SPE

Programming Scalable Systems

Lucía Liu Wang
Inés Pérez Rincón
Carlos Velázquez Arranz

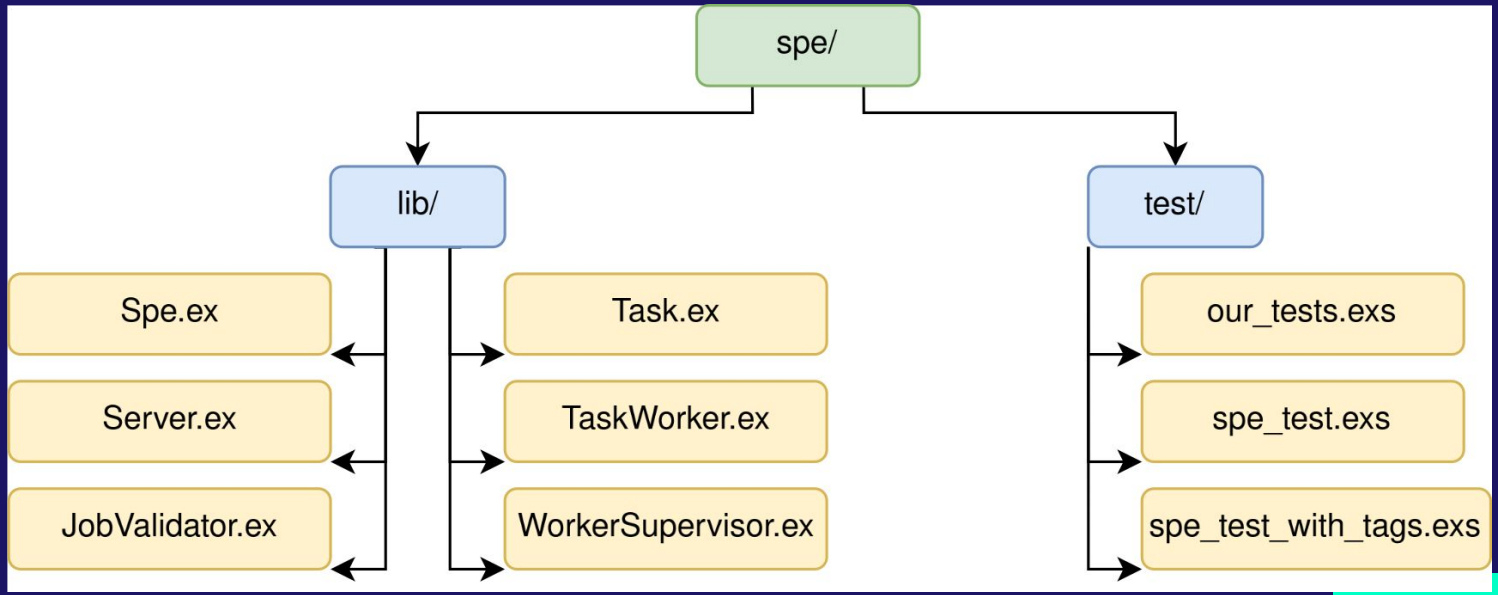
TABLE OF CONTENTS

WHAT WE'VE BUILT

- Overview of SPE: a fault-tolerant job processing engine, that executes complex and dependant tasks concurrently, ensuring real-time result reporting
- Concurrency model with dependency management and parallel task control.
- Based entirely on Elixir and PubSub messaging, making also use of GenServer and Supervisor components.

PROJECT OVERVIEW

This diagram shows the core folder structure of the **spe/** project, focusing on the most relevant files for understanding the system architecture and test setup.



IMPLEMENTATION FILES

FILE	SPECIFICATIONS
Spe.ex	Entry point
Server.ex	Manages jobs and execution
JobValidator.ex	Validates jobs and creates DAGs
Task.ex	Validates single tasks
TaskWorker.ex	Executes a single task
WorkerSupervisor.ex	Supervises task processes

IMPLEMENTATION DETAILS I

Spe: Public API to submit and start jobs

- Starts and supervises the system
- Acts as the main controller to manage and coordinate tasks
- Ensures fault-tolerance through supervision and isolated processes

```elixir
defmodule SPE do
  use Supervisor

  def start_link(options) do···
  end

  def submit_job(job_description) do···
  end

  def start_job(job_id) do···
  end

  def init(options) do···
  end
end
```

IMPLEMENTATION DETAILS II

Server: Core scheduler and coordinator

- Job Submission, validates and stores jobs
- Starts jobs, Launches tasks whose dependencies are all resolved
- Job completion detection
- Broadcasts task status and final job result

```
defmodule SPE.Server do
  defstruct [ ···
  ]

  def start_link(opts) do ···
  end

  def init(state) do ···
  end

  # Callback
  def handle_call({:submit_job, job_description}, _from, state) do ···
  end

  def handle_call({:start_job, job_id}, _from, state) do ···
  end

  def handle_info({:task_completed, job_id, task_name, result}, state) do ···
  end

  defp remove_task(pending_tasks, job_id, task) do ···
  end

  defp schedule_tasks(state) do ···
  end

  defp check_job_completion(job_id, state) do ···
  end

  defp compile_job_results(job_id, state) do ···
  end

  defp build_task_map(tasks) do ···
```

IMPLEMENTATION DETAILS III

JobValidator: Validates the syntactic correctness of a job description before processing

- Validates the basic job structure
- Validates each task individually
- Validates task dependencies ensuring they form a Directed Acyclic Graph (DAG)

```elixir
defmodule SPE.JobValidator do
  @spec validate_job(any()) :: any()
  def validate_job(job_description) do···
  end

  defp validate_structure(%{"name" => name, "tasks" => tasks}) ···
  end

  defp validate_structure(%{"name" => name}) when not is_binary(name) do···
  end

  defp validate_structure(%{"name" => ""}) do···
  end

  defp validate_structure(%{}) do···
  end

  defp validate_structure(_) do
    {:error, :invalid_job}
  end

  defp validate_tasks([]) do···
  end
```

IMPLEMENTATION DETAILS IV

Task: Normalizes and validates task maps

- Returns normalized task map or an error tuple if validation fails

```elixir
defmodule SPE.Task do
@spec normalize_task(any()) :: {:error, :invalid_task_enables | :invalid_task_exec | :invalid_task_name | :invalid_task_timeout | <<_::144>>} | %{
  def normalize_task(task) when is_map(task) do ···
  end

  def normalize_task(_), do: {:error, "Task must be a map"}

  defp validate_name(name) when is_binary(name) and name != "", do: {:ok, name}
  defp validate_name(_), do: {:error, :invalid_task_name}

  defp validate_exec(fun) when is_function(fun, 1), do: {:ok, fun}
  defp validate_exec(_), do: {:error, :invalid_task_exec}

  defp validate_enables(enables) when is_list(enables), do: {:ok, enables}
  defp validate_enables(_), do: {:error, :invalid_task_enables}

  defp validate_timeout(:infinity), do: {:ok, :infinity}
  defp validate_timeout(timeout) when is_integer(timeout) and timeout > 0, do: {:ok, timeout}
  defp validate_timeout(_), do: {:error, :invalid_task_timeout}
end
```

IMPLEMENTATION DETAILS V

TaskWorker: Executes a single task asynchronously within a GenServer

- Returns results wrapped as {:result, value} or failure tuples
- Result reporting to SPE Server

```elixir
defmodule SPE.TaskWorker do
  use GenServer

  @spec start_link(any()) :: :ignore | {:error, any()} | {:ok, pid()}
  def start_link(options) do···
  end

  @spec init(%{:dependencies => any(), :job_id => any(), :server_pid => any(), :task => any(), optional(any()) => any()}) :: {:ok, %{job_id: any(), server_pid:
  def init(%{server_pid: server_pid, job_id: job_id, task: task, dependencies: dependencies}) do···
  end

  def handle_info({ref, result}, state) when ref == state.task_ref.ref do···
  end

  def handle_info({:DOWN, ref, :process, _pid, reason}, state) when ref == state.task_ref.ref do···
  end

  defp execute_task(task, dependencies) do···
  end

  defp report_result(server_pid, job_id, task_name, result) do···
  end

  def terminate(_reason, state) do···
  end
end
```

IMPLEMENTATION DETAILS VI

WorkerSupervisor: Supervises task-related worker processes

- Dynamically manages individual task workers
- :one_for_one supervision strategy

```elixir
defmodule SPE.WorkerSupervisor do
  use Supervisor

  @spec start_link(any()) :: :ignore | {:error, any()} | {:ok, pid()}
  def start_link(_opts) do ···
  end

@spec init(:ok) :: {:ok, {%{auto_shutdown: :all_significant | :any_significant | :never, intensity: non_neg_integ
  def init(:ok) do ···
  end

  @spec start_task(any(), any(), any(), any()) :: :ignore | {:error, any()} | {:ok, pid()} | {:ok, pid(), any()}
  def start_task(server_pid,job_id, task, dependencies) do ···
  end
end
```

TESTING

Two test files

- our_tests.exs → custom tests for validation and logic
- spe_test.exs → official tests
- spe_test_with_tags.exs → official tests with @tag for better testing

```
@tag :submit_good_jobs
test "submit_good_jobs" do···
end
```

Command examples

- Run all
  - `mix test`
- Specific test file
  - `mix test test/spe_test.exs`
  - `mix test test/spe_test_with_tags.exs`
  - `mix test test/our_test.exs`
- Specific tagged test in spe_test_with_tags.exs
  - `mix test --only submit_good_jobs`

CHALLENGES 

- Initially, we mismanaged dependencies between modules, which led to confusing and unstable execution flows.
- We did not properly separate logic across files, making the codebase difficult to understand and maintain.
- We encountered frequent {:already_started, pid} errors, especially when re-running tests or restarting the system. These PID conflicts caused many tests to fail unpredictably, making debugging time-consuming.
- We also faced task duplication issues, where some tasks were incorrectly launched multiple times.

FUTURE IMPROVEMENTS

We focused on completing all mandatory features of the SPE project, including job validation, concurrent execution with dependency resolution, failure handling, and PubSub notifications.

Due to time constraints, we did not implement the optional extensions, which could be valuable future improvements:

- Sleeping Tasks ⟶ Allow tasks to pause and release worker slots until reactivated.
- Distributed SPE ⟶ Enable load balancing and job sharing between nodes.
- Persistence ⟶ Store job and task states on disk to recover after server crashes.
- Task Priority ⟶ Let jobs define task priorities to influence execution order.