

# 1. Process

---

## Functions

```
# Spawn/exit/hybernate
spawn(fun, opts) | spawn(m, f, a, opts)
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) # calls :erlang.process_info/1
list() :: # list of all running PIDs

# Process dictionary
get()
get(key, default \\ nil)
get_keys()
get_keys(value)
put(key, value)
delete(key)

# Process.info(pid)
[
  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: []
]
```

```
# Spawn options

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()}
```

## 2. GenServer

---

used for:

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

### GenServer.start\_link

```
GenServer
  start_link(
    _,
    options:
      name:
        atom
        | {:global, term}
        | {:via, module, name}
      timeout:
        msec \ \ :infinity
  )
```

### GenServer.call

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

### GenServer.cast

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

### GenServer.reply

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

Can be used instead of {:reply, \_, \_} inside `handle_call` .

```
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
```

GenServer.reply can even be done from different process.

### GenServer.stop

Synchronously stops server with given reason Normal reasons (no error logged): :normal | :shutdown | {:shutdown, term}

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

### GenServer timeout mechanism:

:timeout message will be sent if no handle\_\* is invoked in timeout msecs.

Setup: add timeout option to:

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

GenServer
  handle_info(:timeout, _)
```

Because a message may arrive before the timeout is set, even a timeout of 0 milliseconds is not guaranteed to execute. To take another action immediately and unconditionally, use a :continue instruction + handle\_continue callback.

### GenServer.handle\_continue

Asynchronous initialization can cause a race condition: <https://medium.com/@tylerpachal/introduction-to-handle-continue-in-elixir-and-when-to-use-it-53ba5519cc17>

```
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.

## Callbacks

:reply, :noreply, :stop, :continue are **instructions** from() = {pid(), tag :: term()}

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}
```

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()
```

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)

## Process monitoring

```
Process
  monitor

GenServer
  handle_info({:DOWN})
```

2nd parameter is timeout

```
: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
```

## :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}]
  ]
]}
```

## Process statistics using :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

Supervisor = Child specification + Supervision options

#### Child specification

```
%{
  id:
    term() \\ __MODULE__
  start:
    {m, f, a}
  restart:
    :permanent (always restart)
    | :temporary (never restart)
    | :transient
  shutdown:
    :brutal_kill
    | timeout
    | :infinity
}
| {Stack, [:hello]}
| Stack
```

**Restart transient** No restart if exit reason: `:normal`, `:shutdown`, `{:shutdown, term}` Propagate to linked processes if exit reason not `:normal`

**Default shutdown values** `:infinity` for Supervisors `5000` for Workers

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)` .

#### Override child\_spec outside implementation module

```
Supervisor.child_spec(
  {Stack, [:hello]},
  id: 1,
  shutdown: 10_000
)
```

#### Supervision options

Used for top-level or module-based Supervisors:

```
Supervisor.start_link(children, options)
Supervisor.init(children, options)
```

```
strategy:
  :one_for_one
  | :rest_for_one
  | :one_for_all
max_restarts:
  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
```

Functions

- child\_spec/2
- count\_children/1
- delete\_child/2
- init/2
- restart\_child/2
- start\_child/2
- start\_link/2
- start\_link/3
- stop/3
- terminate\_child/2
- which\_children/1

Supervisor

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

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()
  }
]
```

Supervisor children

```
start_child(
  supervisor(),
  :supervisor.child_spec()
```

```

        | {module(), term()}
        | module()
    )
:: { :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

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

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
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} enumerable item result.

## 8. 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.
```

## 9. 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
```