Episode 15: Supervisor

Summary

- Erlang's error handling philosophy
- How to construct a supervision tree
- Real-world examples

Supervisors

- In Elixir, you do not write code to handle unknown errors
- Instead of "what classes do I need?" you ask, "what **processes** do I need?"
- Supervisors watch other processes and automatically restart them if they crash.

Supervision trees enable

1. Error isolation
2. Elegant error recovery
3. Self-healing systems

Elixir 1.5 Changes

> *Note*: Since Supervisors were revamped in Elixir 1.5, the example in the video has been deprecated.

For the new method of using Supervisors, see the documentation first and the `learn_supervisor` example in this repository.

As a result, the code samples for headings past "Strategies" shown below may be outdated.

Module-based Supervisors vs. Implicit Supervisors

There are two ways to create a supervisor.

1. Call `Supervisor.start_link/2` with a list of children (automatically initialized)
   - "A supervisor started with this function is linked to the parent process and exits not only on crashes but also if the parent process exits with `:normal` reason."
2. Call `Supervisor.start_link/3` in its own module. Children are manually initialized in an `init/1` function

1. Create your module (e.g., Sup)
2. Add `use Supervisor`. This automatically defines `child_spec/1`
3. Write `start_link/1` and call `Supervisor.start_link/3`
4. Write `@impl true` with `init/1`. Make a list of children and call `Supervisor.init/2`

The docs recommend to make a supervisor without a callback module only at the top of supervision tree. All other supervisors should be module-based for the automatic `child_spec/1`.

Specifications

Since Elixir 1.9. Subject to change.

Supervisor.start_link(children, options)

```
start_link(module(), term()) :: on_start()
```

```
start_link(
  [:supervisor.child_spec() | {module(), term()} | module()],
  options()
) ::
  {:ok, pid()}
  | {:error, {:already_started, pid()} | {:shutdown, term()} | term()}
```

src

Supervisor.init(children, options)

```
init([:supervisor.child_spec() | {module(), term()} | module()],
↪  [init_option()]) ::
  {:ok, tuple()}
```

src

child_spec/0

> *From src: Supervisor -> child_spec/0*

```
child_spec() :: %{
  :id => atom() | term(),
  :start => {module(), start_fun :: atom(), args :: [term()]},
  optional(:restart) => :permanent | :transient | :temporary,
  optional(:shutdown) => timeout() | :brutal_kill,
  optional(:type) => :worker | :supervisor,
  optional(:modules) => [module()] | :dynamic
}
```

Child Specification

> *From src: Supervisor/Child Specification*

The child specification describes how the supervisor starts, shuts down, and restarts child processes.

The child specification contains 6 keys. The first two are required, and the remaining ones are optional:

- `:id` - any term used to identify the child specification internally by the supervisor; defaults to the given module. In the case of conflicting :id values, the supervisor will refuse to initialize and require explicit IDs. This key is required.
- `:start` - a tuple with the module-function-args to be invoked to start the child process. This key is required.
- `:restart` - an atom that defines when a terminated child process should be restarted (see the "Restart values" section below). This key is optional and defaults to `:permanent`.
- `:shutdown` - an atom that defines how a child process should be terminated (see the "Shutdown values" section below). This key is optional and defaults to `5000` if the type is `:worker` or `:infinity` if the type is `:supervisor`.
- `:type` - specifies that the child process is a `:worker` or a `:supervisor`. This key is optional and defaults to `:worker`.

There is a sixth key, `:modules`, that is rarely changed. It is set automatically based on the value in `:start`.

Restart values (`:restart`)

The `:restart` option controls what the supervisor should consider to be a successful termination or not. If the termination is successful, the supervisor won't restart the child. If the child process crashed, the supervisor will start a new one.

The following restart values are supported in the `:restart` option:

- `:permanent` - the child process is always restarted.
- `:temporary` - the child process is never restarted, regardless of the supervision strategy: any termination (even abnormal) is considered successful.
- `:transient` - the child process is restarted only if it terminates abnormally, i.e., with an exit reason other than `:normal`, `:shutdown`, or `{:shutdown, term}`.

For a more complete understanding of the exit reasons and their impact, see the "Exit reasons and restarts" section.

Shutdown values (`:shutdown`)

> *From Supervisor/Shutdown Values*

The following shutdown values are supported in the :shutdown option:

- `:brutal_kill` - the child process is unconditionally and immediately terminated using `Process.exit(child, :kill)`.
- any integer >= 0 - the amount of time in milliseconds that the supervisor will wait for its children to terminate after emitting a `Process.exit(child, :shutdown)` signal. If the child process is not trapping exits, the initial `:shutdown` signal will terminate the child process immediately. If the child process is trapping exits, it has the given amount of time to terminate. If it doesn't terminate within the specified time, the child process is unconditionally terminated by the supervisor via `Process.exit(child, :kill)`.
- `:infinity` - works as an integer except the supervisor will wait indefinitely for the child to terminate. If the child process is a supervisor, the recommended value is `:infinity` to give the supervisor and its children enough time to shut down. This option can be used with regular workers but doing so is discouraged and requires extreme care. If not used carefully, the child process will never terminate, preventing your application from terminating as well.

Strategies

Strategy	Description
`:one_for_one`	if a child process terminates, only that process is restarted.

Strategy	Description
`:rest_for_one`	if a child process terminates, the terminated child process and the rest of the children started after it, are terminated and restarted.
`:one_for_all`	if a child process terminates, all other child processes are terminated and then all child processes (including the terminated one) are restarted.
`:simple_one_for_one`	for dynamically adding child processes. (*deprecated, replaced by* `DynamicSupervisor`)

> **Suggestion**: use `:one_for_one` until it doesn't work for you

Tips

1. You can create a supervised mix project with `mix new app_name --sup`
2. GenServer processes can be named
3. Worker processes can have IDs
4. Supervisors can be supervised
5. Try a built-in Erlang database for storage

1. Mix Project with a Supervisor

Add `--sup` to generate a supervised mix project.

```
mix new my_app --sup

# lib/my_app.ex will be a supervisor
```

2. GenServer Processes can be Named

You can use `GenServer` processes **without knowing their `pid`** by using the `:name` option:

```
GenServer.start_link(Game.Cache, [], name: Game.Cache)
GenServer.cast(Game.Cache, {:save, state})
```

This is helpful for reflecting your model of the supervisor tree in your code.

3. Workers can have IDs

Using the `:id` option:

```
worker(SupervisedProcess, [], id: "some-id")
```

4. Supervisors can be Supervised

Supervising other supervisors allows for complex supervision trees. Use `supervisor` instead of `worker`:

```
supervisor(Game.Supervisor, [])
```

5. Try ETS for Caching

Don't go straight to a third-party tool if you need a data store. Give the `ets` or `dets` Erlang modules a try.

- `ets` - memory-based store
- `dets` - disk-based store
- Mnesia - relational database