and returns the accumulator value, `acc`.

Four very similar functions are defined for `upcase/1`.

This example isn't necessarily using many macro features; however, the pattern of loading some static data from a file and generating functions for a module is a common enough pattern, which is useful to see.

Testing macros

Testing modular code is generally pretty easy and considered a good thing. Testing macros can be a little like testing black magic. At first blush, it's not always obvious how to even approach testing macros. However, one general rule can help steer the testing suite in the right direction: test the generated code, not the generation code. That is, test the result of the macro, not the macro's ability to generate code.

It's generally much harder to create a proper harness around macros than it is to generate a harness around the code generated by macros. For example, if we wrote a `while` macro, we could write a test for it using a block of code that uses the `while` expression:

```
1.  defmodule WhileTest do
2.    use ExUnit.Case
3.    test "while loops while truthy" do
4.      pid = spawn(fn -> :thread.sleep(:infinity) end)
5.      send self, :one
6.      while Process.isAlive?(pid) do
7.        receive do
8.          :one -> send self, :two
9.          :two -> send self, :three
10.         :three ->
11.           Process.exit(pid)
12.           send self, :done
13.       end
14.     end
15.     assert_received :done
16.   end
```

Thus, if the assertion fails, we know that the termination of the loop is faulty. Some more tests could be added, but the idea is the same.

Similarly, for testing the generated functions, test the expected behaviour of the generated functions. In the case of the standard library, here are the tests for `String.upcase/1`:

```
1.  test "upcase" do
2.    assert String.upcase("123 abcd 456 efg hij ( %$#) kl mnop @ qrst = -_
    uvwxyz") == "123 ABCD 456 EFG HIJ ( %$#) KL MNOP @ QRST = -_ UVWXYZ"
3.    assert String.upcase("") == ""
4.    assert String.upcase("abcD") == "ABCD"
5.  end
6.  test "upcase utf8" do
7.    assert String.upcase("& % # àáâ ãäå 1 2 ç æ") == "& % # ÀÁÂ ÃÄÅ 1 2 Ç Æ"
8.    assert String.upcase("àáâãäåæçèéêëìíîïðñòóôõöøùúûüýþ") ==
    "ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞ"
9.  end
10. test "upcase utf8 multibyte" do
11.   assert String.upcase("straße") == "STRASSE"
12.   assert String.upcase("áüÈß") == "ÁÜÈSS"
13. end
```

These, in themselves, don't seem to be sufficient for testing the `String.upcase/1` function. But knowing how `String.upcase/1` is implemented, this proves to be entirely sufficient.

It shouldn't be necessary to create a one-to-one testing of all the generated functions, but a large enough sample to know the generated functions are behaving correctly.

Remember that testing macros should be treated the same as testing the public interface of a module. Test the functions that are the result of the injected code. This changes it from a problem of testing code generation to a regular module testing problem.

Summary

In this chapter, we covered the following topics:

- We learned about Typespecs which are used as a means for documenting code (with code) such that other programmers (and ourselves) will know at a glance the expected types of certain functions.
- Then we moved on to behaviours can be thought of akin to interfaces from OO languages.
- Protocols are a means of performing high-level pattern matching and function dispatching for certain actions.
- Finally, we discussed Elixir macros and how they enable metaprogramming.

We have reached the end of this course. To summarize, we understood functional programming and got to know the origin of Elixir. We created applications using the Mix tool set. We then explored Elixir to create resilient and scalable applications. We learned to design program-distributed applications and systems. Finally, we explored to do more with less using Elixir's metaprogramming.

We hope you've learned quite a lot through this course and enjoyed it as much as we did. Thank you!

Course Content

udemy.com/elixir-scalable-and-efficient-application-development/learn/v4/t/lecture/7102978

The GNU/Linux- and Unix-based OSes are a great example of an existing process tree that can be studied and inspected. It has a root process, typically `init` with process ID `1`, and it is the ancestor process of all the child processes. Each child process can itself create more child processes. The structure of this chain is a tree rooted at PID `1`.

Process trees in Elixir/Erlang are not too dissimilar. There is a root process for the runtime and application controller. The applications loaded and started upon the startup of the VM are children of the application controller or immediate parent process.

However, OTP takes process trees to a new level with supervision trees. Supervision trees are similar to process trees except that they describe slightly different concepts. Process trees only describe the parent-child relationship between processes, whereas supervisor trees describe the class of parent-child supervisors for running processes and the strategies used to restart dead children:

Supervisor basics A simple example of a supervisor tree is a single supervisor monitoring a single process. The supervisor listens for events about the child process, and will take certain action on the receipt of certain events, the most basic of which is doing nothing. This is the basic premise of process supervision. A supervisor process starts and restarts a child process upon failure or an abnormal exit reason.

The supervisor is itself a process, so it is a natural extension that a supervisor monitor have other supervisors, and thus, the concept of a supervision tree arises. Furthermore, the supervisor can have different restart strategies for each monitored process.

Restart strategies describe how and what actions a supervisor should take when receiving notification of a dead child process. OTP describes four different strategies for use—one for one, one for all, rest for one, and simple one for one.

One for one describes what you might expect—restart the process that has died:

Supervisor one for one

One for all may also seem intuitive; this strategy tells the supervisor to restart all or every process when a single process fails:

Supervisor one for all

Rest for one restarts all the child processes—from the failed to the last. For example, say we have four processes defined in a list, if process 2 dies, then process 3 and 4 will be killed and then process 2, 3, and 4 will be restarted:

Supervisor rest for one Simple one for one is a bit of a special case. It tends to be the most complex strategy of the four. The processes under a simple one for one supervisor are generally the same and aren't necessarily started statically, when the supervisor starts. There are also internal, structural differences between one for one and simple one for one. That is, one for one stores the child processes as a list, whereas simple one for one stores it as a dictionary making the simple one for one much faster with larger numbers of child processes.

Along with supervisor strategies, there are also restart options for child or worker processes. The default used restart option is `:permanent`, which means the process will always be restarted, even if the process is killed normally, for example, `{:shutdown, :normal}`. A common option used for simple one for one is `:temporary` — the process will never be restarted. This is often useful when the supervisor isn't the one process starting the processes, for example, connection pooling each process is associated with a connection. If the connection dies or it is reset, the supervisor monitoring the processes should not restart them. The third, final option for restarts is `:transient`, which means that the supervisor will only restart the process if it was exited abnormally. That is, if the process shutdown reason is anything other than `:normal`, `:shutdown`, or `{:shutdown, term}`, the supervisor will restart the process.

Creating supervisors in Elixir is very similar to creating other OTP behaviours. You need to define a module that uses the `Supervisor` behaviour and a few functions, and start it.

Let's create a simple supervisor for the `KV` process from before:

```
1.  defmodule KV.Supervisor do
2.    use Supervisor
3.    def start_link do
4.      Supervisor.start_link(__MODULE__, :ok)
5.    end
6.    def init(:ok) do
7.      children = [worker(KV, [])]
8.      supervise(children, strategy: :one_for_one)
9.    end
10.  end
```

As usual, there's a `start_link` function, which uses `Supervisor.start_link` to perform the synchronous starting and linking of the supervisor process to the current process. The `init/1` function is where the supervising configuration is set. We define a list of children or monitored processes. These could be worker processes, or other supervisors.

There's another helper function, `supervisor/2`, similar to `worker/2`. The difference is that supervisors have an infinite timeout value for shutdown, whereas workers default to 5 second timeouts.

Loading both modules into an IEx session, we can see the supervisor keep the `KV` process running, even after exits and failures:

```
iex(1)> import_file "kv.exs"
...
iex(2)> {:ok, sup} = KV.Supervisor.start_link
{:ok, #PID<0.71.0>}
iex(3)> Supervisor.which_children(sup)
[{KV, #PID<0.72.0>, :worker, [KV]}]
iex(4)> KV.put(:a, 42)
:ok
iex(5)> KV.get(:a)
42
```

The `KV` process is started after the supervisor starts and it works as we expect. But now, we can cause the `KV` process to exit and see that it restarts:

```
iex(6)> Process.whereis(KV) |> Process.exit :normal
true
iex(7)> KV.get(:a)
nil
```

There is no semblance of persistence with the `KV` process, but it is restarted by the `KV.Supervisor`. We can add persistence by adding more to the supervision tree.

Fail fast(er)

Why would we want process supervision? What is the benefit of using process supervision over exception handling?

Exception handling attempts to keep the process or thread alive by catching errors and having them propagate using a series of the `try-catch` blocks. This often will fail to raise the error to the appropriate level and, worse, it can leave the process or context in a very bad state, possibly masking more subtle, and farther reaching issues.

Failing fast and using process supervision, we can avoid the bad state issue entirely: if a process fails for any reason, restart it. Adding good logging around the supervisors and workers, error propagation to the correct party can be far easier.

Designing with supervisors

The concepts around process supervision are not necessarily complicated. The complexity usually arises in how to design applications using supervisors, what order to define the supervision tree, what level of the tree certain processes should be, and so on. This is not an easy problem, but it is not impossible.

There are some clues and other signs to look for that will guide the design. Look for process dependence. Does a process depend on another? The latter should be higher in the supervisor tree. Does a process depend on an entire process tree? The subtree should be defined before. Are there two processes that are co-dependent? Attempt to isolate them into their own supervisor tree and use `:one_for_all`.

Assumptions of the OTP process initialization

Along with the design of applications using supervisors, it's helpful to know or make explicit the assumptions being made when starting OTP processes. If a `GenServer` process on initializing makes a call to a database, the assumption is that this database will always be alive when this process starts. If it is not, then the process can't start. However, if the database can be expected to not always be available, the assumption of the defined OTP process will fail, and likely cause the entire application to fail because of the number of restarts will be exceeded. Therefore, the assumptions of what is available during initialization and startup of OTP processes must be known a priori.

Summary

In this chapter, we had a look at modifying the article content type, basic page content type and then adding this content to our Drupal website. We moved on to creating roles for different types of users for our blogging website. We explored to setup permissions for the roles and finally we learned to create couple of users.

Quiz Time!

Quiz 10 | 2 Questions

Start Quiz Skip Quiz >

Question 1:

Identify the function that is included in the `GenServer` modules:

- `composer_cast`
- `handle_cast`
- `manage_cast`
- `issues_cast`

Question 2:

Which of the following macros are used to define the behaviour of Elixir?

- `callback`
- `defncallback`
- `defcall`
- `defcallback`