GenServer

A Generic Server Behaviour

What is GenServer

A behaviour module for implementing the server of a client-server relation.
-- Erlang docs
A behaviour is a way to say: give me a module as argument and I will invoke the following callbacks on it, with these argument and so on.
— Jose Valim

Other Behaviours

- Supervisor
- Application
- GenEvent
- GenFsm

Common server features

- Spawn a separate process
- Maintain state
- Handle requests and sending responses
- Server lifecycle

Implementer Provides:

- Initial state
- Kinds of messages the server handles
- When to reply or not
- What to reply with
- State change
- Resource cleanup on termination

How does it work?

Genserver expects the module it is used in to define a set of callbacks

The callbacks are invoked when the corresponding GenServer functions are called

GenServer callbacks

- `init(args)` - called on start, sets initial state
- `handle_call(msg, {from, ref} , state)` - handles sync msgs
- `handle_cast(msg, state)` - handles async messages
- `handle_info(msg, state)` - all other messages
- `terminate(reason, state)` - server about to terminate
- `code_change (old_vsn, state, extra)` - hot code swapping

These are implemented and never called directly

GenServer functions

- `start` and `start_link` - callback
- `call`
- `cast`
- `reply`

These form the basis of the client interface

Implementation

- Define the callbacks
- Interface functions (API)
- Usually in the same file

Implementation

Use the Genserver behaviour in your module

```
defmodule MyStore do
  use GenServer
end
```

Implementation - init

define *init*

```
def init(_) do
  { :ok, Map.new }
end
```

Returns a tuple of :ok and the initial server state

Implementation - call

call - synchronous

```
def handle_call({:get, key}, _from, state) do
  {:reply, Map.get(state, key), state}
end
```

Returns a tuple consisting of :reply, the data to be sent back to the caller and the new state

Implementation - cast

cast - asynchronous

```
def handle_cast({:put, key, value}, state) do
  {:noreply, Map.put(state, key, value)}
end
```

Returns a tuple consisting of :noreply, the data to be sent back to the caller and the new state

Implementation - other

Other callbacks left as an exercise for the reader

Callbacks can have different return values allowing slightly different behaviour

- {:reply, reply, new_state}
- {:reply, reply, new_state, timeout}
- {:reply, reply, new_state, :hibernate}

Running the server

Start the server

```
{:ok, store} = GenServer.start(Store, nil)
```

calls the **`init`** callback with args as the argument

Using GenServer directly

```
iex(4)> {:ok, store } = GenServer.start(Store, nil)
Store init
{:ok, #PID<0.112.0>}
iex(5)> GenServer.cast(store, {:put, :elixir, 1000})
Store put
:ok
iex(6)> GenServer.call(store, {:get, :elixir})
Store get
1000
```

Implementation - Define the client interface

Define functions that call the GenServer function with the appropriate arguments.

```
defmodule Store do
  use GenServer
  # client interface
  def start do
    GenServer.start(__MODULE__, nil)
  end

  def put(pid, key, value) do
    GenServer.cast(pid, {:put, key, value})
  end

  def get(pid, key) do
    GenServer.call(pid, {:get, key})
  end
```

DEMO TIME

RESOURCES

genserver101 repo
Elixir in Action
The Little Elixir and OTP Guidebook

@*martinS*
martin@stannard.id.au