OTP

By Robert Brown

OTP

OTP = Open Telecom Platform

OTP is a fundamental library to Erlang's fault tolerance

Erlang and OTP are sometimes use synonymously

OTP is built on processes

Processes

Basic process operations:

1. `spawn`
2. `send`
3. `receive`
4. `Process.monitor`

Supervisor

Supervisor

- Handles fault tolerance
- Restarts crashed processes as specified
- Lots of configuration options not covered here

Restart Options

- `:permanent` (default)
- `:transient`
- `:temporary`

Permanent

Always restart

Transient

Restart on error

Temporary

Never restart

Supervisor Strategy

- `:one_for_one` (default)
- `:one_for_all`
- `:rest_for_one`
- `:simple_one_for_one` (deprecated)

One for One

Only restart the crashing process

One for All

Restart all processes

All for One

Only for Musketeers

Rest for One

Restart all processes started after the crashing process

Rest for One

Processes are started in order given.

Say a Supervisor starts 5 processes.

If process 3 crashes then processes 3, 4, and 5 are restarted.

If process 5 crashes then only process 5 is restarted.

Rest for One

- Good for when later processes depend on earlier processes.
- May be better to use nested supervisors to make the relationship explicit.
- Favor `:one_for_one` or `:one_for_all`.

Simple One for One

- Deprecated
- Similar to `:one_for_one`.
- Good for when children are added dynamically.
- Replaced by `DynamicSupervisor`

Child Spec

Child Spec

Provides info for Supervisor to start the process

Sample Child Spec

```
def child_spec(arg) do
  %{
    id: __MODULE__,
    start: {__MODULE__, :start_link, [arg]}
  }
end
```

Child Spec Options

- `:id` (required)
- `:start` (required)
- `:restart`
- `:shutdown`
- `:type`
- `:modules`

GenServer Child Spec

`GenServer` provides a default `child_spec` with

```
use GenServer
```

Override Child Spec

Any spec field can be overriden like this:

```
use GenServer, restart: :transient
```

Built-in Supervisors

- `Supervisor`
- `DynamicSupervisor`
- `Task.Supervisor`

Module-based Supervisor

Module-based Supervisor

Typically use `Supervisor` at application level

Use Module-based Supervisors underneath that

Can also use `Task.Supervisor` and `DynamicSupervisor`.

Application Supervisor

```
defmodule MyApp.Application do
  use Application

  def start(_type, _args) do
    children = [
      Child1,                # Same as {Child1, []}
      {Child2, [:hello]},
      Supervisor.child_spec({Child3, [:hello]}, id: OtherID, restart: :transient)
    ]

    opts = [strategy: :one_for_one, name: MyApp.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

Module-based Supervisor

```elixir
defmodule MyApp.Supervisor do
  # Automatically defines child_spec/1
  use Supervisor

  def start_link(init_arg) do
    Supervisor.start_link(__MODULE__, init_arg, name: __MODULE__)
  end

  @impl true
  def init(_init_arg) do
    children = [
      Child1,               # Same as {Child1, []}
      {Child2, [:hello]},
      Supervisor.child_spec({Child3, [:hello]}, id: OtherID, restart: transient)
    ]

    Supervisor.init(children, strategy: :one_for_one)
  end
end
```

DynamicSupervisor

- Started with no children
- Often supervises many processes spawned from one child spec
- Only supports `:one_for_one`

Task.Supervisor

- `DynamicSupervisor` specifically for `Task`
- Use `Task.Supervisor.async_nolink` if you don't want a `Task` to be linked to the current process
- This is to avoid dangling processes

GenServer

GenServer

Short for "Generic Server"

A generic process that encapsulates behavior **and** state data

GenServer Alternatives

- Task
- Agent

Which Should I Use?

	Open Logic	Isolated Logic
Open State	`Module`	`Task`
Isolated State	`Agent`	`GenServer`

Module

```
# A simple function call
output = Module.fun(input)
```

Task

```
{:ok, task} = Task.async(fn ->
  # Do something
end)

# Wait until done or timeout
output = Task.yield(task)

# Wait until done or terminate on timeout
output = Task.await(task)
```

Agent

```
# Start process with initial state
{:ok, pid} = Agent.start_link(fn -> state end)

# Update state
Agent.update(pid, fn x -> ... end)

# Return current state (or some transformation)
output = Agent.get(pid, fn x -> x end)
```

GenServer

```
{:ok, pid} = GenServer.start_link(...)

# Send without expecting response
GenServer.cast(pid, msg)

# Send and wait for response or terminate on timeout
output = GenServer.call(pid, msg)
```

Demo

Questions?