03 GenServer

GenServer 1

In the folder `03-gen_server`, you will find `frequency` project with an implemented GenServer.

Start the `iex` shell inside the project by running:

```
$ iex -S mix
```

Compare the GenServer implementation with `Frequency` and `Frequency2` modules from the previous lesson.

In the `iex` shell use the functions provided by GenServer module to interact with the `Frequency` GenServer.

Start the process:

```
> {:ok, pid} = GenServer.start(Frequency, [])
```

Issue `allocate` and `deallocate` calls:

```
> GenServer.call(pid, :allocate)
> GenServer.call(pid, {:deallocate, 1})
```

Stop the process:

```
> GenServer.stop(pid)
```

Question Is it possible to start multiple `Frequency` GenServers? If yes, why?

Useful tips

- You can use Observer to visually inspect the `Frequency` GenServer:

  ```
  > :observer.start()
  ```

- If Observer isn't working for some reason you can use `:sys.get_state/1` to get information about the current state of the process:

  ```
  > {:ok, pid} = GenServer.start(Frequency, [])
  > :sys.get_state(pid)
  ```

GenServer 2

So far when implementing GenServers we have used simple data structures (maps) to store state, but in the real world state can be more complex than that. We can easily add a layer of control by passing an Elixir struct as a state. Then we are able to leverage more complex mechanisms to validate data or be more descriptive about our state data.

GenServer: UserCache In the `03-gen_server/frequency/lib` directory, create `user_cache.ex` file with the following contents:

```elixir
defmodule UserCache do
  use GenServer

  @impl true
  def init(_) do
    {:ok, %{}}
  end
end
```

The state that will be stored by this process will be represented by two nested structures. The first one:

```elixir
defmodule State do
  defstruct [:amount, :users]
end
```

And the User structure:

```elixir
defmodule User do
  defstruct [:id, :name]
end
```

NOTE: Such state structures might be more complex, could live in a different directory, and so on. For the sake of the exercise we are putting them in the same module.

Now that we have structures that will hold the state, we can update implementation of the GenServer:

```elixir
defmodule UserCache do
  use GenServer

  defmodule User do
    defstruct [:id, :name]
  end

  defmodule State do
    defstruct [:users, :amount]
  end

  @impl true
  def init(_) do
    {:ok, %State{amount: 0, users: []}}
  end
end
```

Exercises

- Add `handle_call/3` implementation that will add a new user to the cache. Example input: `{:add, 1, "Name"}`, output: `:ok`, state modification: a new user is added to the `users` list, `amount` is increased by 1.
- Add `handle_call/3` to fetch the current state.
- Add `handle_call/3` to delete a user from the cache. Input: `{:remove, 1}`, output: `:ok`, state modification: `amount` decreased by 1, the user with the specified id should be removed from the `users` list.