



UNIVERSITÀ DI PISA

Computer Engineering

Distributed System and Middleware Technologies

A distributed computation architecture

Edoardo Geraci

Contents

1. Project specifications.....	3
1.1 Project proposal.....	3
1.2 Functional requirements.....	3
1.3 Non-functional requirements.....	3
2. System architecture.....	4
2.1 Cluster controller.....	5
2.1.1 cowboy_listener.....	5
2.1.2 cowboy_http_requests_handler.....	6
2.1.3 cowboy_ws_requests_handler.....	6
2.2 Worker node.....	8
2.2.1 worker_server.....	8
Key Functionalities.....	8
Resilience Features.....	8
2.3 Web application.....	9
2.3.1 Application Overview.....	9
2.3.2 Declared Routes.....	9
2.3.3 WebSocket Handling.....	10
2.3.4 Process Flow.....	11
3. User manual.....	12
3.1 Cluster.....	12
3.2 Web application.....	13
3.2.1 Registration page.....	13
3.2.2 Login page.....	14
3.2.3 Task management.....	14
3.3.4 Cluster management.....	17

1. Project specifications

1.1 Project proposal

This project aims to create an architecture that enables users to easily distribute work across a group of machines they own, following a map-reduce approach to achieve parallelization. The architecture consists of a central backend that allows users to register and configure distributed tasks, along with two Erlang executables that users must run on their machines to make them ready to receive instructions for the dispatched work and execute it.

1.2 Functional requirements

The application has 2 types of actors:

- Unregistered users;
- Registered users.

The possible actions that each actor can perform are:

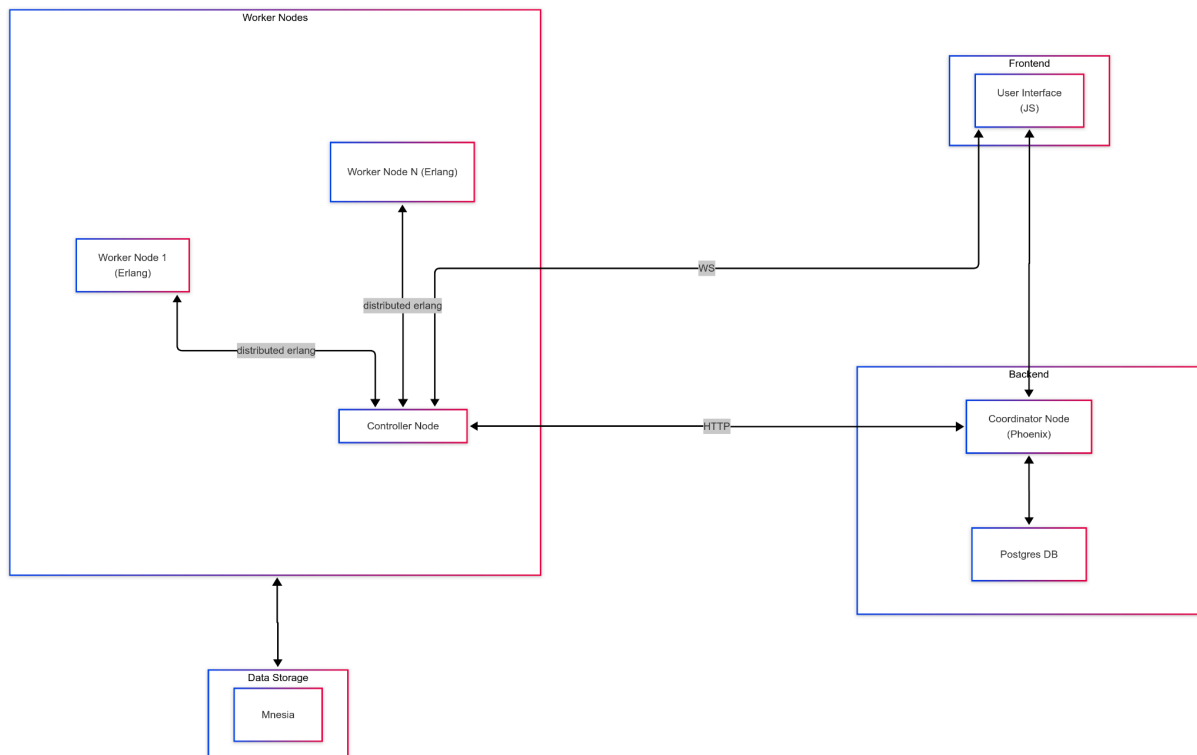
- **Unregistered users:**
 - Create an account
- **Registered users:**
 - Log in / Log out;
 - Submit multiple computation tasks with input data and code;
 - Create and manage multiple worker clusters;
 - View statistics regarding the jobs progress;
 - View the final result;

1.3 Non-functional requirements

- The system must scale horizontally
- The system must ensure fault tolerance:
 - Coordinator and worker nodes must recover gracefully from crashes.

2. System architecture

The following graph shows how the architecture looks with a single worker cluster of a single user:



The web application consists of a Phoenix backend and a JavaScript frontend, allowing registered users to manage clusters and tasks to distribute workload efficiently. Given the nature of the interactions between the web application and the worker cluster, a user could also implement their own web application or CLI interface to communicate with their cluster, entirely bypassing the need for the provided web application.

While the web application is shared between all the users, clusters are deployed by each user on his machines.

A cluster consists of at least two nodes: a controller node and a worker node.

The controller node runs an HTTP server that provides an endpoint for task setup and execution, as well as a WebSocket endpoint for real-time progress updates and result retrieval.

The worker communicates with the controller using distributed Erlang and executes the user-provided Erlang module on a portion of the original input. The input is divided based on the number of worker nodes available at the start of the task.

2.1 Cluster controller

Every cluster needs one cluster controller, which is an Erlang application implementing an HTTP server used both by the phoenix back-end for the task startup and by the JavaScript front-end in order to exploit the WebSocket connection to retrieve the task execution progression and the final result.

The cluster controller implements an HTTP server that exposes two endpoints:

Path	Functionality
/start_cluster	Used to initialize and start a task
/websocket	Upgrade the connection to websocket

The `/start_cluster` endpoint only accepts HTTP POST requests, handled by the `cowboy_http_requests_handler` module.

Upon startup, the application process initiates the creation of a supervisor process, known as `controller_sup` callback module. This supervisor is responsible for handling the lifecycle of the process that operates the Cowboy listener. The process responsible for running Cowboy is designed as a `gen_server` module, known as `cowboy_listener` callback module. The Cowboy server listens on port **1337**, awaiting incoming HTTP requests or WebSocket connection requests. Once a protocol upgrade request is received, the cowboy listener spawns a new process to manage that specific connection. As a result, each instance of the front-end is assigned to a distinct Erlang process.

The HTTP post request on `/start_cluster` is handled by the `cowboy_http_requests_handler` module, while the WebSocket connections are handled by the `cowboy_ws_requests_handler` module.

2.1.1 cowboy_listener

The **`cowboy_listener`** module acts as a controller for managing a distributed Erlang cluster using the **Cowboy HTTP server** and **Mnesia** as a distributed database. It defines a **`gen_server`** behavior and provides the following key functionalities:

- **`start_link/0`** – Initializes the Mnesia database, sets up necessary tables, and starts the Cowboy server.
- **`routes/0`** – Defines HTTP and WebSocket routes for cluster control.
- **`init/1`** – Configures and starts the Cowboy HTTP listener on port 1337, setting up request dispatching.
- **`handle_call/3`**
 - `{add_node_to_mnesia_cluster, MnesiaNode}` – Adds a new node to the Mnesia cluster.
 - `{create_erlang_task, [TaskId, TaskModule, Input, ProcessNumber]}` – Distributes a computational task across worker nodes, managing input splitting and dispatching.

- {ws_request, get_status} – Retrieves the status of the currently running task.
- {get_final_result, TaskId} – Fetches the final computed result of a task.
- Default: Handles unexpected requests with an error message.
- **handle_info/2** – Logs unexpected messages received by the server.
- **handle_cast/2**
 - {worker_communication, Message} – Handles messages from worker nodes, tracking progress and updating task statuses.

2.1.2 cowboy_http_requests_handler

The **cowboy_http_requests_handler** module serves as an HTTP request handler for the Cowboy web server, handling cluster-related HTTP requests. Key functionalities include:

- **init/2** – Handles incoming requests by method and path, specifically processing POST /start_cluster requests.
- **multipart/1,2** – Parses multipart form data from HTTP requests, extracting parameters and file uploads.
- **stream_file/1,2** – Reads and accumulates file uploads in a streaming fashion.
- **start_cluster/2** –
 - Extracts required parameters (TaskId, ErlangModule, Input, ProcessNumber) from the request.
 - Calls cowboy_listener to create and start a distributed Erlang task.
 - Responds with 201 Created if successful, or 400 in case of errors.
- **handle_not_found/2** – Returns a 404 Not Found response for unrecognized routes.
- **parse_body/2** – Extracts and validates required parameters from the request body.
- **check_required_params/2** – Ensures all required parameters are present.
- **get_param/2** – Retrieves a specific parameter value from the parsed request body.

2.1.3 cowboy_ws_requests_handler

The cowboy_ws_requests_handler module manages WebSocket connections for real-time communication with the cluster. It provides the following functionalities:

- **init/2** – Initializes a WebSocket connection and transitions to Cowboy's WebSocket mode.
- **websocket_init/1** – Registers the WebSocket process with a unique name and requests the current cluster status.
- **websocket_handle/2** – Handles incoming WebSocket JSON messages, and based on the keys handles them.
The keys can be:
 - <<"keepalive">> – Responds to keepalive messages.

- <<"command">>, <<"task_id">>

Where the command can be

- <<"get_final_result">> — Retrieves the final result of a task from cowboy_listener and sends it to the client.

- Invalid messages return an error response.

- **websocket_info/2** – Handles asynchronous messages:

- send_status – Fetches the current cluster status from cowboy_listener and sends it to the client.
- {info, Info} – Sends real-time updates (progress or errors) from worker nodes.

- **terminate/3** – Cleans up when the WebSocket connection is closed.

2.2 Worker node

The worker node is the one actually executing the deployed task and the data aggregation to produce the final result. All the worker nodes communicate their existence to the controller node at start-up, and are added to the mnesia cluster, in order to be able to access the stored data and write the partial result calculated during the task execution.

It is designed as a `gen_server` module and implemented via the `worker_server` module.

2.2.1 worker_server

The `worker_server` module implements a **distributed worker** for executing tasks within a cluster using **Erlang's gen_server behavior**. It manages task execution, monitoring, and failure handling.

Key Functionalities

1. **Initialization & Mnesia Cluster Setup**
 - Starts Mnesia for distributed storage.
 - Registers itself with the **controller node** (`cowboy_listener`).
 - Logs initialization steps.
2. **Handling Task Execution**
 - `setup_erlang_task/3`
 - Retrieves the task module from Mnesia.
 - Compiles the module dynamically.
 - Ensures `run/1` and `aggregate/1` functions are exported.
 - `start_erlang_task/2`
 - Retrieves input data from Mnesia.
 - Splits input into smaller workloads.
 - Spawns and starts worker processes for execution.
 - `aggregate_partial_results/2`
 - Collects partial results from Mnesia.
 - Calls the **aggregator function** to finalize results.
3. **Process Monitoring & Fault Handling**
 - **Monitors worker processes** to track failures.
 - If a worker **crashes**, it:
 - Kills all other workers.
 - Notifies the controller (`cowboy_listener`).
 - Clears the state to prevent further errors.
 - If a worker **finishes normally**, it notifies the controller of task completion.
4. **Inter-Process Communication**
 - Uses `gen_server:call/2` for synchronous requests.
 - Uses `gen_server:cast/2` for async messages to the controller.

Resilience Features

- Automatically detects and handles **crashed tasks**.

- Uses **monitors** (DOWN messages) to track worker health.
- Supports **distributed execution** by integrating with a **centralized controller**.

2.3 Web application

2.3.1 Application Overview

The application is built using the Phoenix framework and is structured to handle HTTP requests and WebSocket communication. The backend interacts with a PostgreSQL database and manages real-time events using Phoenix channels.

2.3.2 Declared Routes

The application defines several routes to handle client requests. These routes are primarily defined in the `router.ex` file and map to respective controllers for processing.

HTTP Routes

Method	Path	Controller & Action
GET	/users/register	BackendWeb.UserRegistrationController :new
POST	/users/register	BackendWeb.UserRegistrationController :create
GET	/users/log_in	BackendWeb.UserSessionController :new
POST	/users/log_in	BackendWeb.UserSessionController :create
GET	/	BackendWeb.TaskController :index
GET	/tasks	BackendWeb.TaskController :index
GET	/tasks/:id/edit	BackendWeb.TaskController :edit
GET	/tasks/new	BackendWeb.TaskController :new
GET	/tasks/:id	BackendWeb.TaskController :show
POST	/tasks	BackendWeb.TaskController :create
PATCH	/tasks/:id	BackendWeb.TaskController :update
PUT	/tasks/:id	BackendWeb.TaskController :update
DELETE	/tasks/:id	BackendWeb.TaskController :delete

POST	/tasks/:id/start	BackendWeb.TaskController :start_task
POST	/tasks/:id/update_status	BackendWeb.TaskController :change_status
GET	/clusters	BackendWeb.ClusterController :index
GET	/clusters/:id/edit	BackendWeb.ClusterController :edit
GET	/clusters/new	BackendWeb.ClusterController :new
GET	/clusters/:id	BackendWeb.ClusterController :show
POST	/clusters	BackendWeb.ClusterController :create
PATCH	/clusters/:id	BackendWeb.ClusterController :update
PUT	/clusters/:id	BackendWeb.ClusterController :update
DELETE	/clusters/:id	BackendWeb.ClusterController :delete
DELETE	/users/log_out	BackendWeb.UserSessionController :delete

These routes facilitate user management, cluster management, task execution, and retrieval of results.

2.3.3 WebSocket Handling

The application utilizes Phoenix Channels to establish WebSocket communication, enabling real-time data updates.

WebSocket Endpoints

- **/socket** - WebSocket endpoint defined in `endpoint.ex`.
- **Channel: ClusterChannel** - Handles cluster-related WebSocket events.

WebSocket Events

Event	Description
join	Establishes a WebSocket connection
keepalive	Keeps the connection active
get_final_result	Requests the final result of a task
status_update	Sends real-time task progress updates

WebSocket Logic

- Clients subscribe to `ClusterChannel` for task updates.
- Upon receiving a `get_final_result` event, the server fetches the task result from the database and responds.
- The server periodically broadcasts `status_update` events to clients with real-time progress.

2.3.4 Process Flow

1. A client starts a cluster task by sending a `POST /start_cluster` request.
2. The system processes the request, distributing the workload among worker nodes.
3. Each worker executes its assigned task and reports progress via WebSocket updates.
4. Once all tasks are complete, the final result is stored in the database.
5. Clients retrieve the result via `GET /results/:id` or request it through WebSockets.

3. User manual

3.1 Cluster

The user has to deploy the cluster nodes on his machines. To compile the erlang releases it's necessary to follow the [installation guide for rebar3](#).

Once rebar3 is ready, the user can create the release version of the nodes by executing the following commands

```
rebar3 release -n controller
rebar3 release -n worker
```

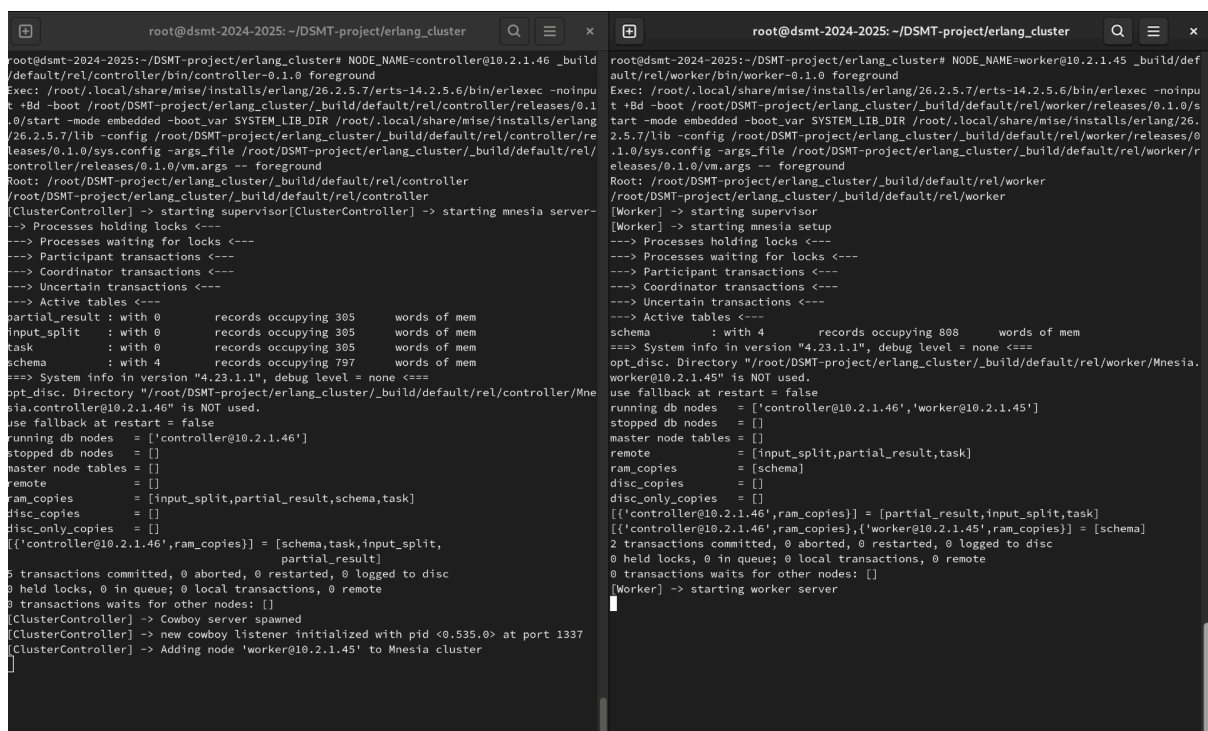
Once this is done, the user must first start the controller node on one of his machines by running

```
NODE_NAME=controller@<machine_ip>
_build/default/rel/controller/bin/controller-0.1.0 foreground
```

and then start as many workers as he need by running

```
NODE_NAME=<different_name_per_worker>@<machine_ip>
_build/default/rel/worker/bin/worker-0.1.0 foreground
```

Each worker will connect to the controller node and the controller and worker node will be something like:




```
root@dsmt-2024-2025: ~/DSMT-project/erlang_cluster
root@dsmt-2024-2025: ~/DSMT-project/erlang_cluster# NODE_NAME=controller@10.2.1.46 _build/default/rel/controller/bin/controller-0.1.0 foreground
Exec: /root/.local/share/mise/installs/erlang/26.2.5.7/erts-14.2.5.6/bin/erlexec -noinput +Bd -boot /root/DSMT-project/erlang_cluster/_build/default/rel/controller/releases/0.1.0/start -mode embedded -boot_var SYSTEM_LIB_DIR /root/.local/share/mise/installs/erlang/26.2.5.7/lib -config /root/DSMT-project/erlang_cluster/_build/default/rel/controller/releases/0.1.0/sys.config -args_file /root/DSMT-project/erlang_cluster/_build/default/rel/controller/releases/0.1.0/vm.args -- foreground
Root: /root/DSMT-project/erlang_cluster/_build/default/rel/controller
/root/DSMT-project/erlang_cluster/_build/default/rel/controller
[ClusterController] -> starting supervisor[ClusterController] -> starting mnesia server-
--> Processes holding locks <--
--> Processes waiting for locks <--
--> Participant transactions <--
--> Coordinator transactions <--
--> Uncertain transactions <--
--> Active tables <--
partial_result : with 0 records occupying 305 words of mem
input_split : with 0 records occupying 305 words of mem
task : with 0 records occupying 305 words of mem
schema : with 4 records occupying 797 words of mem
=== System info in version "4.23.1.1", debug level = none <===
opt_disc. Directory "/root/DSMT-project/erlang_cluster/_build/default/rel/controller/Mnesia.controller@10.2.1.46" is NOT used.
use fallback at restart = false
running db nodes = ['controller@10.2.1.46']
stopped db nodes = []
master node tables = []
remote = []
ram_copies = [input_split,partial_result,schema,task]
disc_copies = []
disc_only_copies = []
[{'controller@10.2.1.46',ram_copies}] = [schema,task,input_split,partial_result]
0 transactions committed, 0 aborted, 0 restarted, 0 logged to disc
0 held locks, 0 in queue; 0 local transactions, 0 remote
0 transactions waits for other nodes: []
[ClusterController] -> Cowboy server spawned
[ClusterController] -> new cowboy listener initialized with pid <0.535.0> at port 1337
[ClusterController] -> Adding node 'worker@10.2.1.45' to Mnesia cluster
]

root@dsmt-2024-2025: ~/DSMT-project/erlang_cluster
root@dsmt-2024-2025: ~/DSMT-project/erlang_cluster# NODE_NAME=worker@10.2.1.45 _build/default/rel/worker/bin/worker-0.1.0 foreground
Exec: /root/.local/share/mise/installs/erlang/26.2.5.7/erts-14.2.5.6/bin/erlexec -noinput +Bd -boot /root/DSMT-project/erlang_cluster/_build/default/rel/worker/releases/0.1.0/start -mode embedded -boot_var SYSTEM_LIB_DIR /root/.local/share/mise/installs/erlang/26.2.5.7/lib -config /root/DSMT-project/erlang_cluster/_build/default/rel/worker/releases/0.1.0/sys.config -args_file /root/DSMT-project/erlang_cluster/_build/default/rel/worker/releases/0.1.0/vm.args -- foreground
Root: /root/DSMT-project/erlang_cluster/_build/default/rel/worker
/root/DSMT-project/erlang_cluster/_build/default/rel/worker
[Worker] -> starting supervisor
[Worker] -> starting mnesia setup
--> Processes holding locks <--
--> Processes waiting for locks <--
--> Participant transactions <--
--> Coordinator transactions <--
--> Uncertain transactions <--
--> Active tables <--
schema : with 4 records occupying 808 words of mem
=== System info in version "4.23.1.1", debug level = none <===
opt_disc. Directory "/root/DSMT-project/erlang_cluster/_build/default/rel/worker/Mnesia.worker@10.2.1.45" is NOT used.
use fallback at restart = false
running db nodes = ['controller@10.2.1.46','worker@10.2.1.45']
stopped db nodes = []
master node tables = []
remote = [input_split,partial_result,task]
ram_copies = [schema]
disc_copies = []
disc_only_copies = []
[{'controller@10.2.1.46',ram_copies}] = [partial_result,input_split,task]
[{'worker@10.2.1.45',ram_copies}] = [schema]
2 transactions committed, 0 aborted, 0 restarted, 0 logged to disc
0 held locks, 0 in queue; 0 local transactions, 0 remote
0 transactions waits for other nodes: []
[Worker] -> starting worker server
```

3.2 Web application

3.2.1 Registration page

A user can register by visiting the registration page at the `/users/register` endpoint. After the registration the user will be logged in and redirected to the tasks management endpoint.

 Task Dispatcher

Register Log in

Tasks Clusters

Register for an account

Already registered? [Log in](#) to your account now.

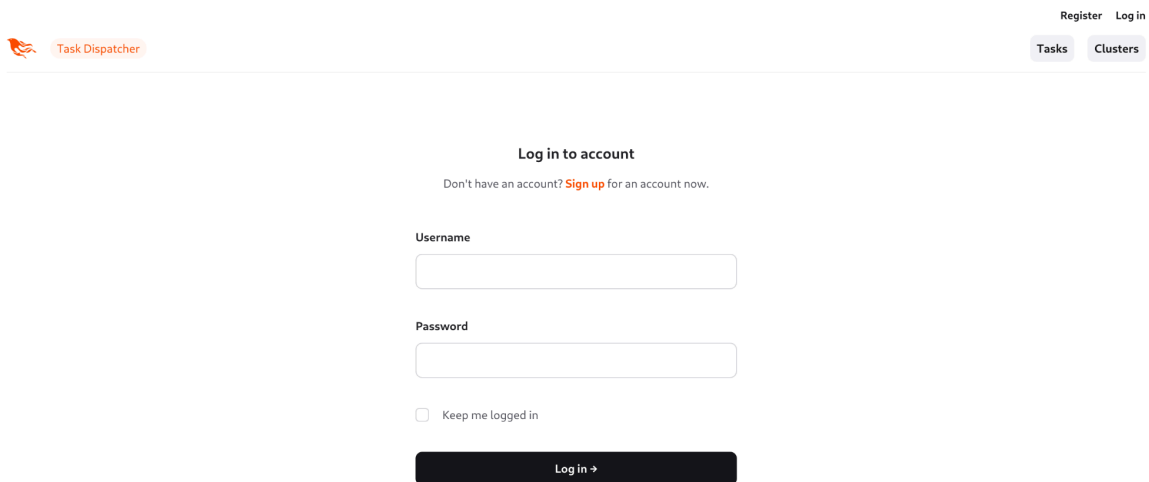
Username

Password

Create an account

3.2.2 Login page

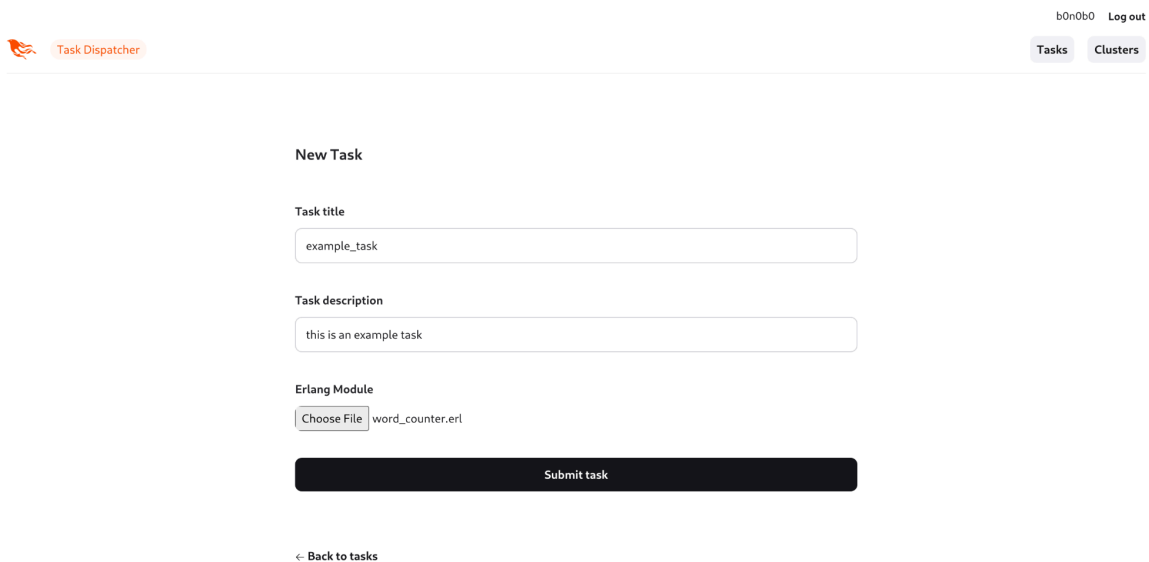
An already registered user can login by visiting the login page at the /users/log_in endpoint. After the login the user will be redirected to the tasks management endpoint.



The screenshot shows the login page of the Task Dispatcher application. At the top left is the 'Task Dispatcher' logo. At the top right are links for 'Register' and 'Log in', and buttons for 'Tasks' and 'Clusters'. The main heading is 'Log in to account'. Below it is a link: 'Don't have an account? [Sign up](#) for an account now.' There are two input fields: 'Username' and 'Password'. Below the password field is a checkbox labeled 'Keep me logged in'. At the bottom is a black button with the text 'Log in →'.

3.2.3 Task management

From the task management interface it is possible to create, delete and manage tasks. The task creation is done via a simple form where the user can set the task title, the task description and upload the erlang module implementing the task's logic.



The screenshot shows the task management page of the Task Dispatcher application. At the top left is the 'Task Dispatcher' logo. At the top right are links for 'b0n0b0' and 'Log out', and buttons for 'Tasks' and 'Clusters'. The main heading is 'New Task'. There are three input fields: 'Task title' (containing 'example_task'), 'Task description' (containing 'this is an example task'), and 'Erlang Module' (containing 'word_counter.erl'). Below the Erlang Module field is a 'Choose File' button. At the bottom is a black button with the text 'Submit task'. At the very bottom is a link: '← Back to tasks'.

Then, once created the task, the user can decide on which of his clusters to execute it and start the task via a similar form or modify it. To start the task, the user must submit an input file and the number of processes per worker

that are to be created to process the input file content.

example_task

Edit task

Description	this is an example task
Erlang model	<pre>-module(task). -export([run/1, aggregate/1]). word_counter([], Counter)-> Counter; word_counter([Word Words], Counter)-> NewCounter = Counter + 1, word_counter(Words, NewCounter). run(InputSplit) -> Result = word_counter(InputSplit,0), Result. sum_partial_counts([],Counter)-> Counter; sum_partial_counts([Count PartialCounts], Counter)-> NewCounter = Counter + Count, sum_partial_counts(PartialCounts, NewCounter). aggregate(PartialResults)-> Counter = sum_partial_counts(PartialResults, 0), Counter.</pre>
Final Result	
Choose cluster	<div>asd</div>
Number of processes per node	<div></div>
Input file	<div>Choose File No file chosen</div>
<div>Start task</div>	


Once executed, the final result is shown in the same page:

example_task

Edit task

Description	this is an example task
Erlang model	<pre>-module(task). -export([run/1, aggregate/1]). word_counter([], Counter)-> Counter; word_counter([Word Words], Counter)-> NewCounter = Counter + 1, word_counter(Words, NewCounter). run(InputSplit) -> Result = word_counter(InputSplit,0), Result. sum_partial_counts([],Counter)-> Counter; sum_partial_counts([Count PartialCounts], Counter)-> NewCounter = Counter + Count, sum_partial_counts(PartialCounts, NewCounter). aggregate(PartialResults)-> Counter = sum_partial_counts(PartialResults, 0), Counter.</pre>
Final Result	336000
<div>← Back to tasks</div>	

The tasks are all accessible from the main task management endpoint:

 Task Dispatcher

b0n0b0 [Log out](#)

Tasks Clusters


Listing Tasks

New Task

example_task	Edit	Delete
--------------	------	--------

3.3.4 Cluster management

From the cluster management interface it is possible to create, delete and manage clusters. The cluster creation is done via a simple form where the user can set the cluster name, the cluster api key (not actually used in the project, but there as a proof of concept) and the url to be used to contact the cluster controller node.

 Task Dispatcher b0n0b0 [Log out](#)

[Tasks](#) [Clusters](#)

New Cluster

Name

example

Cluster API key

example

Cluster controller node URL

<http://example.com>

Save Cluster

[← Back to clusters](#)