Summary

WonderQ is a broker that allows multiple producers to write to it, and multiple consumers to read from it. It runs on a single server. Whenever a producer writes to WonderQ, a message ID is generated and returned as confirmation. Whenever a consumer polls WonderQ for new messages, it gets those messages which are NOT processed by any other consumer that may be concurrently accessing WonderQ.

NOTE that, when a consumer gets a set of messages, it must notify WonderQ that it has processed each message (individually). This deletes that message from the WonderQ database. If a message is received by a consumer, but NOT marked as processed within a configurable amount of time, the message then becomes available to any consumer requesting again.

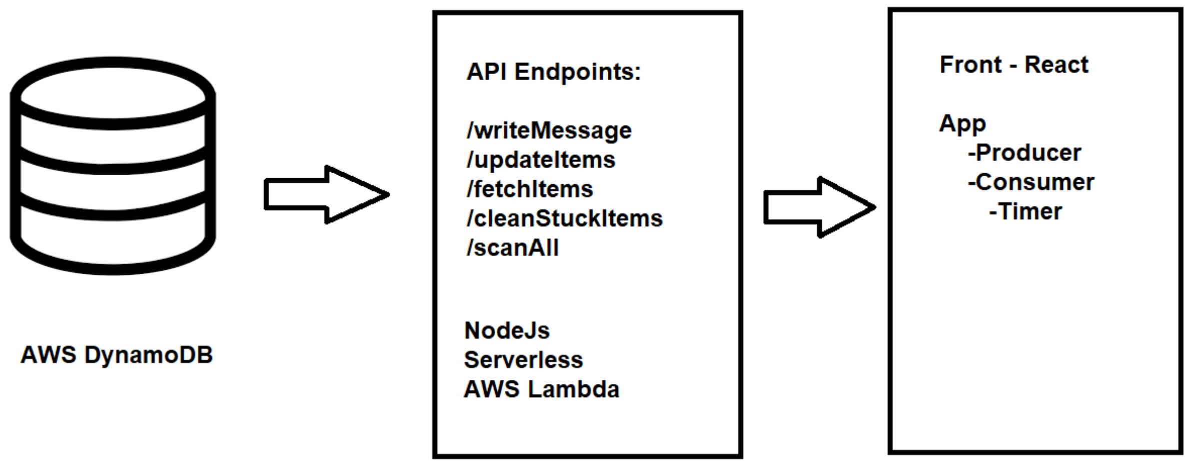
Tasks

- Design a module that represents WonderQ. You can abstract the logic around database storage.
- Setup a test app that will generate messages and demonstrate how WonderQ works.
- Setup a quick and dirty developer tool that can be used to show the current state of WonderQ at any time.
- Write documentation for potential API endpoints. Talk about their inputs/ outputs, formats, methods, responses, etc.
- Discuss in writing: how would you go about scaling this system to meet high-volume requests? What infrastructure / stack would you use and why?
- We'd prefer if you use Node.js and ES6/ES7 as that is what we use.
- We are looking for your software design skills, use of design principles, code clarity, testability, your thinking skills, and ability to add your own ideas.

How to run

- Lambda APIs readily available online, more on that below in the API documentation
- Open the wonderq-app React project, navigate to root folder and type ‘npm start’, the app will be available on localhost:3000
- 2 main modules in front-end:
  - Producer: text field + create button
    - Calls the /writeMessage
  - Consumer: list of messages + refresh (timed to 120s to auto) and submit button
    - Calls /fetchItems and /updateItems
- Consumer Behaviour:
  - If no selection is made, submit/refresh will not make any items processed
  - If selections are made:
    - refresh will not make any items processed, and refresh will clear out the processed array
    - submit will submit the processed items and update the selected items to Processed
  - If timer runs out, simply refresh, nothing is processed even if items were selected

Design



The design itself is pretty standard, a database storage, API endpoints to access databases and do any kind of back end processing needed, and a front-end to demonstrate the app to the users. I used DynamoDB here since it's on the AWS platform which is what I'm used to, and it's the cheapest option that is convenient without having to host PostgreSQL or MySQL on my own server. If given the option, I would use a relational database for this kind of application. The SDK and communication, as well as the speed of DynamoDB wasn't quite up to par with RDB options from what I've experienced.

The APIs run on NodeJS built on the Serverless framework, deployed onto AWS Lambda, these are responsible for write, update, read operations. The front-end uses ReactJs, with material-ui for some styling and fields, simple front-end to demonstrate and test out the app.

The original design was to simulate a queue using the database using the CreatedOn date field to sort, thereby it will be a FIFO type of way. However, I saw that Standard AWS SQS didn't seem to follow this pattern to my knowledge. Additionally, DynamoDB had some limitations in terms of what it can do for read and general data operations compared to the SQL Server RDB that I'm used to, so that affected the final design a bit.

I went with APIs that would talk with the database, and read/write/updates are all relatively simple relative to other RDB options. In terms of what I can do to fetch a certain number of items at a time in a certain order, this is where it starts to fall off; I settled for 3 messages to be displayed to consumers at a time, though this can be increased by specifying a query parameter in the endpoint.

The front end is written in ReactJs, the main being the App.jsx, I used child files for each of the modules here, Producer, Consumer, and Consumer uses a Timer which is another sub-child. The Producer panel has a textfield and a button to submit to database, put in some text that the user might want to request research about and press the button. The message gets written into the database and a message id GUID is returned to confirm.

The Consumer panel is a bit more tricky, and my interpretation of the WonderQ requirement was that there would be different statuses for the messages. Wordings such as “received by a consumer, but not marked as processed within an amount of time” or “delete that message from the database” led me to think that I should have at least 3 statuses, which are Available, Reviewing, and Processed.

- **Available** – message is available to be fetched and viewed by the consumer
- **Reviewing** – message is being looked at by a consumer in a browser, and is not fetchable by any other user (even if multiple users open the app, no 2 users will see the same list)
- **Processed** – if a consumer claims a task and submits, then the message becomes processed. Based on my initial phone screen, I imagine that this means the user is now assigned to this task for research and thereby it will not be available for anyone else. This is basically the same as the “delete” that was specified in the requirement, though it acts more as a “soft-delete” just to keep the record around.

It seems the idea is that the front-end will be used by multiple users looking for assignments in certain research topics, so not showing the same items to concurrent users is how I understood the topic. The user can check off items and submit to claim those tasks, or refresh, or wait for a timeout of 120 seconds for the list to auto-refresh. Submitting will also clear the module and refresh the list. If the timer runs out before the user can submit the selection they made, those selection will also be cleared (not processed).

In terms of scaling for high-volume, I would stick to the Node/Lambda setup as Node is largely popular and very powerful as a back end provider. Lambda basically auto-scales itself based on the number of requests that comes in, very good for hosting serverless APIs that are short and quick to call. There may be other options like Go, C# or Express, but from my experience with these kinds of services, Node/Lambda running on serverless is easy to extend and scale better without having to add on additional (physical) servers if using it.

In terms of database, I would make the switch to a RDB, if speaking by what I know, SQL Server or MySQL. But speaking for cost effectiveness and performance, I’ve heard good things about PostgreSQL so that would be my pick if I had more experience in Postgres.

In terms of front end, I would say React is still one of the top picks, there’s some tricks in React that I’m not quite as experienced with yet, but overall very solid. The only other pick I can think of is probably Vue.js, which I heard is a much lighter version of React.

API Endpoints

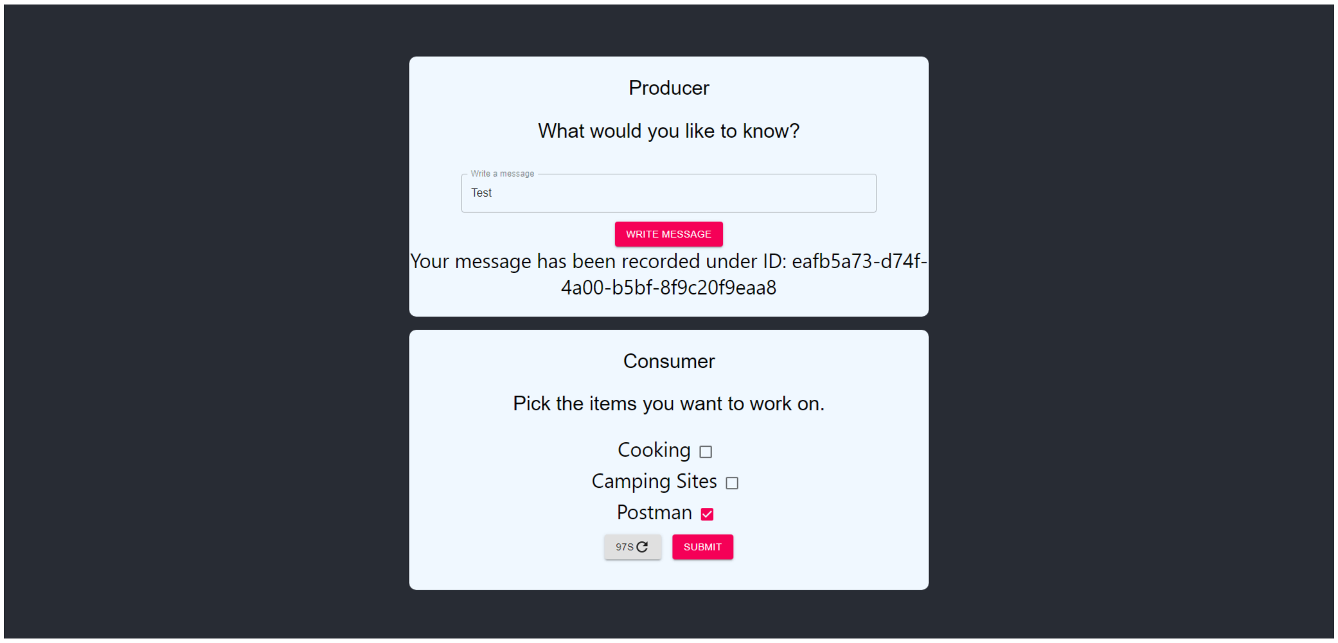
The APIs themselves are stateless, async GET/POSTs, they run as needed on the lambda and AWS cleans them out after execution. Below are the 5, one of the thing I’d fix is the domain name, I would create a domain just so the naming of these APIs would be nicer instead of some generated code from AWS, but at the time I didn't have one and I would've had to purchase a domain to use.

- **GET** - https://nqliq0rkvb.execute-api.us-east-1.amazonaws.com/dev/scanAll
  - No input parameters
  - Outputs a JSON, with one of them being an "Items" array object, which is an array of JSONs containing the records from the database
  - This is just a simple scan of the DynamoDB, which would be equatable to a Select * from Message in a SQL language, it's the developer tool requested to return the state of WonderQ at any time
- **POST** - https://nqliq0rkvb.execute-api.us-east-1.amazonaws.com/dev/writeMessage
  - Input parameters: Message string in body (may need to be x-www-form-urlencoded)
  - Outputs a JSON with messageId as the return value, this is the messageId assigned to the newly created message in the database, GUID format
  - I didn't put a check for duplicates here since I assume that many users may want to make the same requests. At the same time, if I could, I would set a non-duplicate restriction on the Message column, but I didn't see a way to do that in DynamoDB while keeping the design I wanted; this is where a more familiar RDB would've helped
- **POST** - https://nqliq0rkvb.execute-api.us-east-1.amazonaws.com/dev/updateItems
  - Input parameters: a list/array of messageIds to be updated (application/json format), and the target status in body
  - Outputs a message to confirm, along with all the messageIds that were updated
- **GET** - https://nqliq0rkvb.execute-api.us-east-1.amazonaws.com/dev/fetchItems
  - Input parameters: number of items (optional) as a query parameter. If this is not provided then it will just default to 3
  - Outputs the specified number of items as an array of JSON objects
  - The optional limit variable is just for some customization for the front-end developer for how many items they want to fetch
- **GET** - https://nqliq0rkvb.execute-api.us-east-1.amazonaws.com/dev/cleanStuckItems
  - No input parameters
  - Outputs the items that have been updated back to Available status
  - This is basically an implementation of the "within a customizable amount of time" requirement. I have a 120s timer on the front-end, but this is also a back-end scheduled API that runs every 60 minutes (can be changed) that basically checks the database if there's any item with "Reviewing" status that may have gotten stuck, and updates those back to Available. Basically a safety measure and an automated system to clean up any stuck items if there are any.

While it's possible to call these endpoints manually with something like Postman, since these are internal back-ends for developers to consume, it's fairly straightforward in what it needs to take in and output with not much for other inputs. If the input gets through the API and makes its way to the database, and the database declines it, the error message will be returned.

Conclusion & Other thoughts

Overall it’s a nice little project across the stack and involved some design and thought in terms of how I wanted to structure some things. Working with DynamoDB was a bit strange as they have their own SDKs to do operations on the data, and then there’s limitations on what I can do with primary keys and indexing for things like sorting or querying certain data that I wanted. This caused some moments where I needed to build workarounds or do things a certain way since DynamoDB doesn’t support certain features like a relational database would. There may be some things I missed or forgot to mention, but would love to hear feedback or if there are any questions, feel free to ask. There’s already some dummy data in the database for use, or you can add your own. I’ve put some more pictures below just for references.


Producer

What would you like to know?

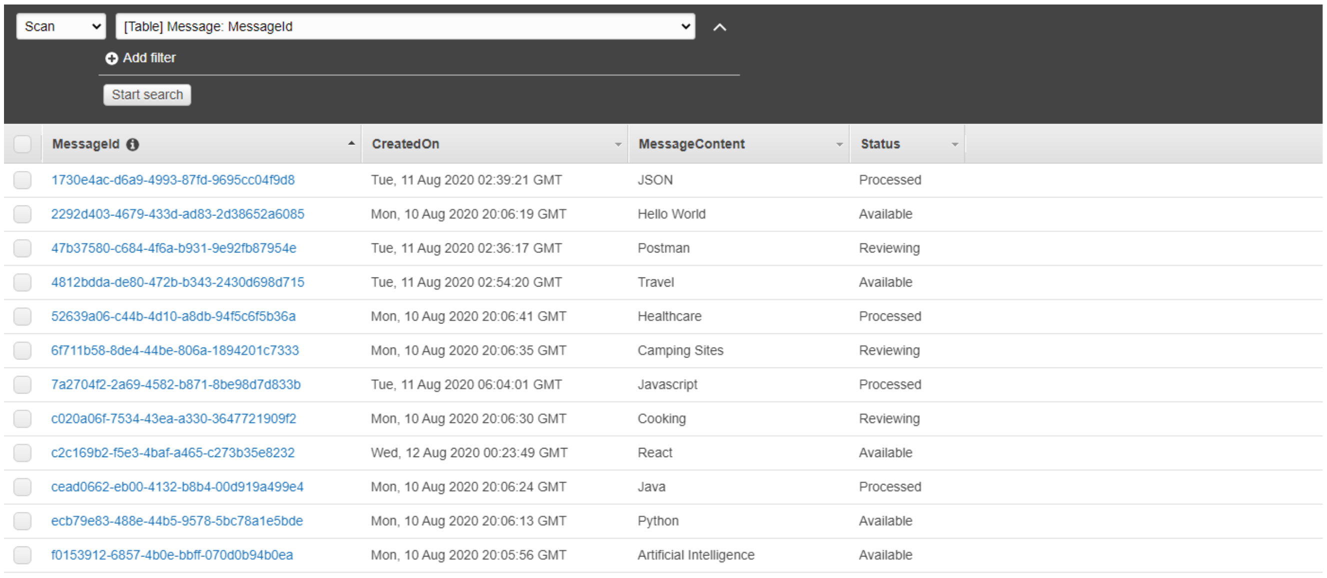
Write a message: Test

Your message has been recorded under ID: eafb5a73-d74f-4a00-b5bf-8f9c20f9eaa8

Consumer

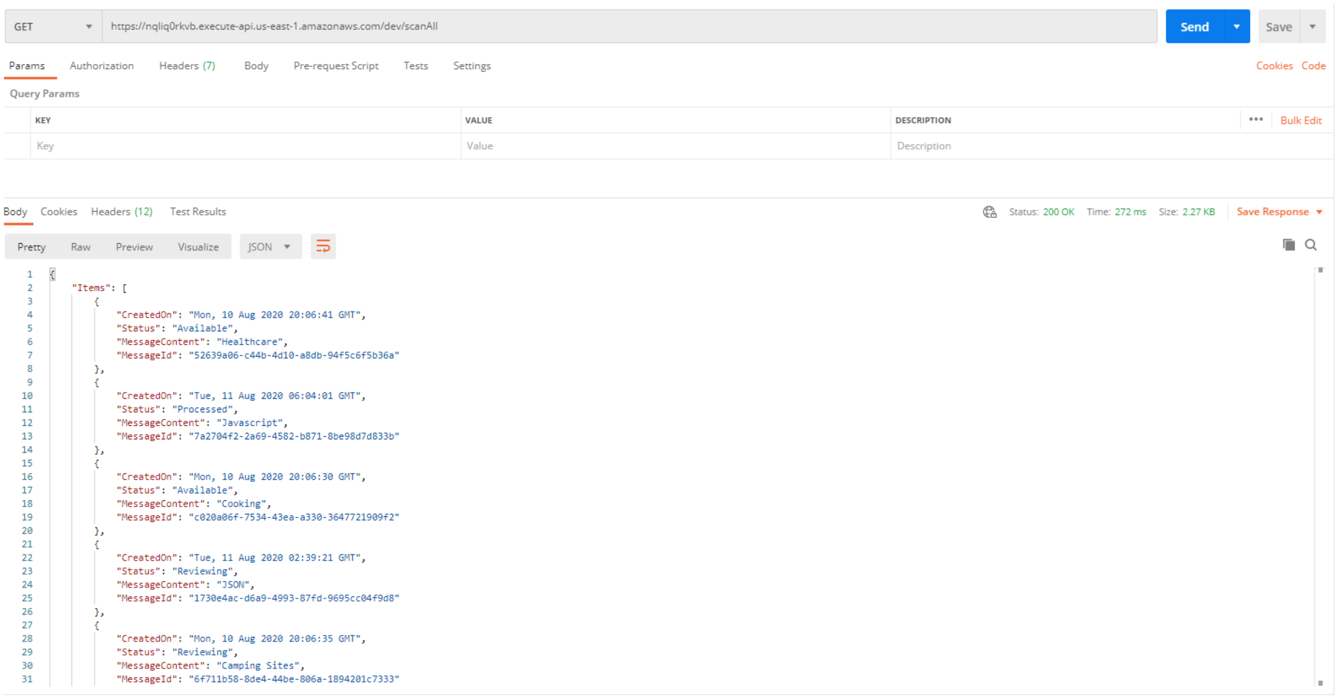
Pick the items you want to work on.

Cooking ☐
Camping Sites ☐
Postman ☑



Scan | [Table] Message: MessageId

	MessageId	CreatedOn	MessageContent	Status
	1730e4ac-d6a9-4993-87fd-9695cc04f9d8	Tue, 11 Aug 2020 02:39:21 GMT	JSON	Processed
	2292d403-4679-433d-ad83-2d38652a6085	Mon, 10 Aug 2020 20:06:19 GMT	Hello World	Available
	47b37580-c684-4f6a-b931-9e92fb87954e	Tue, 11 Aug 2020 02:36:17 GMT	Postman	Reviewing
	4812bdda-de80-472b-b343-2430d698d715	Tue, 11 Aug 2020 02:54:20 GMT	Travel	Available
	52639a06-c44b-4d10-a8db-94f5c6f5b36a	Mon, 10 Aug 2020 20:06:41 GMT	Healthcare	Processed
	6f711b58-8de4-44be-806a-1894201c7333	Mon, 10 Aug 2020 20:06:35 GMT	Camping Sites	Reviewing
	7a2704f2-2a69-4582-b871-8be98d7d833b	Tue, 11 Aug 2020 06:04:01 GMT	Javascript	Processed
	c020a06f-7534-43ea-a330-3647721909f2	Mon, 10 Aug 2020 20:06:30 GMT	Cooking	Reviewing
	c2c169b2-f5e3-4baf-a465-c273b35e8232	Wed, 12 Aug 2020 00:23:49 GMT	React	Available
	cead0662-eb00-4132-b8b4-00d919a499e4	Mon, 10 Aug 2020 20:06:24 GMT	Java	Processed
	ecb79e83-488e-44b5-9578-5bc78a1e5bde	Mon, 10 Aug 2020 20:06:13 GMT	Python	Available
	f0153912-6857-4b0e-bbff-070d0b94b0ea	Mon, 10 Aug 2020 20:05:56 GMT	Artificial Intelligence	Available



GET https://nqliq0rkvb.execute-api.us-east-1.amazonaws.com/dev/scanAll

KEY	VALUE	DESCRIPTION
Key	Value	Description

Status: 200 OK Time: 272 ms Size: 2.27 KB

```
{
    "Items": [
        {
            "CreatedOn": "Mon, 10 Aug 2020 20:06:41 GMT",
            "Status": "Available",
            "MessageContent": "Healthcare",
            "MessageId": "52639a06-c44b-4d10-a8db-94f5c6f5b36a"
        },
        {
            "CreatedOn": "Tue, 11 Aug 2020 06:04:01 GMT",
            "Status": "Processed",
            "MessageContent": "Javascript",
            "MessageId": "7a2704f2-2a69-4582-b871-8be98d7d833b"
        },
        {
            "CreatedOn": "Mon, 10 Aug 2020 20:06:30 GMT",
            "Status": "Available",
            "MessageContent": "Cooking",
            "MessageId": "c020a06f-7534-43ea-a330-3647721909f2"
        },
        {
            "CreatedOn": "Tue, 11 Aug 2020 02:39:21 GMT",
            "Status": "Reviewing",
            "MessageContent": "JSON",
            "MessageId": "1730e4ac-d6a9-4993-87fd-9695cc04f9d8"
        },
        {
            "CreatedOn": "Mon, 10 Aug 2020 20:06:35 GMT",
            "Status": "Reviewing",
            "MessageContent": "Camping Sites",
            "MessageId": "6f711b58-8de4-44be-806a-1894201c7333"
```