1. Process

Conveniences for working with processes

```
# spawn_options().t
Fun = function()
Options = [spawn_opt_option()]
priority_level() = low | normal | high | max
max_heap_size() =
    integer() >= 0 |
    {size => integer() >= 0,
      kill => boolean(),
      error_logger => boolean()}
message_queue_data() = off_heap | on_heap
spawn_opt_option() =
    link | monitor |
    {priority, Level :: priority_level()} |
    {fullsweep_after, Number :: integer() >= 0} |
    {min_heap_size, Size :: integer() >= 0} |
    {min_bin_vheap_size, VSize :: integer() >= 0} |
    {max_heap_size, Size :: max_heap_size()} |
    {message_queue_data, MQD :: message_queue_data()}
```

```
# Process.info(pid) (:erlang.process_info/1)
[
  current_function: {:gen_server, :loop, 7},
  initial_call: {:proc_lib, :init_p, 5},
  status: :waiting,
  message_queue_len: 0,
  links: [],
  dictionary: [
    "$initial_call": {:erl_eval, :"-expr/5-fun-3-", 0},
    "$ancestors": [#PID<0.153.0>, #PID<0.75.0>]
  ],
  trap_exit: false,
  error_handler: :error_handler,
  priority: :normal,
  group_leader: #PID<0.64.0>,
  total_heap_size: 233,
  heap_size: 233,
  stack_size: 11,
  reductions: 64,
  garbage_collection: [
    max_heap_size: %{error_logger: true, kill: true, size: 0},
    min_bin_vheap_size: 46422,
    min_heap_size: 233,
    fullsweep_after: 65535,
    minor_gcs: 0
  ],
  suspending: []
]
```

Functions

```
# Spawn/exit/hybernate
spawn(fun, spawn_options())
| spawn(m, f, a, spawn_options())
exit(pid, reason)
alive?(pid)
hibernate(m, f, a)

# Flag
flag(flag, value) | flag(pid, flag, value)

# Send
send(dest, msg, options)
send_after(dest, msg, time, opts \\ [])

# Link to calling process
link(pid_or_port)
unlink(pid_or_port)

# Monitor from calling process
monitor(item)
```

```
demonitor(monitor_ref, options \\ [])

# Registration
register(pid_or_port, name)
unregister(name)
registered() :: [name]
whereis(name) :: pid | nil

# send_after/3 timers
read_timer(timer_ref)
cancel_timer(timer_ref, options \\ [])

# Debugging
sleep(timeout)
info(pid)
list() # list of all running PIDs
```

---

2. GenServer behaviour

used for:

- mutable state (by abstracting receive loop)
- enabling concurrency
- isolating failures

Functions

start_link

Starts a GenServer process linked to the current process. Once the server is started, the `init/1` function of the given module is called with `init_arg` as its argument to initialize the server.

```
start_link(
    module_name,
    init_arg,
    options:
        name:
            atom
            | {:global, term}
            | {:via, module, name}
        timeout:
            msecs \\ :infinity
)
```

call

Makes a synchronous call to the server and waits for its reply.

```
call(
    server,
    request,
    timeout \\ 5000
) :: response
```

cast

Sends an asynchronous request to the server.

```
cast(
    server,
    request
) :: :ok
```

reply

Can be used instead of {:reply, _, _} inside `handle_call`. Can even be invoked from a different process.

```
reply(pid, term) :: :ok

def handle_call(:reply_in_one_second, from, state) do
  Process.send_after(self(), {:reply, from}, 1_000)
  {:noreply, state}
end

def handle_info({:reply, from}, state) do
  GenServer.reply(from, :one_second_has_passed)
  {:noreply, state}
end
```

stop

Synchronously stops server with given reason Normal reasons (no error logged):

```
:normal | :shutdown | {:shutdown, term}
```

```
GenServer
    stop(
        server,
        reason \\ :normal,
        timeout \\ :infinity
    ) :: :ok
```

Types

```
from() :: {pid(), tag :: term()}
```

Callbacks

Each callback returns tuple with some instruction as a first element `:reply, :noreply, :stop, :continue`.

handle: init

```
init(init_arg :: term()) ::
      {:ok, state}
    | {
        :ok,
        state,
        timeout() | :hibernate | {:continue, term()}
    }
    | :ignore
    | {:stop, reason :: any()}
```

handle: call

Invoked to handle synchronous call/3 messages.

```
handle_call(
    request :: term(),
    from(),
    state :: term()
) ::  {:reply, reply, new_state}
    | {:reply,
        reply,
        new_state,
        timeout() | :hibernate | {:continue, term()}
    }
    | {:noreply, new_state}
    | {
        :noreply,
        new_state,
        timeout() | :hibernate | {:continue, term()}
    }
    | {:stop, reason, reply, new_state}
    | {:stop, reason, new_state}

# Invoked to handle synchronous call/3 messages.
# call/3 will block until a reply is received, or call times out.
```

handle: cast

Invoked to handle asynchronous cast/2 messages.

```
handle_cast(
    request :: term(),
    state :: term()
) ::  {:noreply, new_state}
    | {:noreply,
        new_state,
        timeout() | :hibernate | {:continue, term()}
    }
    | {:stop, reason :: term(), new_state}
```

handle: info

```
handle_info(
    msg :: :timeout | term(),
    state :: term()
) :: return_same_as_handle_cast()
```

handle: continue

```
handle_continue(
    continue :: term(),
    state :: term()
) :: return_same_as_handle_cast()
```

Example of using `:continue` for additional work during init:

```
GenServer
    init
        :: {:ok, state, {:continue, :more_init}}

GenServer
    handle_call(:work, _, state)
        :: {
            :reply,
            _,
            state,
            {:continue, :more_work}
        }
end

GenServer
    handle_continue(:more_init, state)
        :: {:noreply, new_state}
```

`handle_continue` doesn't block caller process, and also ensures nothing gets in front of it in a GenServer's mailbox.

`handle_call` + `handle_continue` = respond + immediate handle_info. `init` + `handle_continue` = init + immediate handle_info.

handle: terminate

Invoked when the server is about to exit. It should do any cleanup required.

```
terminate(reason, state :: term())
    :: term()
when reason:
    :normal | :shutdown | {:shutdown, term()}
```

`reason` is exit reason.

It's called if any of callbacks (except `init`):

- returns a `:stop` instruction
- raises or returns invalid value
- traps exits and parent process sends an exit signal (probably not important if part of Supervision tree)
- If GenServer.stop or Kernel.exit is called

Terminate is not invoked for `System.halt(0)`

If part of Supervision tree, during tree shutdown, GenServer will receive an exit reason, depending on `child_spec` `shutdown` option:

- for `:brutal_kill` option `:kill` (terminate not called)
- for `{:shutdown, timeout}` option `:shutdown` (terminate called with time limit)

So it's not reliable...

Important clean-up rules belong in separate processes either by use of `monitoring` or by `link + trap_exit` (as in Supervisors)

Empty mailbox timeout mechanism

Timeout may be included in return value of `init` or `handle_*` callbacks. If no messages appear in mailbox during specified interval, `:timeout` info message will be sent.

```
GenServer
    init :: {:ok, _, timeout}
GenServer
    handle_* :: {_, _, timeout}

GenServer
    handle_info(:timeout, _)
```

Process monitoring

Monitoring, unlike linking is one-way. Monitored process is not affected by monitoring process failure.

```
ref = Process.monitor(pid)

# monitored process failure is handled in monitoring process:
handle_info({:DOWN, ref, :process, object, reason})
```

Debugging processes with the :sys module

```
:sys.get_state/2
:sys.get_status/2 - see :sys.process_status section
:sys.statistics/3 - see :sys.statistics section
:sys.no_debug/2

:sys.suspend/2
:sys.resume/2

# 2nd parameter is timeout
```

:sys.process_status

```
{:status, #PID<0.127.0>, {:module, :gen_server},
 [
   [
     "$initial_call": {:erl_eval, :"-expr/5-fun-3-", 0},
     "$ancestors": [#PID<0.104.0>, #PID<0.76.0>]
   ],
   :running,
   #PID<0.104.0>,
   [statistics: {{{2020, 3, 6}, {14, 1, 44}}, {:reductions, 251}, 1, 1}],
   [
     header: 'Status for generic server <0.127.0>',
     data: [
       {'Status', :running},
       {'Parent', #PID<0.104.0>},
       {'Logged events', []}
     ],
     data: [{'State', 4}]
   ]
 ]}
```

:sys.statistics

```
{:ok, pid} = Agent.start_link(fn -> 1 end)
Agent.update(pid, fn state -> state + 1 end)
```

```
:sys.statistics pid, :get

=> {:ok, :no_statistics}

:sys.statistics pid, :true
Agent.update(pid, fn state -> state + 1 end)
:sys.statistics pid, :get

=> {:ok,
 [
   start_time: {{2020, 3, 6}, {14, 1, 44}},
   current_time: {{2020, 3, 6}, {14, 1, 52}},
   reductions: 120,
   messages_in: 1,
   messages_out: 1
 ]}
```

---

3. Supervisor behaviour

Supervisor = Child specification + Supervision options.

Child specification

Creation

By `use GenServer`, `use Supervisor`, `use Task` `child_spec` callback is generated. It returns child specification, to be used by Supervisors:

```
child_spec().t :: %{
    id:
        term() \\ __MODULE__
    start:
        {m, f, a}
    restart:
        :permanent
        | :temporary
        | :transient
    shutdown:
        :brutal_kill
        | timeout
        | :infinity
    type:
        :worker
        | :supervisor
}
```

Usage in Supervisor

1. Without modification

```
Supervisor.init([
    supervisor_child_spec()
], strategy: ...)
```

2. With some fields overridden

```
Supervisor.init(
    [
        Supervisor.child_spec(supervisor_child_spec(), id: 1),
        Supervisor.child_spec(supervisor_child_spec(), id: 2)
    ],
    strategy: ...
)
```

child_spec:restart and exit reasons

```
                    :normal  |  :shutdown, {:shutdown, term} | other
permanent_restart:  yes      |  yes                          | yes
temporary_restart:  no       |  no                           | no
```

```
transient_restart:  no       |  no                           | yes

exit_logged:        no       |  no                           | yes
linked_proc_exit:   no       |  yes*                         | yes*
```

yes* - restart with same reason, unless trapping exits

child_spec:shutdown

Defaults: `:supervisor` - `:infinity`, `:worker` - `5000`

So if a Worker is trapping exits, it will receive `Process.exit(:shutdown)`, and will have 5000 to do cleanup, before being sent a `Process.exit(:kill)`.

Supervision options

```
# Examples for top-level or module-based Supervisors
Supervisor.start_link(children, options)
Supervisor.init(children, options)
```

```
strategy:
    :one_for_one \\ default
    | :rest_for_one
    | :one_for_all
max_restarts: # if reached, supervisor exits with :shutdown reason
    count \\ 3
max_seconds:
    count \\ 5
name:
    same_as_gen_server
```

Module-based configuration

Encapsulate Worker's configuration inside module

```
# Automatically defines child_spec/1
use GenServer, restart: :transient
```

Encapsulate Supervisor's configuration inside module

```
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 = [
      {Stack, [:hello]}
    ]

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

Types

```
supervisor_child_spec().t ::
    child_spec()
    | {module(), term()}
    | module()

supervisor_init_opts().t ::
    {:strategy, strategy()}
```

```
    | {:max_restarts, non_neg_integer()}
    | {:max_seconds, pos_integer()}
```

Functions

```
start_link/2 # same as GenServer
start_link/3 # same as GenServer
```

```
# Used inside init/1 callback
init(
    [supervisor_child_spec()],
    supervisor_init_opts()
) :: {:ok, tuple()}
```

```
stop(
    supervisor(),
    reason :: term(),
    timeout()
) :: :ok
```

```
child_spec(supervisor_child_spec(), overrides) ::
  child_spec()

# supervisor.child_spec() is often used to pass spec id,
# to be able to start multiple instances of same module.
# examples:
Supervisor.child_spec(
    { Stack, [:hello] },
    id: MyStack,
    shutdown: 10_000
})
Supervisor.child_spec(
    {Agent, fn -> :ok end},
    id: {Agent, 1}
)
```

```
count_children(supervisor()) :: %{
  specs: non_neg_integer(),
  active: non_neg_integer(),
  supervisors: non_neg_integer(),
  workers: non_neg_integer()
}
```

```
which_children(supervisor()) :: [
    {
        term() | :undefined, = child_id
        child() | :restarting, = pid
        :worker | :supervisor,
        :supervisor.modules()
    }
]
```

```
start_child(
    supervisor(),
    supervisor_child_spec()
)
::  {:ok, child()}
    | {:ok, child(), info :: term()}
    | {:error,
            {:already_started, child()}
            | :already_present
            | term()
    }
```

```
restart_child(
    supervisor(), child_id
)
::  {:ok, child()}
    | {:ok, child(), term()}
```

```
    | {:error, :not_found
            | :running
            | :restarting
            | term()
    }
```

```
# Terminates a running child process
terminate_child(
    supervisor(), child_id
)
::  :ok
    | {:error, :not_found}
```

```
# Deletes specification for a non-running child process
delete_child(
    supervisor(), child_id
)
::  :ok
    | {
        :error,
        :not_found | :running | :restarting
    }
```

---

4. DynamicSupervisor behaviour

DynamicSupervisor is started without Child Specification. Children are started on-demand.

Module-less:

```
children = [
    {
        DynamicSupervisor,
        strategy: :one_for_one,
        name: MyApp.DynamicSupervisor
    },
    ...
]

Supervisor.start_link(children, init_option())
```

Module-based:

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

  def start_child(foo, bar, baz) do
    spec = {MyWorker, foo: foo, bar: bar, baz: baz}
    DynamicSupervisor.start_child(
        __MODULE__,
        spec
    )
  end

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

  @impl true
  def init(_init_arg) do
    DynamicSupervisor.init(init_option())
  end
end
```

```
init_option() ::
  {:strategy, strategy()}
  | {:max_restarts, non_neg_integer()}
  | {:max_seconds, pos_integer()}
  | {:max_children, non_neg_integer() \\ :infinity}
  | {:extra_arguments, [term()]}
```

Where extra_arguments is `init` arguments, that will be prepended to `start_child` arguments for each started child.

Functions

```
child_spec/1
count_children/1
init/1
start_child/2
start_link/1
start_link/3
stop/3
terminate_child/2
which_children/1
```

```
start_child(
  Supervisor.supervisor(),
  Supervisor.child_spec()
  | {module(), args}
  | module()
)
::  {:ok, child()}
    | {:ok, :undefined} (if child process init/1 returns :ignore)
    | {:error, :max_children}
    | {:error, error}

terminate_child(
    Supervisor.supervisor(), pid()
)
::  :ok
    | {:error, :not_found}

which_children(supervisor())
:: [
    {
        :undefined = child_id
        child() | :restarting = pid
        :worker | :supervisor
        :supervisor.modules()
    }
    ...
]
```

TODO: fix elixir docs to show correct return values for DynamicSupervisor.start_child

---

5. Registry

A local, decentralized and scalable key-value process storage. It allows developers to lookup one or more processes with a given key.

Keys types: `:unique keys` - key points to 0 or 1 processes `:duplicate keys` - key points to n processes

Different keys could identify the same process.

Usage:

- name lookups (using the :via option)
- associate value to a process (using the :via option)
- custom dispatching rules, or a pubsub implementation.

Example 1: Registration using `via` tuple

```
{:ok, _} = Registry.start_link(keys: :unique, name: Registry.ViaTest)

VIA_no_value =
```

```
    {:via, Registry, {Registry.ViaTest, "agent"}}
VIA_value =
    {:via, Registry, {Registry.ViaTest, "agent", :hello}}

{:ok, _} =
    Agent.start_link(fn -> 0 end, name: VIA_...)

Registry.lookup(Registry.ViaTest, "agent")

VIA_no_value
#=> [{agent_pid, nil}]

VIA_value
#=> [{agent_pid, :hello}]
```

Example 2:

- registration of `self()` process with `Registry.register`
- duplicate registration
- pub/sub using dispatch/3, enabling partitions for better performance in concurrent environments

```
Registry
    .start_link(
        keys: :duplicate,
        name: Registry.MyRegistry,
        partitions: System.schedulers_online()
    )
    => {:ok, _}

    .lookup(Registry.MyRegistry, "hello")
    => []

    .register(Registry.MyRegistry, "hello", :world)
    => {:ok, _}

    .lookup(Registry.MyRegistry, "hello")
    => [{self(), :world}]

    .register(Registry.MyRegistry, "hello", :another)
    => {:ok, _}

    .lookup(Registry.MyRegistry, "hello"))
    => [{self(), :another}, {self(), :world}]

    .dispatch(
        Registry.MyRegistry,
        "hello",
        fn entries ->
            for {pid, _} <- entries,
            do: send(pid, {:broadcast, "world"})
        end
    )
    => :ok
```

Functions

```
child_spec([start_option()])
    :: Supervisor.child_spec()

start_link([start_option()])
    :: {:ok, pid} | {:error, term()}

start_option() ::
  {:keys, :unique | :duplicate}
  | {:name, registry}
  | {:partitions, pos_integer() \\ 1}
    # the number of partitions in the registry.
  | {:listeners, [atom()]}
    # list of named processes which are notified of
    # :register and :unregister events.
    # The registered process must be monitored by the
    # listener if the listener wants to be notified
    # if the registered process crashes.
  | {:meta, [{meta_key, meta_value}]}
    # :meta - a keyword list of metadata to be
    # attached to the registry.
```

```
:partitions Defaults to 1.
:listeners -
:meta - a keyword list of metadata to be attached to the registry.


register(registry, key, value)
    :: {:ok, pid}
        | {:error, {:already_registered, pid}}

unregister(registry(), key())
    :: :ok
unregister_match(registry, key, pattern, guards \\ [])
    :: :ok


lookup(registry, key)
    :: [{pid, value}]
match(registry, key, match_pattern, guards)
    :: [{pid, value}]
select(registry, spec)
    :: [term()]


dispatch(registry, key, mfa_or_fun, opts \\ [])
    :: :ok

count(registry)
    :: count
count_match (registry, key, pattern, guards \\ [])
    :: count

keys(registry, pid)
    :: [key]

update_value(registry, key, f)
    :: {new_value, old_value} | :error

meta(registry, key)
    :: {:ok, meta_value} | :error
put_meta(registry, key, value)
    :: :ok
```

---

6. Task

Execute function in a new process, monitored by, or linked to a caller.

It's better to spawn tasks with `Task.Supervisor`, instead of using Task.{start_link/1, async/3}

```
Task.{async/3, start_link/1} - link to caller
Task.{start/1} - no link to caller

Task.{async/3} - reply expected
Task.{start/1, start_link/1} - no reply expected
```

`Task.async/3` can be handed to:

- `Task.await/2` error after timeout
- `Task.yield/2` - can be invoked again after timeout

```
task = Task.async(fn -> do_some_work() end)
res = do_some_other_work()
res + Task.await(task)
```

Module-based

Limitation: can't be awaited on.

```
Supervisor.start_link(
    [
        {MyTask, arg}
    ],
    strategy: :one_for_one
```

```
)

defmodule MyTask do
  use Task # default restart: :temporary
           # (never restarted)

  def start_link(arg) do
    Task.start_link(__MODULE__, :run, [arg])
  end

  def run(arg) do
    # ...
  end
end
```

---

7. Task.Supervisor

Dynamically spawn and supervise tasks. Started with no children.

```
[your code] -- calls --> [supervisor] ---- spawns --> [task]

[your code]              [supervisor] <-- ancestor -- [task]
    ^                                                  |
    |--------------------- caller ---------------------|
```

```
# Short example

{:ok, pid} = Task.Supervisor.start_link()

task =
  Task.Supervisor.async(pid, fn ->
    # Do something
  end)

Task.await(task)
```

```
# As a part of Supervision tree

Supervisor.start_link([
  {Task.Supervisor, name: MyApp.TaskSupervisor}
], strategy: :one_for_one)

# no response:
Task.Supervisor.start_child(MyApp.TaskSupervisor, fn ->
  # Do something
end)

# await response:
Task.Supervisor.async(MyApp.TaskSupervisor, fn ->
  # Do something
end)
|> Task.await()
```

Functions

```

async(supervisor, f, options \\ [])
| async(supervisor, m, f, a, options \\ [])
:: Task.t()

async_nolink(supervisor, f, options \\ [])
| async_nolink(supervisor, m, f, a, options())
:: Task.t()

async options
    shutdown: timeout \\ 5000 | :brutal_kill

--------------------------------------------

async_stream(supervisor, enumerable, fun, options \\ [])
| async_stream(supervisor, enumerable, m, f, a, options \\ [])
```

```
:: Enumerable.t()

async_stream_nolink(supervisor, enumerable, fun, options \\ [])
| async_stream_nolink(supervisor, enumerable, m, f, a, options \\ [])
:: Enumerable.t()

async_stream options
    max_concurrency:
        number_of_concurrent_tasks \\ System.schedulers_online/0
    ordered:
        keep_results_order \\ true
    timeout:
        timeout_for_a_task \\ 5000
    on_timeout:
        :exit (default) # process that spawned the tasks exits
        | :kill_task    # task is killed, return value for task is {:exit, :timeout}
    shutdown:
        shutdown: timeout \\ 5000 | :brutal_kill

--------------------------------------------

children(supervisor)
    :: [pid, ...]

start_child(supervisor, f, options \\ [])
| start_child(supervisor, m, f, a, options \\ [])
    :: same_as_dynamic_supervisor

    start_child options
        restart: :temporary | :transient | :permanent
        shutdown: timeout \\ 5000 | :brutal_kill

terminate_child(supervisor, pid)
    :: :ok | {:error, :not_found}
```

More on Task.Supervisor.async

This function spawns a process that is linked to and monitored by the caller process. The linking part is important because it aborts the task if the parent process dies. It also guarantees the code before async/await has the same properties after you add the async call. For example, imagine you have this:

```
x = Task.async(&heavy_fun/0)
y = some_fun()
Task.await(x) + y
```

As before, if heavy_fun/0 fails, the whole computation will fail, including the parent process. If you don't want the task to fail then you must change the heavy_fun/0 code in the same way you would achieve it if you didn't have the async call. For example, to either return {:ok, val} | :error results or, in more extreme cases, by using try/rescue. In other words, an asynchronous task should be thought of as an extension of a process rather than a mechanism to isolate it from all errors.

If you don't want to link the caller to the task, then you must use a supervised task with Task.Supervisor and call Task.Supervisor.async_nolink/2.

More on Task.Supervisor.async_nolink

Use it if task failure is likely, and should be handled in some way.

In case of task failure, caller receives `:DOWN` message: `{:DOWN, ref, :process, _pid, _reason}`

```
defmodule MyApp.Server do
  use GenServer

  # ...

  def start_task do
    GenServer.call(__MODULE__, :start_task)
  end

  # In this case the task is already running, so we just return :ok.
  def handle_call(:start_task, _from, %{ref: ref} = state) when is_reference(ref) do
    {:reply, :ok, state}
  end

  # The task is not running yet, so let's start it.
  def handle_call(:start_task, _from, %{ref: nil} = state) do
    task =
```

```
      Task.Supervisor.async_nolink(MyApp.TaskSupervisor, fn ->
        # ...
      end)

    # We return :ok and the server will continue running
    {:reply, :ok, %{state | ref: task.ref}}
  end

  # The task completed successfully
  def handle_info({ref, answer}, %{ref: ref} = state) do
    # We don't care about the DOWN message now, so let's demonitor and flush it
    Process.demonitor(ref, [:flush])
    # Do something with the result and then return
    {:noreply, %{state | ref: nil}}
  end

  # The task failed
  def handle_info({:DOWN, ref, :process, _pid, _reason}, %{ref: ref} = state) do
    # Log and possibly restart the task...
    {:noreply, %{state | ref: nil}}
  end
end
```

`async_nolink` function requires the task supervisor to have :temporary as the :restart option (the default), as async_nolink/4 keeps a direct reference to the task which is lost if the task is restarted. TODO: clarify if `which is lost if the SUPERVISOR is restarted` is true, fix docs

More on Task.Supervisor.async_stream

Failure in Task brings caller down as well.

More on Task.Supervisor.async_stream_nolink

Failure in Task doesn't bring caller down, but results in {:exit, error} enumberable item result.

---

8. GenStage

Stages are used for:

- provide **back-pressure**
- leverage **concurrency**

Use `Task.async_stream` instead if both conditions are true:

- list to be processed is already in memory
- **back-pressure** is not needed

Concurrency in GenStage pipeline is achieved by having multiple Consumers for same Producer. Adding more Consumers allows to max out CPU and IO usage as necessary.

Producer implements **back-pressure** mechanism:

- has it's own **demand** (sum of Consumers' demands)
- emits to each Consumer `n` **events**, where `n <= consumer demand`. Dispatcher is used to send **events**.

Consumer <-> Producer is a **many-to-many** relationship.

Protocol details

1. Consumers send to Producers:

- start subscription
- cancel subscription
- send demand for a given subscription

2. Producers send to Consumers:

- cancel subscription (used as confirmation of clients cancellations, or to cancel upstream demand)
- send events to given subscription

Protocol visualization

```
 --- EVENTS (downstream) --->
  [A]  --->  [B]  ---> [C]
 <--- DEMAND (upstream) ---
```

Consumer max and min demand is often set on subscription:

- `:max_demand` - max amount of events that must be in flow
- `:min_demand` - minimum threshold to trigger for more demand.

`:max_demand` = 1000, `:min_demand` = 750. Possible Consumer actions:

- demand 1000 events
- receive 1000 events
- process at least 250 events
- ask for more events

Simple Example

```
defmodule A do
  use GenStage

  def start_link(number) do
    GenStage.start_link(A, number, name: __MODULE__)
  end

  def init(counter) do
    {:producer, counter}
  end

  def handle_demand(demand, counter) when demand > 0 do
    # If the counter is 3 and we ask for 2 items, we will
    # emit the items 3 and 4, and set the state to 5.
    events = Enum.to_list(counter..counter+demand-1)
    {:noreply, events, counter + demand}
  end
end
```

```
# ProducerConsumers act as a buffer.
# Getting the proper demand values is important:
# too small buffer may make the whole pipeline slower
# too big buffer may unnecessarily consume memory

defmodule B do
  use GenStage

  def start_link(multiplier) do
    GenStage.start_link(B, multiplier)
  end

  def init(multiplier) do
    # Manual subscription
    # {:producer_consumer, multiplier}
    # + GenStage.sync_subscribe(b, to: a)

    # Automatic subscription, relies on named Producer process.
    # Consumer crash will automatically re-subscribe it
    {
        :producer_consumer,
        multiplier,
        subscribe_to: [{A, max_demand: 10}]
    }
  end

  def handle_events(events, _from, multiplier) do
    events = Enum.map(events, & &1 * multiplier)
    {:noreply, events, multiplier}
  end
end
```

```
defmodule C do
  use GenStage

  def start_link(_opts) do
```

```elixir
    GenStage.start_link(C, :ok)
  end

  def init(:ok) do
    {:consumer, :the_state_does_not_matter}
  end

  def handle_events(events, _from, state) do
    # Wait for a second.
    Process.sleep(1000)

    # Inspect the events.
    IO.inspect(events)

    # We are a consumer, so we would never emit items.
    {:noreply, [], state}
  end
end
```

Subscription :manual vs :automatic

1. Manual subscription (no Supervision)

```elixir
subscription
{:ok, a} = A.start_link(0) # starting from zero
{:ok, b1} = B.start_link(2) # multiply by 2
{:ok, b2} = B.start_link(2)
{:ok, c} = C.start_link([]) # state does not matter

# Typically subscription goes from bottom to top:
GenStage.sync_subscribe(c, to: b)
GenStage.sync_subscribe(b1, to: a)
GenStage.sync_subscribe(b2, to: a)
```

2. Automatic subscription during Consumer's `init`

```elixir
children = [
  {A, 0},
  Supervisor.child_spec({B, [2]}, id: :c1),
  Supervisor.child_spec({B, [2]}, id: :c2)
  C
]

# Termination of Producer A causes termination of all Consumers.
#To avoid too many failures in a short interval:
# - use `:rest_for_one`
# - put Consumers under separate Supervisor
# - use ConsumerSupervisor (best approach)

Supervisor.start_link(children, strategy: :rest_for_one)
```

Buffering

1. Buffering events. => Buffer sent, but not delivered to consumers events until a consumer is available. To control buffer size, use `:buffer_size \\ 10_000` option in Producer `init` callback. (potential :buffer_size error)

2. Buffering demand. Consumers ask producers for events that are not yet available => buffer the consumer demand until events arrive (not a problem)

Example of manual control over events buffer in Producer

If events are sent without corresponding demand, they will wait in Producer's internal buffer. By default max size of internal buffer is 10_000, and `:buffer_size` error is thrown if it's exceeded.

Solution: We manage buffer manually, and don't let Producer dispatch events without corresponding demand. By managing queue and demand, we can control:

- how to behave when there are no consumers
- how to behave the queue grows too large
- ... Only case internal buffer may be used - if Consumer crashes without consuming all data.

```elixir
defmodule QueueBroadcaster do
  @moduledoc """
```

```
  If events, but no demand -> buffer events
  If demand, but no events -> buffer demand
  """

  use GenStage

  @doc "Starts the broadcaster."
  def start_link() do
    GenStage.start_link(__MODULE__, :ok, name: __MODULE__)
  end

  @doc "Sends an event and returns only after the event is dispatched."
  def sync_notify(event, timeout \\ 5000) do
    GenStage.call(__MODULE__, {:notify, event}, timeout)
  end

  ## Callbacks

  def init(:ok) do
    # {:queue.new, 0} - {events buffer, pending demand}
    {:producer, {:queue.new, 0}, dispatcher: GenStage.BroadcastDispatcher}
  end

  @doc """
  Incoming event:
  - add event to events queue in state
  - try to dispatch events
  """
  def handle_call({:notify, event}, from, {queue, pending_demand}) do
    queue = :queue.in({from, event}, queue)
    dispatch_events(queue, pending_demand, [])
  end

  @doc """
  Incoming demand:
  - increase pending demand in state
  - try to dispatch events
  """
  def handle_demand(incoming_demand, {queue, pending_demand}) do
    dispatch_events(queue, incoming_demand + pending_demand, [])
  end

  @doc """
  Pending demand = 0

  => Dispatch events (demand end reached)
  """
  defp dispatch_events(queue, 0, events) do
    {:noreply, Enum.reverse(events), {queue, 0}}
  end

  @doc """
  Pending demand > 0

  => Recursively build events
  if (not (empty? queue))
    :: dispatch_events(queue -- e, demand - 1, [e | events])

    => Dispatch events (queue end reached)
    if (empty? queue) # recursion stop condition
      :: {:noreply, events, new_state = {queue, demand}}

  """
  defp dispatch_events(queue, demand, events) do
    case :queue.out(queue) do
      {{:value, {from, event}}, queue} ->
        # TODO: understand why reply is here
        GenStage.reply(from, :ok)
        dispatch_events(queue, demand - 1, [event | events])
      {:empty, queue} ->
        {:noreply, Enum.reverse(events), {queue, demand}}
    end
  end
end
```

```
defmodule Printer do
  use GenStage

  @doc "Starts the consumer."
  def start_link() do
```

```
    GenStage.start_link(__MODULE__, :ok)
  end

  def init(:ok) do
    # Starts a permanent subscription to the broadcaster
    # which will automatically start requesting items.
    {:consumer, :ok, subscribe_to: [QueueBroadcaster]}
  end

  def handle_events(events, _from, state) do
    for event <- events do
      IO.inspect {self(), event}
    end
    {:noreply, [], state}
  end
end
```

```
# Demo

# Start the producer
QueueBroadcaster.start_link()

# Start multiple Printers (each sends it's demand to QueueBroadcaster)
Printer.start_link()
Printer.start_link()
Printer.start_link()

QueueBroadcaster.sync_notify(:hello_world)

# With [buffered demand] and [not empty queue],
# => QueueBroadcaster dispatches event to each Printer
```

Asynchronous work and handle_subscribe

Consumer and ProducerConsumer first `handle_events/3`, and then send **demand** upstream. This means demand is sent synchronously by default. There are two options to send demand asynchronously:

1. Manual:

- implement the `handle_subscribe/4` callback and return `{:manual, state}` instead of the default `{:automatic, state}`,
- use `GenStage.ask/3` to send demand upstream when necessary. `:max_demand` and `:min_demand` should be manually respected.

2. Using ConsumerSupervisor: ConsumerSupervisor module processes events asynchronously by starting a process for each event and this is achieved by manually sending demand to producers.

Back-pressure

`handle_subscribe/4` + `manual` is also useful for implementing custom **back-pressure** mechanisms.

Default back-pressure mechanism

When data is sent between stages, it is done by a message protocol that provides back-pressure. - consumer subscribes to the producer. Each subscription has a unique reference.

- once subscribed, consumer may ask the producer for messages for the given subscription. A consumer must never receive more data than it has asked from a Producer.

A producer may have multiple consumers, where the demand and events are managed and delivered according to a GenStage.Dispatcher implementation. A consumer may have multiple producers, where each demand is managed individually (on a per-subscription basis). See example below:

Example of custom back-pressure mechanism in Consumer

Implement a consumer that is allowed to process a limited number of events per time interval:

```
defmodule RateLimiter do
  @moduledoc """
  The trick is - Consumer manages Producers' pending demand,
  instead of Producer doing this.
  There are 2 main pieces of puzzle:

  1. ask_and_schedule calls itself recursively with an interval:
    - GenStage.ask(from, pending)
      -> trigger handle_events
```

```elixir
    - resets pending to 0, which results in possible GenStage.ask(from, 0) repeated calls,
      but it's harmless, as handle_demand(0) is ignored by Producers.

  2. handle_events (triggered by GenStage.ask(from, pending))
    - gets new events
    - processes them
    - sets pending to length(events)
      -> thanks to this ask_and_schedule will repeat 1-2 cycle
  """
  use GenStage

  def init(_) do
    # Our state will keep all producers and their pending demand
    {:consumer, %{}}
  end

  @doc """
  from() :: {pid(), subscription_tag()}
  The term that identifies a subscription associated with the corresponding
  producer/consumer.
  """
  def handle_subscribe(:producer, opts, from, _state = producers) do
    # We will only allow max_demand events every 5000 milliseconds
    pending = opts[:max_demand] || 1000
    interval = opts[:interval] || 5000

    # Register the producer in the state
    producers = Map.put(producers, from, {pending, interval})
    # Ask for the pending events and schedule the next time around
    producers = ask_and_schedule(producers, from)

    # Returns manual as we want control over the demand
    {:manual, producers}
  end

  def handle_cancel(_, from, producers) do
    # Remove the producers from the map on unsubscribe
    {:noreply, [], Map.delete(producers, from)}
  end

  def handle_events(events, from, producers) do
    # Bump the amount of pending events for the given producer
    producers = Map.update!(
      producers,
      from,
      fn {pending, interval} ->
        {pending + length(events), interval}
      end
    )

    # Consume the events by printing them.
    Process.sleep(:rand.uniform(10_000)) # simulate actual work
    IO.inspect(events)

    # A producer_consumer would return the processed events here.
    {:noreply, [], producers}
  end

  def handle_info({:ask, from}, producers) do
    # This callback is invoked by the Process.send_after/3 message below.
    {:noreply, [], ask_and_schedule(producers, from)}
  end

  defp ask_and_schedule(producers, from) do
    case producers do
      %{^from => {pending, interval}} ->
        # Ask for any pending events
        GenStage.ask(from, pending)
        # And let's check again after interval
        Process.send_after(self(), {:ask, from}, interval)
        # Finally, reset pending events to 0
        Map.put(producers, from, {0, interval})
      %{} ->
        producers
    end
  end
end
```

```elixir
{:ok, a} = GenStage.start_link(A, 0)
{:ok, b} = GenStage.start_link(RateLimiter, :ok)
```

```
# Ask for 10 items every 2 seconds
GenStage.sync_subscribe(b, to: a, max_demand: 10, interval: 2000)
```

Callbacks

Define/override child_spec:

```
use GenStage,
  restart: :transient,
  shutdown: 10_000
```

Required callbacks: `init/1` - choice between `:producer`, `:consumer`, `:producer_consumer` stages `handle_demand/2` - `:producer` stage `handle_events/3` - `:producer_consumer`, `:consumer` stages

init

```
init(args) ::
  {:producer, state}
  | {:producer, state, [producer_option()]}
  | {:producer_consumer, state}
  | {:producer_consumer, state, [producer_consumer_option()]}
  | {:consumer, state}
  | {:consumer, state, [consumer_option()]}
  | :ignore
  | {:stop, reason :: any()}
```

```
:init/1 options

:producer
  demand:
    :forward (default)
    # forward demand to the `handle_demand/2`

    :accumulate
    # accumulate demand until set to :forward via demand/2

# :accumulate is useful as a synchronization mechanism,
# where the demand is accumulated until all consumers are subscribed.

:producer and :producer_consumer
  :buffer_size
    10_000 (:producer default)
    :infinity (:producer_consumer default)
    # The size of the buffer to store events without demand.

  :buffer_keep \\ :last
    # whether the :first or :last entries should stay in buffer
    # if :buffer_size is exceeded

  :dispatcher \\ GenStage.DemandDispatch
    DispatcherModule
    | {DispatcherModule, options}
    # the dispatcher responsible for handling demand

:consumer and :producer_consumer
  :subscribe_to
    [ProducerModule]
    | [{ProducerModule, [subscription_options()]}]
```

handle_call

```
handle_call(request, from, state) ::
  {:reply, reply, [event], new_state}
  | {:noreply, [event], new_state}
  | {:stop, reason, reply, new_state}
  | {:stop, reason, new_state}

  {:reply, reply, [events], new_state}
    -> dispatch events (or buffer)
    -> send the response reply to caller
    -> continue loop with new state
  {:noreply, [event], state}
    -> dispatch events (or buffer)
```

```
    -> continue loop with new state
    -> manually send response with `GenStage.reply`
  {:stop, reason, new_state}
    -> stop the loop
    -> terminate is called with a reason and new_state
```

handle_cast, handle_info, handle_demand, handle_events

```
handle_cast(request, state) ::
handle_info(message, state) ::

# required for :producer stage
handle_demand(demand :: pos_integer(), state) ::

# required for :producer_consumer and :consumer stages
handle_events(events :: [event], from(), state) ::

  {:noreply, [event], new_state}
  | {:stop, reason, new_state}

  # {:noreply, [event], new_state}
  # -> dispatch or buffer events
  # -> continue loop with new state
```

handle_subscribe

Invoked in both producers and consumers when consumer subscribes to producer.

```
handle_subscribe(
  producer_or_consumer :: :producer | :consumer,
  subscription_options(),
  from(),
  state :: term()
) ::
  {:automatic | :manual, new_state}
  | {:stop, reason, new_state}

  # {:automatic, new_state} (default)
  # demand is sent automatically to producer

  # {:manual, new_state}
  # supported only by Consumers
  # demand must be sent via ask(from(), demand)
```

handle_cancel

Invoked when a consumer is no longer subscribed to a producer.

```
handle_cancel(
  cancellation_reason :: {
    :cancel # cancellation from GenStage.cancel/2 call
    | :down, # cancellation from an EXIT
    reason
  },
  from(),
  state
) ::
  {:noreply, [], new_state} (default)
  | handle_cast_returns
```

Types

stage().t

Stage registered name or pid

```
stage() ::
  pid()
  | atom()
  | {:global, term()}
  | {:via, module(), term()}
  | {atom(), node()}
```

from().t

Producer subscription identifier

```
from() ::
  {pid(), subscription_tag()}
# Can be obtained in handle_subscribe/4, and stored in stage's state.
```

subscription_options().t

Option used by the subscribe* functions

```
subscription_options() ::
  min_demand:
  max_demand:

  cancel:
    :permanent (default)
    # consumer exits if the producer cancels subscription or exits
    :transient
    # consumer exits if reason not in
    # [:normal, :shutdown, or {:shutdown, reason}]
    :temporary
    # consumer never exits

  _: # any other option
    # Example for GenStage.BroadcastDispatcher:
    GenStage.sync_subscribe(
        consumer,
        to: producer,
        selector:
          fn %{key: key} ->
            String.starts_with?(key, "foo-")
          end)
    )
```

Functions

```
# Same as GenServer
start_link(module, args, options \\ [])
stop(stage, reason \\ :normal, timeout \\ :infinity)
call(stage, request, timeout \\ 5000)
cast(stage, request)
reply(client, reply)
```

subscription()

Subscribe consumer to the given producer. Usually done from `init`, by use of `subscribe_to` option. As a result of subscription, `subscription_tag()` is passed to consumer's `handle_subscribe/4` callback.

`resubscribe` cancels subscription with given reason, can be used to update `subscription_options`.

```
subscribe_args ::
  stage,
  subscription_options()

sync_returns ::
  {:ok, subscription_tag()}
  | {:error, :not_a_consumer}
  | {:error, {:bad_opts, String.t()}}

async_subscribe(...subscribe_args)
  :: :ok
# done automatically from `init` with `subscribe_to` option

sync_subscribe(...subscribe_args, timeout)
  :: sync_returns
# -> returns before consumer:handle_subscribe/4 is called

resubscribe_args ::
  stage,
  subscription_tag(),
  reason,
  subscription_options
```

```
async_resubscribe(...resubscribe_args)
  :: :ok

sync_resubscribe(...resubscribe_args, timeout())
  :: sync_returns
```

demand()

```
demand(stage) :: :forward | :accumulate
# Returns producer's demand

demand(stage, :forward | :accumulate) :: :ok
# Sets producer's demand
```

ask()

Asks the given demand to the producer. Same args/return as `Process.send(dest, msg, opts)`.

```
ask(from(), demand, opts \\ []) :: :ok
# Can only be used if `handle_subscribe/4` returns `:manual`
# if demand = 0, does nothing
```

cancel()

```
Cancels the given subscription on the producer.
Same args/return as `Process.send(dest, msg, opts)`.

cancel(from(), reason, opts \\ []) :: :ok

# Consumer will react according to the :cancel option,
#  given when subscribing, for example:
reason(:shutdown) + consumer(:permanent) = crash
```

info()

```
# info message, that is delivered to handle_info:
# - for consumers: immediately
# - for producers: queued after all currently buffered events

async_info(stage, msg) :: :ok
# immediate return

sync_info(stage, msg, timeout \\ 5000) :: :ok
# return :ok after message has been queued
```

from_enumerable() (higher level function)

Starts a producer stage (linked to current process) from a stream (most common case) or other enumerable. Producer will take items from the enumerable when there is demand.

The enumerable is consumed in batches, retrieving `max_demand` items the first time and then `max_demand - min_demand` the next times. Therefore, for streams that cannot produce items that fast, it is recommended to pass a lower `:max_demand` option value.

Enumerable should: -> block until the current batch is fully filled -> return batch or terminate

When Enumerable finishes or halts, stage exits with normal reason, and consumer (**Possible duplicate info**):

```
? consumer(cancel: :permanent)  (default)
  -> consumer exits(:normal)
? consumer(cancel: :transient)
  -> consumer exits(reason), only for error reasons
? consumer(cancel: :temporary)
  -> consumer never exits
```

Resulting Producer can be used with GenStage or Flow (integrated by use of `Flow.from_stages/2`).

```
GenServer.on_start() ::
  {:ok, pid()}
  | :ignore
```

```
  | {:error,
    {:already_started, pid()}
    | term()
  }

producer_opts() ::
  link: \\ true,
  dispatcher: \\ GenStage.DemandDispatch
    DispatcherModule
    | {DispatcherModule, opts},
  demand: :forward | :accumulate
  ...all_start_link_options

from_enumerable(Enumerable.t(), producer_opts())
  :: GenServer.on_start()
```

stream() (higher level function)

Creates a stream that subscribes to the given producers and emits the appropriate messages.

```
producers() :: [
  producer
  | {producer | subscription_options()}
]

opts() ::
  demand: \\ :forward
  producers: \\ producers()
  # If some of producers() are :producer_consumers,
  # pass here only actual producers.

stream(
  producers(),
  opts()
) :: Enumerable.t()

=> GenStage.stream([{producer, max_demand: 100}])
```

If the producer process exits, the stream will exit with the same reason. To halt stream instead, set the cancel option to either `:transient` or `:temporary` as described in `subscription_options().t`:

```
GenStage.stream([{
    producer,
    max_demand: 100,
    cancel: :transient
}])
```

Once all producers are subscribed to, their demand is automatically set to :forward mode.

`GenStage.stream/1` will "hijack" the inbox of the process enumerating the stream to subscribe and receive messages from producers.

---

9. GenStage dispatchers

GenStage.Dispatcher behaviour

This module defines the behaviour used by `:producer` and `:producer_consumer` to dispatch events.

GenServer has three built-in implementations: `DemandDispatcher`, `BroadcastDispatcher`, `PartitionDispatcher`.

Implementation is chosen in Producer:

```
def init(:ok) do
    {
        :producer,
        state,
        dispatcher:
            GenStage.BroadcastDispatcher
            | {GenStage.BroadcastDispatcher, dispatcher_options()}
    }
end
```

Callbacks:

```
init(opts)
subscribe(opts, from, state)
ask(demand, from, state)
dispatch(events, length, state)
cancel(from, state)
info(msg, state)
```

GenStage.DemandDispatcher

A dispatcher that sends batches to the highest demand.

This is the default dispatcher used by GenStage. In order to avoid greedy consumers, it is recommended that all consumers have exactly the same maximum demand.

GenStage.BroadcastDispatcher

A dispatcher that accumulates demand from all consumers before broadcasting events to all of them. It guarantees that events are dispatched to all consumers without exceeding the demand of any given consumer.

Example with BroadcastDispatcher-specific `:selector` option:

```
# Inside consumer's init/1
{
    :consumer,
    :ok,
    subscribe_to:
        [{
            producer,
            # Consumers receive events, filtered by selector
            selector: fn %{key: key} ->
                String.starts_with?(key, "foo-")
            end
        }]
}
end
# or
GenStage.sync_subscribe(
    consumer,
    to: producer,
    selector:
        fn %{key: key} ->
            String.starts_with?(key, "foo-")
        end
)
```

GenStage.PartitionDispatcher

A dispatcher that sends events according to partitions.

Keep in mind that, if partitions are not evenly distributed, a backed-up partition will slow all other ones.

```
dispatcher_options() ::
    partitions :: integer or list
    # 4 = partitions with keys 0, 1, 2, 3
    # 0..3 = same

    hash :: (event) => {event, partition_key}
    # function to hash event
    # example:
```

When subscribing to a GenStage with a partition dispatcher the `:partition` option is required.

```
# Choose dispatcher inside producer's init/1
{
    :producer,
    state,
    dispatcher: {
        GenStage.PartitionDispatcher,
        partitions: 0..3,
        hash: fn event ->
            {
```

```
                event,
                :erlang.phash2(
                    event,
                    Enum.count(partitions)
                )
            }
        end
    }
}

# Inside consumer's init/1
{
    :consumer,
    :ok,
    subscribe_to: [{producer, partition: 0}]
}
# or
GenStage.sync_subscribe(
    consumer,
    to: producer,
    partition: 0
)
```

---

10. ConsumerSupervisor behaviour

A supervisor that starts children as events flow in Can be used as the consumer in a GenStage pipeline.

Can be attached to a producer by returning `:subscribe_to` from `init/1` or explicitly with `GenStage.sync_subscribe/3` and `GenStage.async_subscribe/2`.

Once subscribed, the supervisor will:

- ask the producer for `:max_demand events`
- start child processes per event as events arrive (event is appended to the arguments in the child specification)
- as child processes terminate, the supervisor will accumulate demand and request more events after `:min_demand` is reached

ConsumerSupervisor is similar to a pool, except a child process is started per event. `:min_demand` < `amount of concurrent children per producer` < `:max_demand`.

Example

```
defmodule Consumer do
  use ConsumerSupervisor

  def start_link(arg) do
    ConsumerSupervisor.start_link(__MODULE__, arg)
  end

  def init(_arg) do
    # Note: By default child.restart = :permanent
    # ConsumerSupervisor supports only :termporary or :transient
    children = [%{
        id: Printer,
        start: {Printer, :start_link, []},
        restart: :transient
    }]
    opts = [
        strategy: :one_for_one,
        subscribe_to: [{Producer, max_demand: 50}]
    ]
    ConsumerSupervisor.init(children, opts)
  end
end

defmodule Printer do
  def start_link(event) do
    # Note: this function must:
    # - return the format of {:ok, pid}
    # - like all children started by a Supervisor,
    #   the process must be linked back to the supervisor
    # Task.start_link/1 satisfies these requirements
    Task.start_link(fn ->
      IO.inspect({self(), event})
    end)
```

```
  end
end
```

Callbacks

```
init_options() :: [
    max_restarts \\ 3,
    max_seconds \\ 5,
    subscribe_to:
        [Producer]
        | [{Producer, max_demand: 20, min_demand: 10}]
        # [{Producer, subscription_options().t}]
]

init(args) ::
  {:ok, [:supervisor.child_spec()], init_options()}
  | :ignore
```

Functions

```
# Supervisor
init(children, init_options()) # For module based supervisor
start_link(mod, args) # For module based supervisor
start_link(children, init_options()) # For in-place supervisor

# Children
start_child(supervisor, child_args)
terminate_child(supervisor, pid)
count_children(supervisor)
which_children(supervisor)
```

---

11. Application configuration

```
# config.exs
import Config

config :some_app,
  key1: "value1",
  key2: "value2"

import_config "#{Mix.env()}.exs"

# Usage
"value1" = Application.fetch_env!(:some_app, :key1)
```

Config.Provider behaviour

Specifies a provider API that loads configuration during boot. Config providers are typically used during releases to load external configuration while the system boots. Results of the providers are written to the file system.

```
defmodule JSONConfigProvider do
  @behaviour Config.Provider

  # Let's pass the path to the JSON file as config
  def init(path) when is_binary(path), do: path

  def load(config, path) do
    # We need to start any app we may depend on.
    {:ok, _} = Application.ensure_all_started(:jason)

    json = path |> File.read!() |> Jason.decode!()

    Config.Reader.merge(
      config,
      my_app: [
        some_value: json["my_app_some_value"],
        another_value: json["my_app_another_value"],
      ]
    )
  end
```

```
end

# mix.exs -> :releases
releases: [
  demo: [
    # ...,
    config_providers: [{JSONConfigProvider, "/etc/config.json"}]
  ]
]
```

Functions:

```
resolve_config_path!(config_path()) :: path
validate_config_path!(config_path()) :: :ok
    config_path() ::
        {:system, env_value_key, path}
        | path

# Example:
System.put_env("BLAH", "blah")
resolve_config_path!({:system, "BLAH", "/rest"})
# => "blah/rest"
```

Callbacks:

```
init(term()) :: state()
```

Invoked when initializing a config provider.

A config provider is typically initialized on the machine where the system is assembled and not on the target machine. The `init/1` callback is useful to verify the arguments given to the provider and prepare the state that will be given to load/2.

Furthermore, because the state returned by `init/1` can be written to text-based config files, it should be restricted only to simple data types, such as integers, strings, atoms, tuples, maps, and lists. Entries such as PIDs, references, and functions cannot be serialized.

```
load(config(), state()) :: config()
```

Loads configuration (typically during system boot).

It receives the current config and the state returned by init/1. Then you typically read the extra configuration from an external source and merge it into the received config. Merging should be done with `Config.Reader.merge/2`, as it performs deep merge. It should return the updated config.

Note that `load/2` is typically invoked very early in the boot process, therefore if you need to use an application in the provider, it is your responsibility to start it.

Config.Reader

API for reading config files defined with Config. Can also be used as a Config.Provider:

```
# mix.exs
releases: [
  demo: [
    # ...,
    config_providers:
        [{Config.Reader, "/etc/config.exs"}]
        | [{Config.Reader, {:system, "RELEASE_ROOT", "/config.exs"}}]
  ]
]
```

mix release By default Mix releases supports runtime configuration via a `config/releases.exs`. If a `config/releases.exs` exists in your application, it is automatically copied inside the release and automatically set as a config provider.

Functions:

```
merge(config1:keyword(), config2:keyword())

read!(file, imported_paths \\ [])
# Reads configuration file.
# Example: mix.exs:releases config in a separate file.
releases: Config.Reader.read!("rel/releases.exs")
```

```
read_imports!(file, imported_paths \\ [])
# Reads configuration file and it's imports.
```

---

12. Mix release

Assembles a self-contained release for the current project. Benefits:

- Code preloading
- Configuration and customization of system and VM
- Self-contained, includes ERTS and stripped versions of Erlang and Elixir
- Scripts to start, restart, connect to the running system remotely, execute RPC calls, run as daemon

```
MIX_ENV=prod mix release # relies  on default_release: NAME
MIX_ENV=prod mix release NAME
```

Build/deploy environment must have same OS distribution and versions.

Release configuration

By default `:applications` includes the current application and all applications the current application depends on, recursively.

```
def project do
  [
    releases: [
      demo: [
        include_executables_for: [:unix],
        applications: [
            runtime_tools: application_option()
        ],

        config_providers: list \\ [],
        strip_beams: bool \\ true,
        path: path \\ "_build/MIX_ENV/rel/RELEASE_NAME",
        version: version \\ current_app_version,
        include_erts: bool \\ true,
        include_executables_for: [:unix | :windows] \\ []
        overlays: overlays(),
        steps: steps()
      ],

      ...
    ]
  ]

  # Release config can be passed a function
  | releases: [
    demo: fn ->
      [version: @version <> "+" <> git_ref()]
    end
  ]
end

application_option() \\ :permanent
  :permanent
  # application is started and the node shuts down
  # if the application terminates, regardless of reason
  :transient
  # application is started and the node shuts down
  # if the application terminates abnormally
  :temporary
  # application is started and the node does not shut down
  # if the application terminates
  :load
  # the application is only loaded
  :none
  # the application is part of the release but it is neither loaded nor started

overlays() \\ "rel/overlays"
# Directory for extra files to be copied into root folder of release.

steps() \\ [:assemble]
# Dynamically build Release struct:
```

```
releases: [
  demo: [
    steps: [&set_configs/1, :assemble, &copy_extra_files/1]
  ]
]
```

Application configuration

Releases provides two mechanisms for configuring OTP applications: build-time and runtime.

App configuration: build-time

```
# config/config.exs, config/prod.exs...
import Config
config :my_app,
  :secret_key,
  System.fetch_env!("MY_APP_SECRET_KEY")
```

- evaluated during code compilation or release assembly

App configuration: run-time

1. Using runtime configuration file (`releases.exs` by default)

```
# `config/releases.exs`
import Config
config :my_app,
  :secret_key,
  System.fetch_env!("MY_APP_SECRET_KEY")
```

- evaluated early during release start
- writes computed configuration to `RELEASE_TMP` (by default `$RELEASE_ROOT/tmp`) folder
- restarts release

Rules for runtime configuration file:

- It MUST import Config at the top instead of the deprecated use Mix.Config
- It MUST NOT import any other configuration file via import_config
- It MUST NOT access Mix in any way, as Mix is a build tool and it not available inside releases

2. Using config providers

- loads configuration during release start, using custom mechanism, for example read JSON file, or access a vault
- writes computed configuration to `RELEASE_TMP` (by default `$RELEASE_ROOT/tmp`) folder
- restarts release

```
# mix.exs
releases: [
  demo: [
    # ...,
    config_providers: [{JSONConfigProvider, "/etc/config.json"}]
  ]
]
```

VM and environment configuration

```
mix release.init

* creating rel/vm.args.eex
* creating rel/env.sh.eex
* creating rel/env.bat.eex

# In those files following variables can be used:
RELEASE_NAME,
RELEASE_COMMAND (start, remote, eval...),
RELEASE_VSN,
RELEASE_ROOT
```

Interacting with a release

```
# Start system
_build/prod/rel/my_app/bin/my_app start

# Stop system (vm, app and supervision trees in opposite to starting order)
Send SIGINT/SIGTERM to OS process
| bin/RELEASE_NAME stop
```

```
# One-off commands

defmodule MyApp.ReleaseTasks do
  def eval_purge_stale_data() do
    Application.ensure_all_started(:my_app)

    # Code that purges stale data
  end
end

# >
bin/RELEASE_NAME eval "MyApp.ReleaseTasks.eval_purge_stale_data()"
bin/RELEASE_NAME rpc "IO.puts(:hello)"
```

```
# All commands (`bin/RELEASE_NAME` help)

start          Starts the system
start_iex      Starts the system with IEx attached
daemon         Starts the system as a daemon
daemon_iex     Starts the system as a daemon with IEx attached
eval "EXPR"    Executes the given expression on a new, non-booted system
rpc "EXPR"     Executes the given expression remotely on the running system
remote         Connects to the running system via a remote shell
restart        Restarts the running system via a remote command
stop           Stops the running system via a remote command
pid            Prints the operating system PID
               of the running system via a remote command
version        Prints the release name and version to be booted
```