Challenge description

You are required to write a web API with 2 endpoints:

1. A “Set timer” endpoint.
   - Receives a JSON containing hours, minutes, seconds and web url
   - This endpoint should return a JSON with a single field - “id"
   - This endpoint should define an internal timer, which shoots a webhook to the defined URL after the time ends (a POST HTTP call with an empty body)
   - For example:
     - `POST /timers {hours: 4, minutes: 0, seconds: 1, url:"https://someserver.com"}` should return `{id: 1}`
     - After 4 hours and 1 second, server should call `POST https://someserver.com/1`
   - The counter id should be appended to the URL, i.e.: `https://someserver.com/:counter_id`

2. A “Get timer status” endpoint
   - Receives timer id in the URL, as the resource id
   - Returns a JSON with the amount of seconds left until the timer expires. If timer already expired, returns “0”
   - For example:
     - `GET /timers/1` Should return `{id: 1, time_left: 645}`

Additional requirements:
- The timers should persist.
- If we shut down the process and restart it, timers should be saved.
- If a timer should have fired when the server was down, the server should fire the web hook after initializing.

Notes:
- Please use Python/ Javascript / Typescript / Java or C#.
- Please add a readme with clear execution instructions (solution will be tested in macos environment).
- If you’re using another language, please package accordingly so we could install and run it with minimum friction (ie packed in docker).
- Solutions will be measured by code structure and by their ability to handle large scale (large number of timers).
- Feel free to add any extra functionality you wish.
- You’re welcome to approach me for any questions you may have.

Instructions

The project is using the technologies below:

- NodeJS
- Typescript
- MongoDB
- Redis

If you don't have installed MongoDB and Redis please follow these steps:

MongoDB

Type in the terminal:

```
$ brew tap mongodb/brew
$ brew install mongodb-community
$ brew services start mongodb-community
If you have a previous version of mongodb
$ brew services stop mongodb
$ brew uninstall mongodb
$ brew tap mongodb/brew
$ brew install mongodb-community
$ brew services start mongodb-community
```

To execute MongoDB as a service:

```
$ brew services start mongodb
```

Redis

Install redis:

```
$ brew install redis
```

Start redis server:

```
brew services start redis
```

That's everything we need to run our project.

Run project

1. Start MongoDB
    a. Run `brew services start mongodb`
2. Start Redis server
    a. Run `brew services start redis`
3. Install project dependencies
    a. Run `npm install`
4. Run project
    a. `npm run start`

After the steps above the application is ready to receive requests, you can use a client like Postman to call the local endpoints.

Endpoint	**Description**	**Response**
`GET /timers`	This endpoint fetches all of the crons we have scheduled in our app.	`[ { "_id": "6212c9c117070cffd45631d0", "url": "testing.com", "hours": 0, "minutes": 0, "seconds": 20, "isTriggered": false, } ]`
`GET /timers/:id`	Fetches the current timer status.	`{ "id": "6212c9c117070cffd45631d0", "time_left": 0 }`
`POST /timers {hours: 4, minutes: 0, seconds: 1, url:"https://someserver.com"}`	Schedule a new timer to trigger a webhook once the timer is expired	`{ "id": "6212c9c117070cffd45631d0", }`
`DELETE /timers`	Remove all of the timers in the Database. This was used for testing.	{}

Scaling

- For simplicity we are using MongoDB. DynamoDB would be a better choice as we can scale horizontally almost limitlessly by adding more servers, it’s fast and we take a lot of the cloud benefits.
- Why are we using Redis?
    - Redis is a short-term memory that helps us to make better use of the resources we have. Usually caching takes advantage of the locality of reference principle: “Recently requested data is likely to be requested again”. So instead of hitting the database each time to check a timer status, the very first time we check the database and further request will search first in the cache. API calls are way more performant as we just need to compute business logic of time left instead of overwhelming the database with search operations each time.
- Scheduled jobs: We are using scheduled jobs to launch parallel processes that will trigger the webhook once the time has expired. We are improving performance and scalability because as we grow, we launch “n” async processes that are scheduled based on the expiration time, we are saving CPU usage and computations by getting rid of a service/jobscheduler process that runs constantly and checks each second for any timer expiration.
- As we could have scenarios where the server might go down, at the server start, we perform a check to validate if any timer has expired already but has not been triggered. For those timers, we trigger the webbook right away.
    - For distributed systems, we should always avoid single point of failures as we might degrade the service in case of failures, so that's why it's a good idea to have redundancy of critical components that help us to keep providing service in case of failures, and replication to have consistent information between resources. This helps us with reliability, fault-tolerance and accessibility.