10-k Itemization Backend V2.0

Overview

A REST API abstracting the complexity of itemizing a 10k filing and managing storage of processed filing content.

Functional Requirement

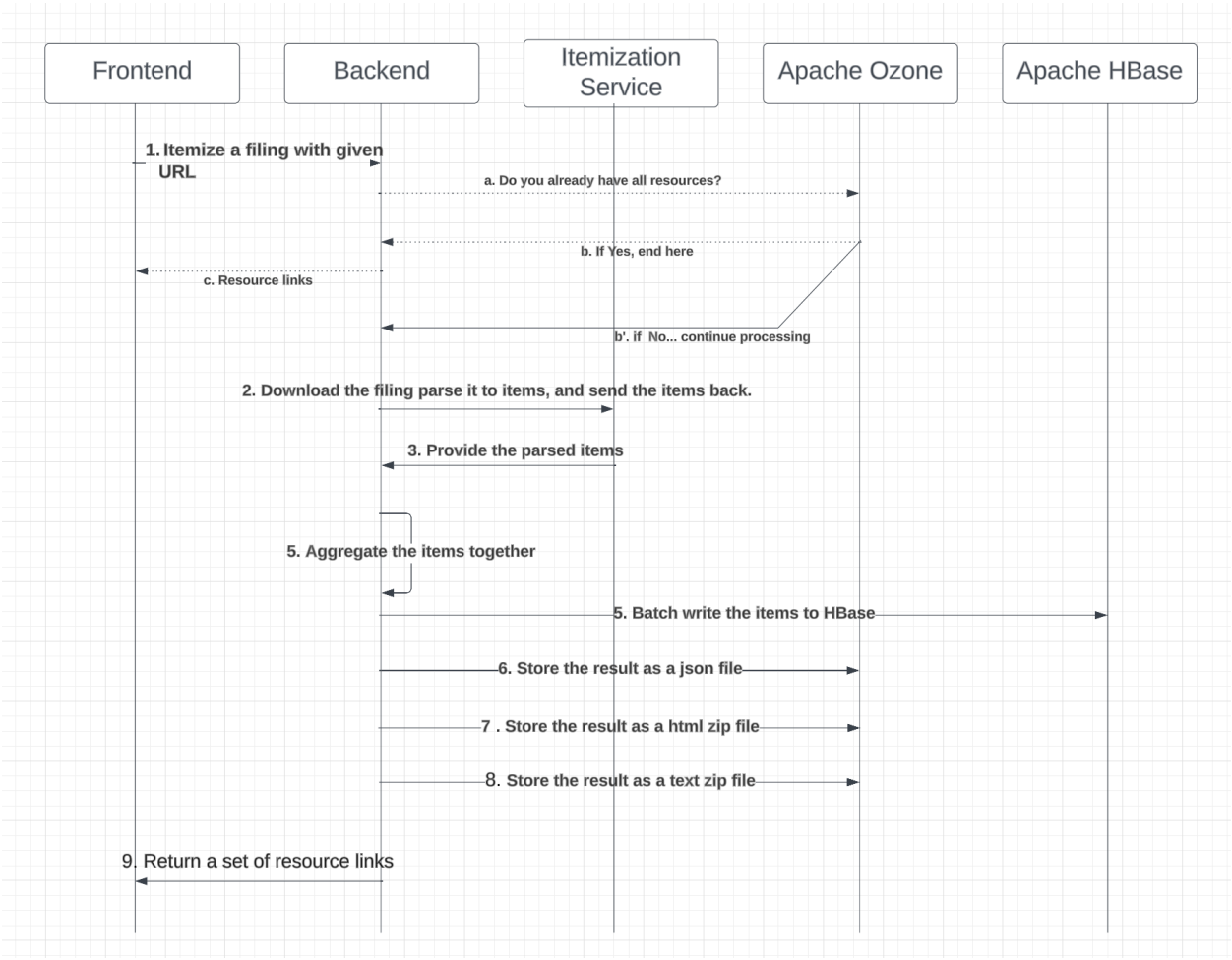
1. Being able to extract, process, and persist itemized filing items given an SEC filing URL.
2. Able to download filing resources given a downloading request. The possible resource could be a JSON file used to render the page on the Frontend side; a zipped plain text formatted filing item; or a zipped html formatted filing item.
3. A mapping between the company ticker name(e.g.: AAPL-2021) chosen by the Frontend, and the corresponding SEC filing URL.

Critical User Journey(CUJ)

1. After a user enters the page, show a list of tickers that maps to a set of sample filings URLs. User can choose any one of them to view the itemized filing. During this process, the Frontend will fetch the mapping between a company ticker and a 10-k filing URL. Then store this mapping so that it can send the correct filing URL corresponding to the ticker selected by an end user to the Backend.

2. User can either provide a filing url, or select one of the sample tickers to enter the itemized filing display page. During the process, the Frontend side would first call the Backend to extract and process itemized filing resources, then persist them and provide a set of resource links to the Frontend. Immediately after getting the resource links, the Frontend would download the JSON file resource, parse them, and render them to pages. Each page represents an item of a particular filing. This action will be performed every time a particular itemized filing is to be displayed on the Frontend.

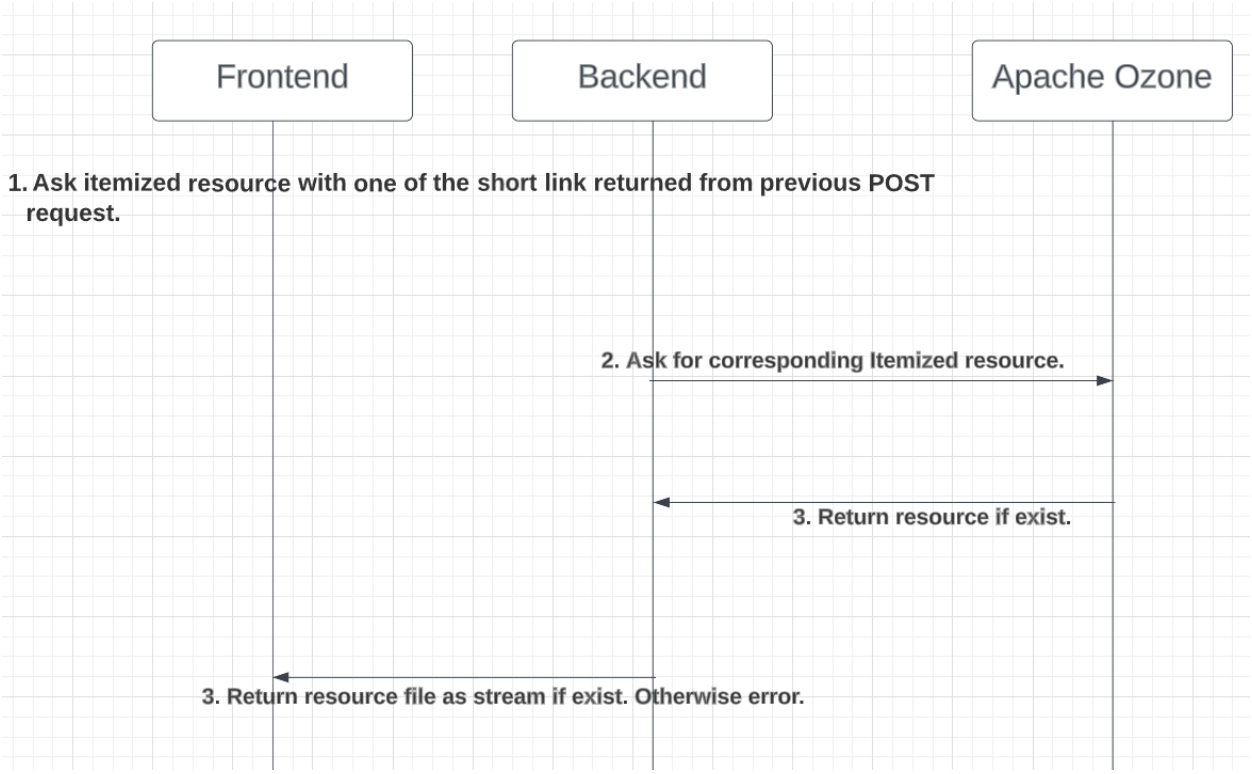
   Below is the workflow on the backend side during this process.



Note: Step 5-8 can be done in parallel to reduce the time waste caused by blocking IO.

3. User will have the option to download the entire itemized filing content. The optional filing resource types are text and html file. To perform this flow, the Frontend would send a downloading request to the Backend using the resource link returned from the backend in CUJ 2.

   Below is the workflow on the backend side during this process.



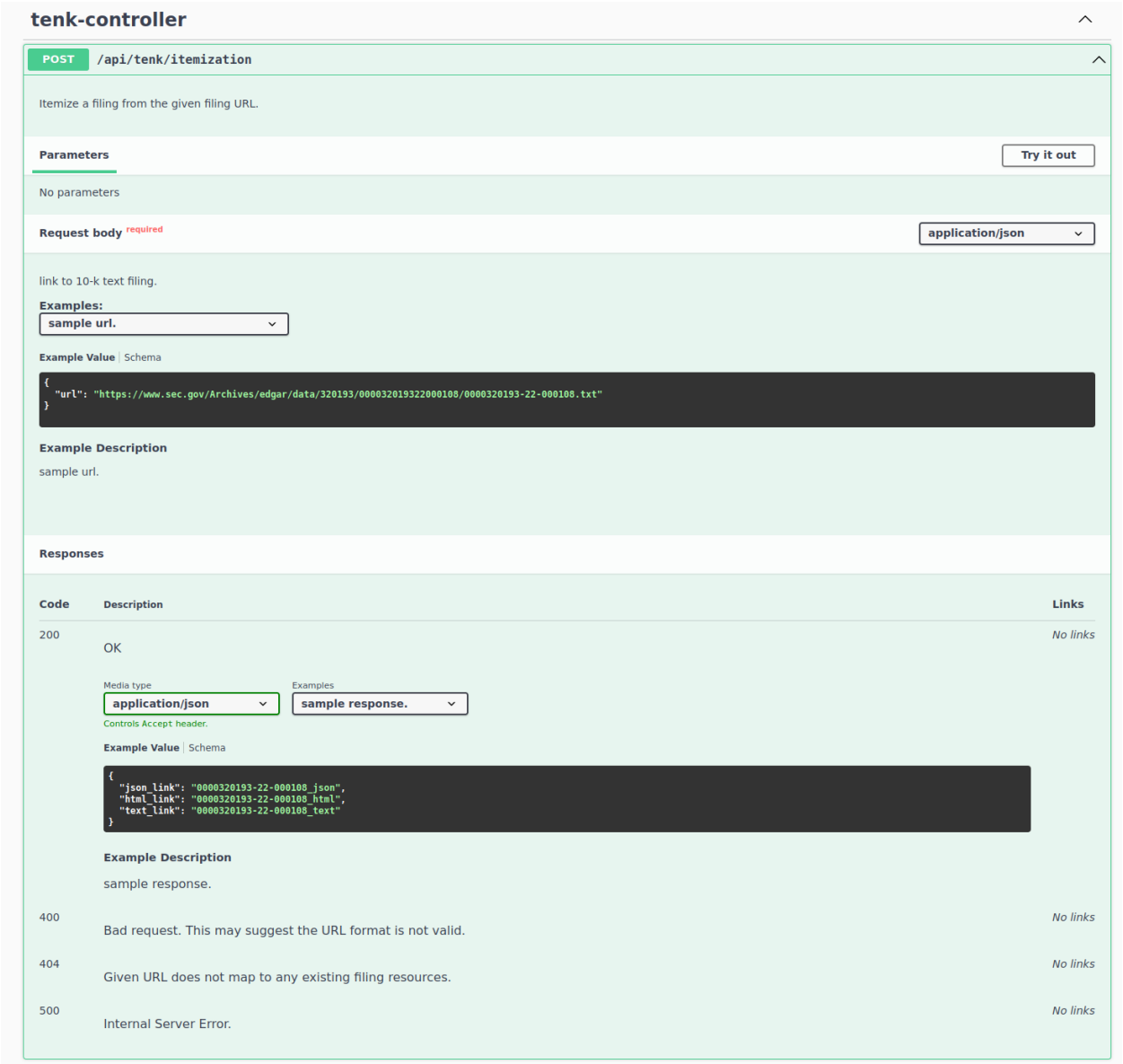
API

Download filing, Itemize, and store it at the background given a filing URL

```
REQUEST
        POST api/tenk/itemization

        Body: {“url”:”${tenk-filing-url}”}
—---------------------------------
RESPONSE
Body: {“json_link”:”xxx”, “html_link”:”xxx”, “text_link”:”xxx”}
```

Check below for an example:



Get sample filing links from ticker name

```
GET api/tenk/samples
——————————————————————
RESPONSE
Body: (a sample map)

```

Get itemized filing content

As a text zip file

```
GET api/tenk/sample-form?filing_name=${text_link}
—----------------------------------
RESPONSE
Body: a byte stream of a zip file

```

- text_link is the one returned in this section.

As a HTML zip file

```
GET api/tenk/sample-form?filing_name=${html_link}
—----------------------------------
RESPONSE
Body: a byte stream of a zip file

```

- html_link is the one returned in this section.

As a JSON items stream

```
GET api/tenk/sample-form?filing_name=${json_link}
—----------------------------------
RESPONSE
Body: a byte stream of a JSON file

```

- json_link is the one returned in this section.

Storage

(html/text)Itemized Filing zip file

Storage Choice: Ozone
Schema:

<filing_name>_(html|text).zip: file

Item Table

Storage: HBase
Schema:

rowkey: {filing_name}#{part_number}#{item_number}
columns:
    content: html
    content: text

The improvement compared with V1

- The algorithm is packed into another service(docker) and is triggered by RPC, the app and the algorithm can be deployed separately. There is no more git module issue.
- IO blocking time is reduced by running several tasks in parallel.
- API has less confusing field names in the response JSON.
- S3 is replaced with Ozone, so we can control all resources on our side without paying AWS.
- The Frontend request flow is simpler. Now, the only entity that can interact with object storage is the Backend server. So all the request is between the Frontend and the Backend. This change reduces downloading speed but gives much better control and hides the object storage behind, so we don't need to worry about setting a gateway for it.

Assumption

- Filing Data are stored in docker temporarily since they can be produced at any time if they are not persisted.
- The biggest processed filing is below 1G.
- The amount of storage space needed for this app(at this stage) is negligible.
- We would have another Functional Request that read a particular item from another HBase that has data already stored very soon (current pattern is url→itemized form). This service then will use that HBase as the persistence layer. Also we should also develop at least the DAO interface for this coming request. Even though at this moment, HBase is actually not needed.

- Ozone adapts well to the Hadoop ecosystem. Thus we'd choose it as the Object Store. We'd develop the solution now using the ozone docker cluster, and later we would migrate the storage there once our own Ozone Cluster is set up.

- As long as a resource is not found in Ozone, it is considered missing and will return 404 to Frontend. It is expected that the user will manually reload the page and trigger another POST /itemization request to fetch the resource.

- The query load is negligible(max QPS < 10). And users can wait for 8-12s to get itemized responses during the POST request execution (this only happens if the resource is not stored in the Object Storage). The connection will keep open during the waiting time, and there is no need for a message queue.

- No Read/Write conflict since those 10-k filings are meant to be read-only. Once they are published on SEC, it keeps unchanged.