DFM

Coding Challenge

The task is to implement a custom priority queue system with both frontend and backend components. This project entails building a dashboard site that enables users to create simulations for this custom priority queue system, offering an interactive and visually engaging experience to understand and manage the priority queue.

The queue system has the following specifications:

• The server queue must be designed to ensure thread safety, with the assumption that it operates in multi-threaded environments.

• The queue has a user-determined fixed size (default: 100).

• When the queue reaches its capacity, the producer will be blocked until space becomes available.

• If the queue is empty, the consumer will be blocked until there are items in the queue.

• Each item in the queue has an assigned priority value.

• Priority values are positive integers ranging from 1 to 10, with 1 being the highest priority.

• Items can share the same priority value.

• The dequeuing process adheres to the following rules:

- Items with higher priority are dequeued first.
- For items with the same priority, the FIFO (First-In-First-Out) order is maintained.
- Rate Limit: Based on a rate limit value "N", "N" items with priority "x" are dequeued. After this, the next item must have a priority of "x+1". It is not necessary to dequeue "N" items consecutively; examples will be provided.

Reset or Ignore rate limit from this point?

Examples

Rate limit of N=2

x+1 priority item is dequeued since we hit rate limit. Move on to other

1. Input: "1 2 3 2 1 1 3"
   Output: "1 1 2 1 2 3 3 "
2. Input: "1 5 1 2 1 3 2"
   Output: "1 1 2 1 2 3 5 "
3. Input: "3 1 5 2 1 4 2 3 2 1 1 3 3 5 2 1 6 1 1 2 4 2 4 1 3 2 1 4 2 1 1 2 2 1 1"
   Output: "1 1 2 1 1 2 3 1 1 2 1 1 2 3 4 1 1 2 1 1 2 3 1 2 2 3 4 5 2 2 3 4 4 5 6 "

Tech Stack:

The frontend should be implemented using React Hooks and Material Design (MUI), while the backend should be developed in NodeJS using Express. There is no specific database requirement; you can follow an in-memory approach to store your sessions.

Starter projects are provided as part of this task. Please build your solutions on top of these projects.

Requirements

Implement the priority queue system using NodeJS with the assumption that it will be communicated with via REST calls from the frontend. The REST is implemented using Express.

The frontend starter code shows a landing screen that has two sections, History and Simulation.

The history section is expected to show the previous inputs (sessions) that a user provided. It also shows at the top a way for the user to enter an input sequence as well as the rate limit.

There is a button "Enqueue". When clicking this button, it will send a REST request so the backend will add it to system.

Below is a low-fidelity snapshot for the expected UI controls. Please make sure to implement it using MUI. The rate limit value will have its impact on the server logic as explained previously.

Bonus: Use Formik validation with the enqueue button when submitting a REST form request. If you are not going to use Formik, at least make sure that the inputs are a sequence of numbers. You can provide the required JavaScript sanitization function that strips alpha characters. Make sure that the Number Textfield Rate limit has a range of 2-10.

Assumption: Rate Limit Does not need to show up on FrontEnd. Keep Track since it impacts server logic

When selecting a session (clickable session) on the history pane, it will be reflected on the simulation section.

When creating a new session after clicking the enqueue button, it will be auto selected. Sessions are ordered based on their creation date where newest sessions come first.

The simulation section for a selected session has a toolbar with two main actions, Next and All. The Next action sends a request to the server to dequeue an item from the queue. On the other hand, the All action requests the server to keep dequeuing until all items are cleared.

How the dequeuing is represented visually?

Most recent session should be in Simulation

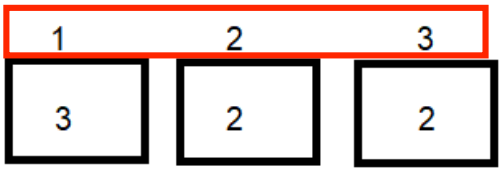
**1 is highest priority

The first time the server receives an input sequence, it will be divided and ordered based on priority into buckets.

Give the following input: "1 2 3 2 1 1 3", let us also assume that the rate limit is 2.

Dequeue should be a server method

In such a case, we are going to visually show three blocks (buckets) for each priority. Each block will show the priority occurrences (low-fidelity snapshot is below):



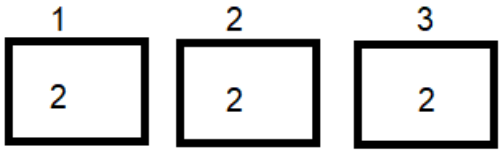
Priority Vals

# of occurances

3 of priority 1
2 of priority 2
2 of priority 3

Assumption: With more priorities, we need more blocks

When hitting next, this will dequeue an item of the highest priority, which is one. Thus, the next visualization will be:

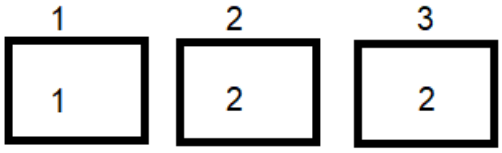


Rate limit N=2 -> means can only Dequeue 2 times

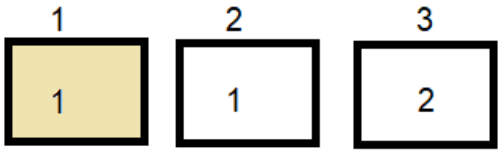
When rate limit hits, switch to processing items of priority X+1. DON'T DEQUEUE ANYTHING IN THIS STEP

Proceed as normal onwards

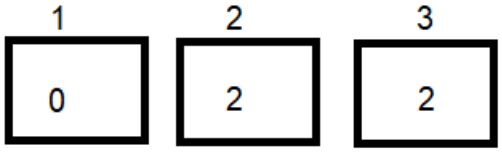
When hitting next again, it will be:



Now, we reached our rate limit "2". This means that hitting next will process items of the priority X +1. Thus, it will be as below. You will notice that the rate limited block is highlighted in a way that makes it possible to catch which blocks are rate-limited. It is possible in scenarios that multiple blocks can be blocked at the same time based on the rate limit provided.



When hitting next again, we know that the rate limit has been reset; this means that we will be back to the first block as below:



Note: As a part of the UI implementation, please make sure to show the stack of execution of the dequeuing processes.

Note: The “All” action button is used to dequeue a queue until no items are left. Make sure that all steps are displayed.

Bonus: Show animation when clicking the “All” button. The animation will show ste-by-step execution instead of only listing the steps as blocks right away.

Tests

Please make sure to implement test cases for your server code logic.

Documentation

Add Markdown (MD) README files to both starter codes, explaining your logic, and how to test and run the server.

Docker

- Dockerize both frontend and backend projects: Create Dockerfiles for both the frontend and backend projects. Ensure that the containers are optimized for size and do not exceed 100 MB of memory usage when deployed.
- Use Docker Compose to run both Docker containers.
- Deploy the application to a virtual server hosted on the cloud (for example, an EC2 instance on AWS).

Deployment

Please deploy your server and make it accessible to be tested using any of the available services such as Heroku. Alternatively, you can use forward tunneling services such as ngrok.

Technical Notes

- The root component of your API should assume two main parameters: data and onChange. Data is in JSON format, and its content determines what will render. onChange is a single event trigger with the following format: onChange({action: "update", value: someJson}). This allows for a single event handler for the entire page.
  A simple overview Example:

```
const [data, setData] = useState({mode: "configuration", selection: "option1"});
const onChangeExample = useCallback(({action, payload})=>{
switch (action) {
  case "startConfiguration":
    // You can make changes to your data using setData. For example: setData({... data, mode: "configuration", selection: null});
    return "do something";
  case "error":
    // You can make changes to your data using setData. For example: setData({... data, mode: "error"});
    return "do something";
  default:
    return "not recgonized";
  }
}, [data]);
return <YourRootComponent {... props} onChange={onChangeExample} data={data}/>
```

- Only use React Hooks.
- Must use i18Next for localization strings. There must not be any hardcoded strings.
- Only use Material Design (MUI) as your framework.
- Do not use or externalize styles in separate css/scss files. We need to use the MUI theme or externalize the styles props such as sx. In in ither words, there should not be any *.scss or *.css files that are needed to use in order for the UI to behave as expected.
- You do not have to use a state store library, but if you did, please only use Redux Saga. Using Redux Saga is a bonus.
- Try to decompose your components into smaller components as much as you can. We should avoid having a single file containing the entire UI rendering logic.
- Make good use of the React Hooks principals and build-in hooks such as useMemo, useCallback, useEffect, and useState. For instance, the main idea of using data and onChange is to help you decompose your components and elements and make them standalone. An element for instance, can useTranslation, useState, useMemo internally, while there is a single source of data "data" that can be changed using "onChange", which acts as the communication point between this component/element and other components.

Additional Questions

1. What is the difference in behaviour between these two snippets:

```
const [data, setData] = useState({});
useEffect(()=>{
  setData({});
});
useEffect(()=>{
  setData({});
}, []);
```

First: For all updates and re-renders, setData() runs. Useful for side effects for every render. Can create inf loop

Second: Runs setDate() only at initial render, when component mounts

2. What do you think of this snippet?

```
const [data, setData] = useState({});
useEffect(()=>{
  alert(data);
}, []);
```

At initial render, an alert with empty object is shown as “Object object”, thus not displaying the correct info

3. Consider the code below: We are implementing an observer design pattern. We have an object listenerManager that registers new observers. It is passed or retrieved when starting a component. Below is code for a component that registers a new observer. What are the possible issues that are related to this code?

```
const observer = ()=>{
   console.log(“do something”);
};
useEffect(()=>{
   listenerManager.register(observer);
});
```

- The code registers the observer at every mount and render. This can be inefficient

- Adding [] at the dependencies can make sure it is only registered at start