Maersk Line Offline Technical Challenge

Introduction

This document provides a homework challenge for candidates for software engineering positions within Maersk Line.
The goal is to see:

- How you approach a problem
- How you design, structure, implement, and deliver a complete solution that is adapted to the proposed problem below

This will be followed by a discussion with Maersk Line software engineers.

General guidelines

- The delivery must include a markdown readme file with concise and complete instructions on how to use (unpack, build, install, execute, access, etc.) your service.
- Your code and other deliverables must be provided as a link to a private Git repository (VSTS, GitLab, GitHub or Bitbucket)
    - Please push your work regularly
- Your solution must only be accessible to you and us; please make sure it is not available for a wider audience, especially not publicly. We would like to reuse the challenge, please help us keeping it fair!

The problem

Sorting algorithms are well known in computer science. They are divided in two categories, linear and divide and conquer algorithms.
The latter offer better performance and one of them is the mergesort algorithm. If you are not familiar with the algorithm you can start with the relevant Wikipedia article or browse other Web sources.

The task

You are asked to develop a .NET Core web API that should do the following:

- Implement the mergesort algorithm
- Allow the user to provide an unsorted array of numbers
- Invoke the service asynschronously for each array provided by the user

- Each execution should be identified by a job id, timestamp and run time (duration)
- Retrieve all jobs with their id's, timestamps and status {pending, complete}
- Retrieve specific jobs given their id's with the resulting sorted array

We expect your solution to include the code for the algorithm - but you are welcome to reuse existing implementations of the mergesort algorithm, modified as well as unmodified.

Specifications

You are expected to meet all the minimal requirements. Extra credit will be given for meeting any of the additional requirements.

Functional requirements

Your delivery should:

- Include complete code needed to execute the solution
- Include a suite of unit tests covering key components of your solution
- Contain logging functionality
- Either contain all additional dependencies (e.g., external libraries, if you use any) or handle the download and installation thereof as needed (ideally, as part of the build process)
- Provide a REST API interface

Technical frameworks

- Readme.md (or .txt) to explain how to use the solution
- .NET Core Web API
- .NET Core v2+
- E.g. XUnit, Moq, FluentAssertions, and AutoFixture for the test framework

Minimally, it should be possible to

- Build and start your solution as a service on a local machine
- Interact with your solution through the REST API (e.g. using Swagger)
- All other dependencies should be handled within the images by a package manager corresponding to your code
- For persistence an in-memory repository sufficient

Additional requirements

- There should be no requirement to install anything on the host other than what is needed to provide an environment for executing the solution
- For the REST API, provide a complete documentation of all endpoints, with details of the required arguments, input and output formats, possible http status codes, and their semantics,

e.g. using Swagger

REST API

The following is a concise list of the REST API's calls for the solution. See the examples section below for their usage with curl.

- POST /mergesort
  - Content: application/json with mergesort problem specification, see below
  - Input: json with an array of numbers
  - Output: json with execution object, i.e. job id, timestamp, status
- GET /mergesort/executions
  - Output: json with list of executions and input array
- GET /mergesort/executions/{id}
  - Output: json with execution object and output sorted array

The json requests and responses should have at least the following formats:

```
#execution
{
    "execution": {
        "id": # non-negative number,
        "timestamp": #time/date started,
        "duration": # duration,
        "status": # {pending, completed}
        "input": # array of numbers,
        "output": # array of numbers, same size as input
    }
}

#executions
{
    "executions": {
         # array of job executions with statuts for each job
    }
}
```

Example session

The following is a hypothetical session, listening on port 8888:

> $ curl -XPOST -H 'Content-type: application/json' http://localhost:8888/mergesort/
> -d [3,5,1,83,51,99]
> {"id":"123","status":"pending"}

> $ curl -XGET -H http://localhost:8888/mergesort/executions/123
> {"id": "123", "timestamp":"112382233","duration":"12431",status:"completed" "input":

> [3,5,1,83,51,99], "output": [1,3,5,51,83,99]}