Concurrent Programming in Erlang

futurelearn.com/courses/concurrent-programming-erlang/1/steps/169249

This exercise will give you a chance to implement the frequency server as a `gen_server`, with the same interface as for the “raw Erlang” implementation.

A template module `gf.erl` for you to use is available from the foot of this page, and the supporting file `frequency2.erl` is also avaiable. There is a further file, `frequency_gen_server.erl`, which provides *our* solution to this exercise - please do attempt this exercise yourself first, before referring and comparing to our solution.

Use the comments on this step to share your approaches and solutions, and then in the next step we’ll discuss how you’ve got on with this exercise.

Implementing the `gen_server` behaviour

To implement a particular `behaviour`, such as `gen_server`, the module needs to contain a behaviour declaration, thus:

```
-behaviour(gen_server).
```

This should follow the module declaration, and “behaviour” can be spelled with or without a “u”.

It is also necessary to export the interface functions, as well as the implementations of the functions in the callback interface, which we turn to next.

The interface functions

The interface presented to the outside world by the frequency server consists of these four functions:

- `start/0` start the frequency server
- `stop/0` shut down the frequency server
- `allocate/0` allocate a frequency, if one is available, and return an error value if not
- `deallocate/1` deallocate the given frequency

These should be implemented as calls to functions in the `gen_server` module, including `start_link/4` (start the server), `call/2` (synchronous message handling), `cast/2` (asynchronous message handling) and so on.

The callback functions

For the `gen_server` to have the specific behaviour of the frequency server, these callback functions need to be implemented:

- `init/1` initialise the state of the server, based on the argument
- `terminate/2` code performed on termination, dependent on the reason for termination and the current state.

- `handle_cast/2` handle a message asynchronously, taking the message and current state as arguments
- `handle_call/3` handle a message synchronously, taking the message, the PId of the sender and current state as arguments

Call or cast?

Have you implemented `stop/0` and `deallocate/0` as calls or casts? The natural way of doing so would be as calls, since we've written them with a reply, but they don't need to have a reply; so, if you have done this, try rewriting them as casts. If you have already done this, how would you write them as calls?

Adding additional functionality

If you want to take things further, add a function `report/0` that reports the numbers of free and allocated frequencies: will you implement this as a call or a cast?

You could also add a function to introduce new frequencies to the server: the reply should confirm the addition, or give an error if a frequency is duplicated in the addition. Do you use call or cast for this functionality?