# Distributed Task Scheduler

## Project Overview

We shall design a distributed task scheduler in which a client can register a task and the time it should be executed. The task needs to be picked up and executed within 10 seconds of its scheduled time of execution. The tasks can be of two types:

* **one-time task:** These are scheduled at a specific time, and once executed will not be repeated again
* **recurring tasks**: These can be scheduled using Cron syntax and the system needs to execute them repeatedly according to the schedule

Once a task is executed it should be logged in a separate view in the UI. The tasks donʼt need to actually execute anything, just be logged along with the exact time they got executed.

## Requirements

### Functional Requirements

| FR01 | Clients must be able to register a task with either a specific time of execution or cron syntax for recurring tasks |
| --- | --- |
| FR02 | Clients must be able to see their current list of scheduled tasks. |
| FR03 | Clients must be able to delete their tasks. |
| FR04 | Clients must be able to edit their tasks. |
| FR05 | Clients must be notified about the execution of their tasks. |
| FR06 | Clients must be able to define a group of users to be notified about a task execution |
| FR07 | Clients must be able to export their task to Google Agenda application |
| FR08 | One-time tasks shall be registered in date time format |
| FR09 | Recurrent tasks shall be registered in cron expression format |
| FR10 | Clients must be able to sign up in the application. |

### Non-Functional Requirements

| NFR01 | The scheduled date for one-time tasks shall be saved in the Pacific time zone |
| --- | --- |
| NFR02 | A task should be picked up for execution within 10 seconds of its scheduled execution |
| NFR03 | The system shall durable, no matter what happens to any component in the system |
| NFR04 | The BE application shall be able to scale vertically and horizontally without errors |
| NFR05 | The system shall prevent multiple nodes from executing the same job simultaneously |
| NFR06 | BE and FE communication shall be protected by JWT token |
| NFR07 | The system shall be able to retry a job execution in case of failures |
| NFR08 | The system shall auto restart in case of failures |

### MVP Requirements

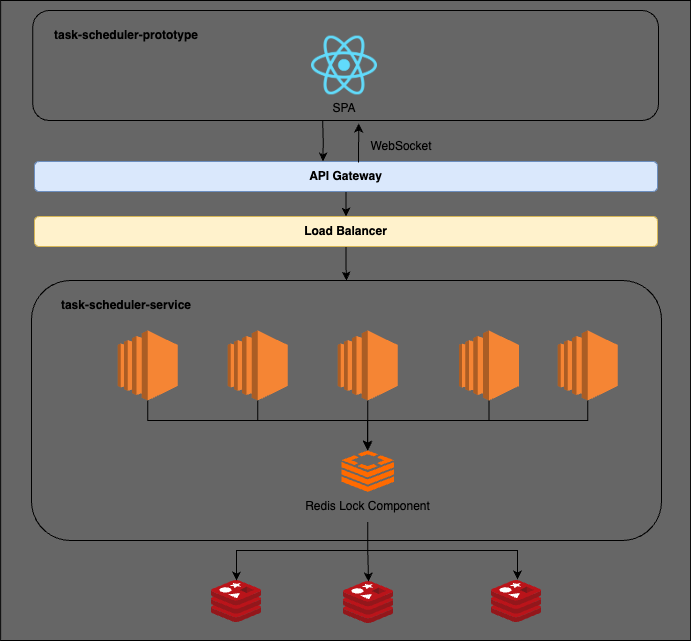
Since this is a MVP, the following requirements were implemented: FR01, FR02, FR03, FR05, FR08, FR09, NFR01, NFR02, NFR03, and NFR08. The remaining requirements were postponed to the next phases of the project.

## Architecture

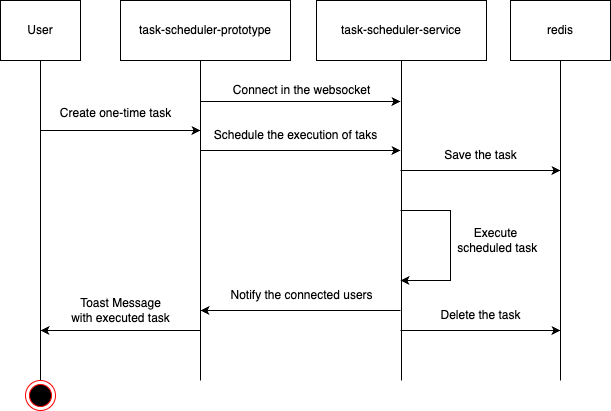
To meet all the proposed requirements, we designed a distributed task scheduling architecture leveraging Redis for efficient task storage and execution management. By using Redis, we ensure a distributed cron system, preventing duplicate task execution in a simple yet effective manner. To further enhance concurrency control, we implement Redis Redlock, a distributed locking mechanism that ensures mutually exclusive access to shared resources. The backend is built with Express.js, exposing RESTful APIs to create, edit, list, and delete scheduled tasks while also sending real-time notifications to the frontend via WebSockets. On the frontend, a React SPA is responsible for managing tasks, utilizing React Query for caching and state synchronization. The UI listens for execution updates using a WebSocket client, ensuring seamless real-time task monitoring and user notifications.

### Diagrams

Below is a high-level architecture diagram illustrating all the components and their interactions. Additionally, on the next page, we present an activity diagram detailing the steps required to create a one-time task, from the moment the user accesses the UI to when they receive a notification confirming the task's execution.

**Architecture Diagram**

**Activity Diagram**



## Architectural Trade-off & Potential Challenges

| **Aspect** | **Pros** | **Cons** |
| --- | --- | --- |
| Redis as the Task Store | Redis for task storage and execution allows for easy horizontal scaling, making it well-suited for distributed environments. | Redis is an in-memory database, meaning it has limited persistence and could lose scheduled tasks if not properly configured (e.g., AOF persistence or snapshots). A more durable alternative could be a message queue (Kafka, RabbitMQ) or a dedicated task scheduler (Temporal, BullMQ). |
| Redlock for Distributed Locks | Implementing Redis Redlock ensures that scheduled tasks are executed only once, even in a distributed system. | While Redlock helps prevent duplicate execution, it introduces additional network overhead and may have reliability issues under high latency or partitioned networks. |
| WebSockets for Notifications | WebSockets provide low-latency notifications, improving the user experience by delivering task execution updates instantly. | WebSockets require maintaining open connections, which may not scale efficiently with a high number of users. |
| Frontend React Query Usage | Using React Query optimizes API calls, reducing load on the backend and improving UI responsiveness. | With a high number of tasks the volume of cache invalidations could be very high. |