**1. Product Requirements:**

This document outlines the requirements for the following

1) Content (Image) storing with durability

2) Resizing the images to a standard pixel size.

3) Retrieval of the images uploaded by the user.

**2. Functional & Non-functional specifications :**

**2.1 Functional Requirements :**

**2.1.1 Image Input and Output**

* **Input**: The service will receive images. Images may be in pre-configured but changeable formats such as JPEG, PNG, etc.
* **Output**: The service will save resized images to HDFS/or any replicated storage. Each resized image will be stored in a specified directory.

**2.1.2 Image Resizer**

* **Functionality**: The service will resize images to standardized dimensions (width and height).
* **Supported Dimensions**: The service should support resizing to a fixed and/or dynamic dimension.
* **Format**: Resized images should be saved in the same format as the original or a user-specified format.

**2.2 Non-Functional Requirements :**

**2.2.1 Performance**

* **Scalability**: The service should handle large volumes of image storing and processing efficiently, leveraging distributed processing.
* **Latency**: Retrieval and Resizing operations should be completed within a SLA bound time frame, considering the size and number of images.

**2.2.2 Security, Availability & Durability**

* **Access Control**: Only authorized users should be able to request image
* **Data Privacy**: Ensure that image data is handled securely and complies with data protection regulations.
* **The data(image and image metadata) once uploaded or saved must not be lost i**.e. it must be replicated to always have a copy in a different geo-location /file-server.

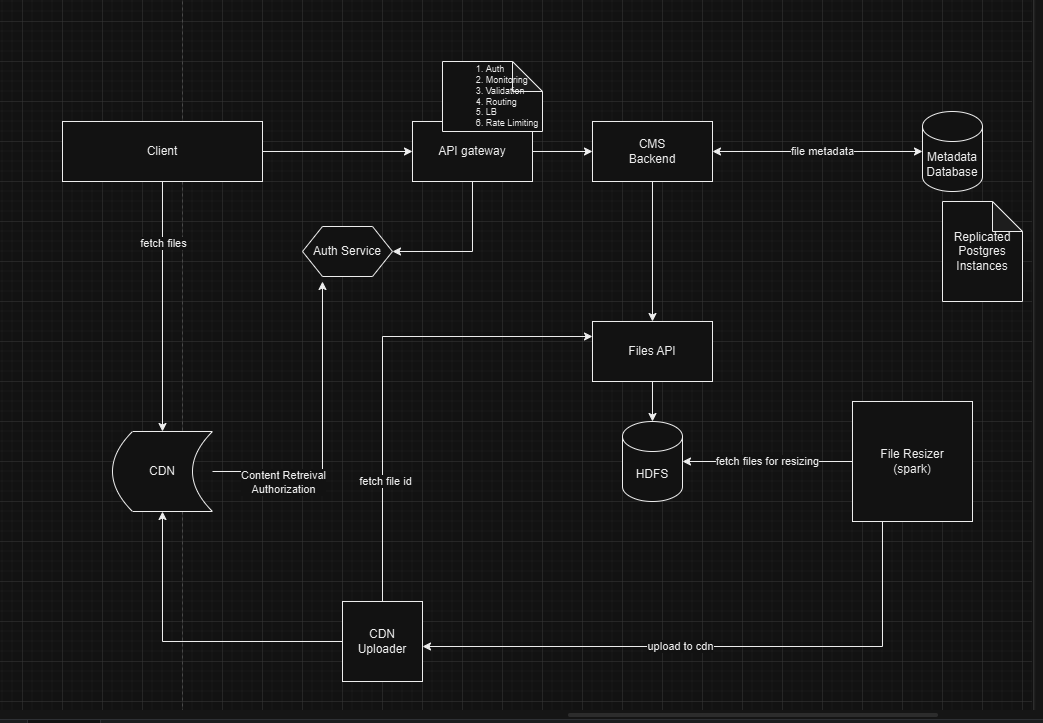
**2.2.3 Usability**

* **API Interface**: The service should provide a RESTful API for initiating image uploads and downloads.
* **Documentation**: Provide clear documentation for the API endpoints, request parameters, and error codes.

**3. User Stories**:

* **Image upload:** As an actor user, the upload of the image is attempted from a web interface.
* **Image retrieval:** As an actor user, he/she collects all the images uploaded / has an access to.
* **Processing of the images:** As an actor system, the system detects new image uploads or the pending images to be resized and processes them
* **Moving the images to CDN:** The system implicitly moves the images to the CDN close to the user for future retrieval.

High Level Design Diagram:



**High-Level Design (HLD) Considerations:**

1. Users upload images through the Gateway.
2. The implementation should be extensible to handle more file formats and file operations too(if & when required).
3. Users interact with the CDN, which delivers content from the nearest edge. The CDN forwards all requests to the Gateway, which then communicates with the Auth service to validate requests according to security policies.
4. The API Gateway handles routing, authorization, content validation, load balancing, and rate limiting for backend servers.
5. The Content Management Service (CMS) manages file uploads and saves files to user-specified folders.
6. The CMS also facilitates content delivery through the Gateway.
7. The CMS stores content metadata (e.g., file info, user info, file location on HDFS/file system) in a persistent, replicated database for disaster recovery and read scalability, as we forward the read load to read replicas and write load is sent only to the leader.
8. The Files API manages file operations, including storing and retrieving files from HDFS. It serves as an interface, allowing for future transitions to different storage types like Amazon S3.
9. The Image/File Resizer processes image resizing as asynchronous jobs after images are saved. It can use Spark for parallel processing to meet Sla requirements.
10. The CMS submits jobs to the File Resizer, which retrieves job details from the metadata database.
11. To handle potential spikes in upload volume, the process is designed to be asynchronous. Users can complete uploads without delay, while resizing is performed in the background. This approach ensures that if resizing fails, it can be retried without impacting user experience.
12. In our Postgres leader-follower setup, we need to implement leader failover to switch to a new leader if the current leader becomes unavailable. As, we do synchronous replication we made sure each write happens to at least two (Defined N nodes) to ensure data availability.

LLD ( Low Level Design and Flow charts) for figure Flow Chart 2.0

Constraints & scope

* 1. We have interface to handle file operations , since we don’t have s3 access , we use this interface to work with NTFS
  2. The container has a local mounted volume form the host ( windows) to which all files are written
  3. We have not handled the situation of file replication
  4. We use a single postgres instance ( the follower is not the part of the demo )
  5. We do not have the CDN thus, all calls to download content go through CMS and then to the fileserver service to deliver the file
  6. The largest file upload able is 50MB
  7. The system only handles images but then we can allow different file extensions my the config change
  8. The resizer code only picks the images and does not process the pdf for which we have to have different staretgies to process.
  9. There is no oAuth integration thus there is no way to find out which use uploaded the image and which user has requested to download the image
  10. We have not impelemented the Gateway service.
  11. As There is no CDN used, the CDN uploader is not implemented
  12. To scale out the storage , we need to depend only on the fileService which can be implemented using s3, NetApp, HDFS
  13. We could have used spark if we had used HDFS to do the batch processing of resizing the files but currently it is handled through a job .
  14. The responses are not compressed but it is a good way to send the compressed image content to give better response times of download and reduced latency
  15. We could have impleneted the jobs to re work on the imaged which were failed inresizing taks but we have considered doing it to not miss on the failures of resizing.

This could be done to store the time when processing begin and the current status of the file .If the time elapsed is > 1 hour ( configurable ) and the status is is\_being\_being the file should be re picked for processing.Each processing should happen in a new thread as if one file could not be processed, the next in the list has no impact as it takes its own thread stack.

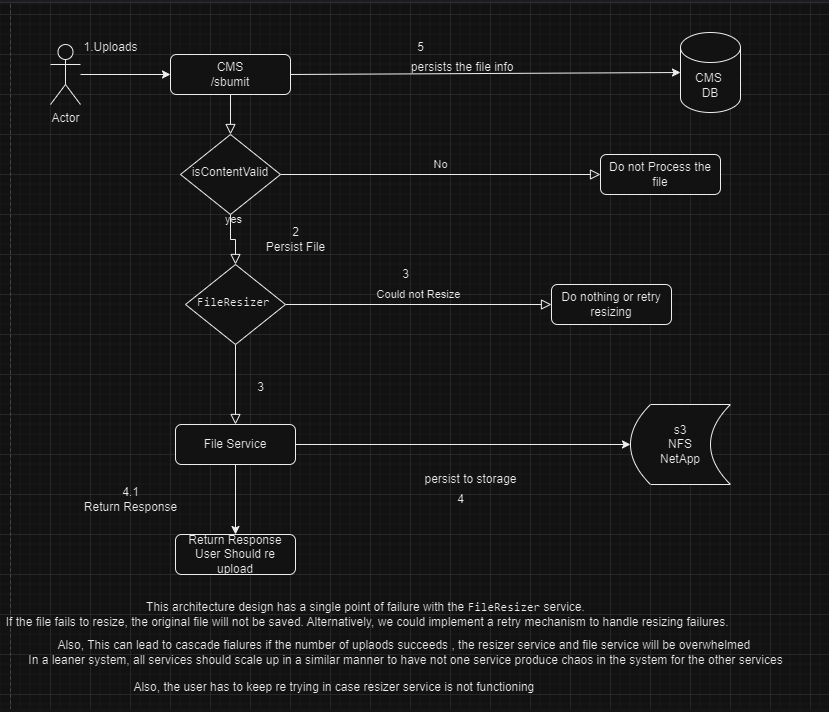
APIs

|  |  |  |
| --- | --- | --- |
| API | use | returns |
| /submit | Receives an image, validates,saves | Status of the upload |
| /list | Returns all the files info(with thumbnails) | Returns all the files info(with thumbnails) |

1. **User Interaction:**
   * User uploads an image through the UI's /submit route.
2. **CMS Service:**
   * Receives the image upload request.
   * Generates a unique identifier for the image.
   * Stores basic image metadata (e.g., filename, upload time) in the database.
   * Sends the image and metadata to the FileService.
3. **FileService:**
   * Stores the image in the storage system (HDFS, S3, NFS).
   * Returns a RemoteFile object containing storage details.
4. **CMS Service (Continued):**
   * Updates image metadata with the RemoteFile information.
   * Creates a new FileMetadata record with initial state (unprocessed).
   * Returns a success response to the user.
5. **Job Scheduler:**
   * Triggers the Resizer and Thumbnail Creator jobs at configured intervals.
6. **Resizer Service:**
   * Picks unprocessed images from the database.
   * Resizes the image using the RemoteFile information.
   * Updates the FileMetadata with the resized image location.
7. **Thumbnail Creator Service:**
   * Picks images without thumbnails from the database.
   * Creates a thumbnail using the RemoteFile information.
   * Updates the FileMetadata with the thumbnail location.
8. **CMS Service (List API):**
   * Retrieves paginated list of FileMetadata.
   * Includes thumbnail information (if available).
   * Returns the list to the user.
9. **CMS Service (Image Download API):**
   * Retrieves FileMetadata based on image ID.
   * Calls FileService to fetch the requested image (original or resized).
   * Returns the image to the user.

**Key Differences and Improvements**

* **Job Scheduler:** Explicitly shows the scheduling mechanism for Resizer and Thumbnail Creator jobs.
* **Asynchronous Processing:** Emphasizes the independent nature of image processing tasks.
* **Metadata Management:** Clearly outlines the role of FileMetadata in tracking image state and locations.

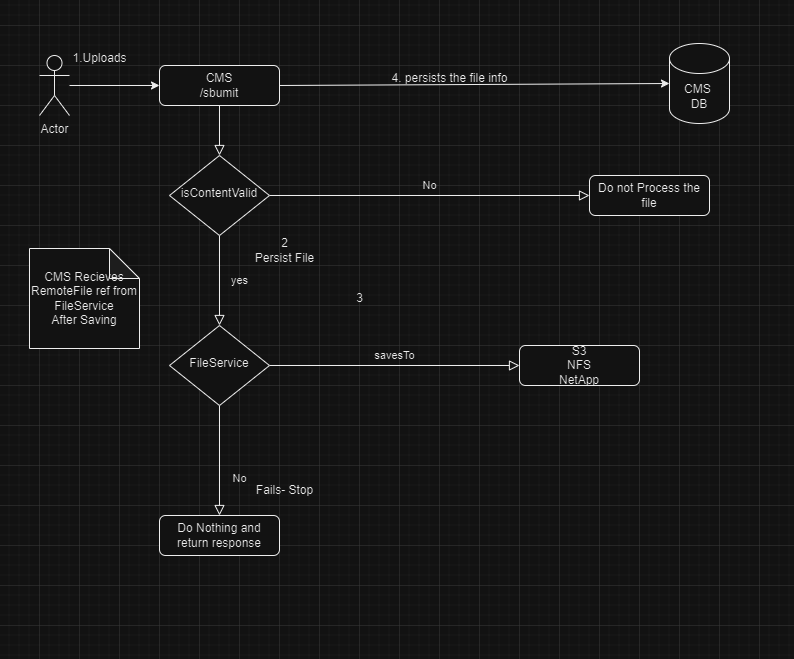
FLOW CHART 1.0 ( has single point of failures)  


1. **User uploads a file:** The process starts with a user uploading a file.
2. **File validation:** The system checks if the uploaded file is valid.
3. **File storage:** If valid, the file is saved to a storage service like S3, NFS, or NetApp.
4. **File resizing:** The system attempts to resize the file as a sync op.
5. **Error handling:** If resizing fails, the system can either retry or do nothing.(currently re trying I not implemented as we chose another architecture that has re tries and is robust)
6. **Response:** Finally, the system returns a response to the user, indicating success or failure.

**Issues with this design:**

* **Single point of failure:** The file resizer is a single point of failure, meaning if it fails, the entire process might be affected.
* **Cascading failures:** If many uploads occur simultaneously, the resizer and file services might become overwhelmed, leading to system instability.
* **User experience:** Users might need to retry uploads if the resizer is not working correctly.

FLOW CHART 2 .0 (improved)



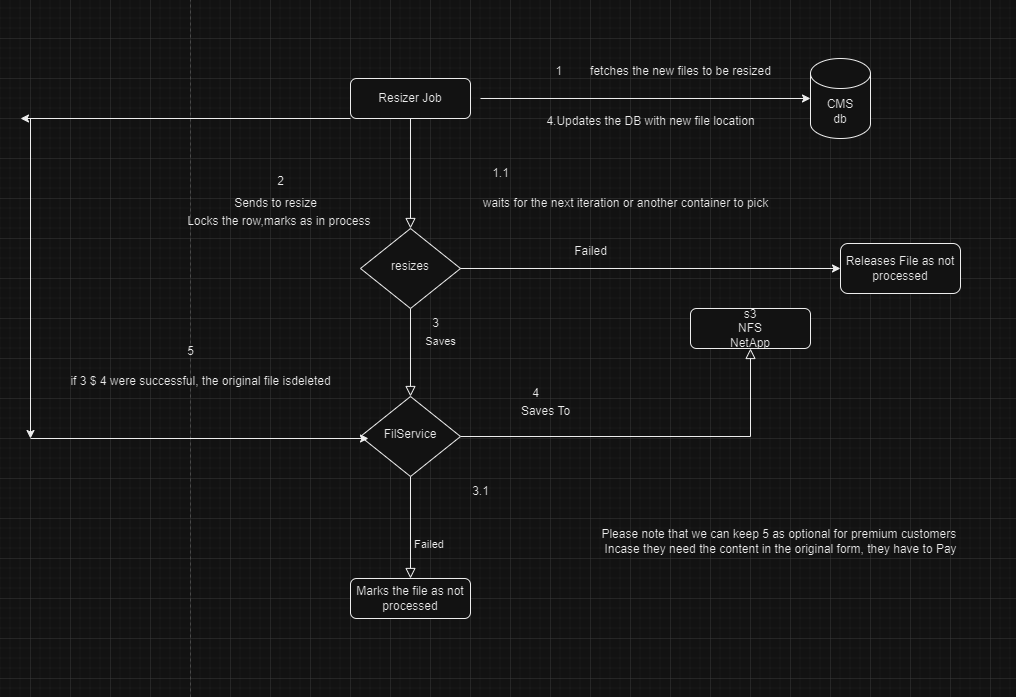
**Here's how it works:**

1. **User uploads a file:** The process starts with a user uploading a file.
2. **File validation:** The system checks if the uploaded file is valid.
3. **File storage:** If valid, the file is saved to a storage service like S3, NFS, or NetApp.
4. **File resizing:** The system attempts to resize the file.
5. **Error handling:** If resizing fails, the system can either retry or do nothing.
6. **Response:** Finally, the system returns a response to the user, indicating success or failure.

**Issues with this design:**

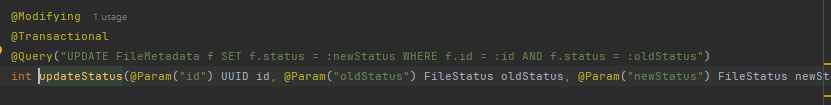
* **Single point of failure:** The file resizer is a single point of failure, meaning if it fails, the entire process might be affected.
* **Cascading failures:** If many uploads occur simultaneously, the resizer and file services might become overwhelmed, leading to system instability.
* **User experience:** Users might need to retry uploads if the resizer is not working correctly.

Resizer Job flow chart



The flow chart above illustrates the process of the Resize Job:

1. The service retrieves all files that have not been resized.
2. Since this is a distributed system, some files in the list from step 1 might have been processed by other containers running the resize operation. To prevent multiple containers from processing the same file, we update the row to indicate it is being processed. The update record ensures that no other container picks the file if it is still in the list from step 1. The update query performs the update only if the status has not changed, using the following query:

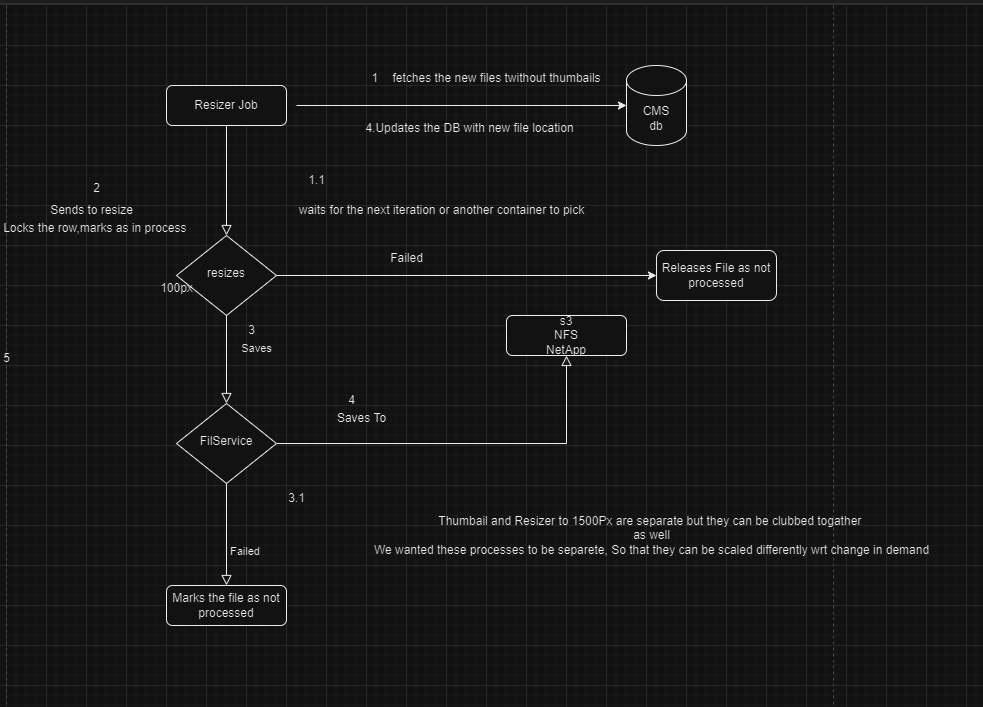


To further safeguard against concurrency issues, you can acquire a pessimistic row lock. If you cannot claim a file because it is already claimed by another container, the system moves on to the next file in the list for resizing.

I considered sending events/messages to the resizer to pick image IDs for resizing. However, I was concerned about ensuring ACID properties and guaranteeing that no file would be missed. Using an RDBMS system table as a queue for job processing provides assurance that records are neither missed nor processed multiple times.

**The downside of this approach is that job containers need to continuously poll for new records, which results in increased database queries and interactions.**

Thumbnail creator Job flow chart



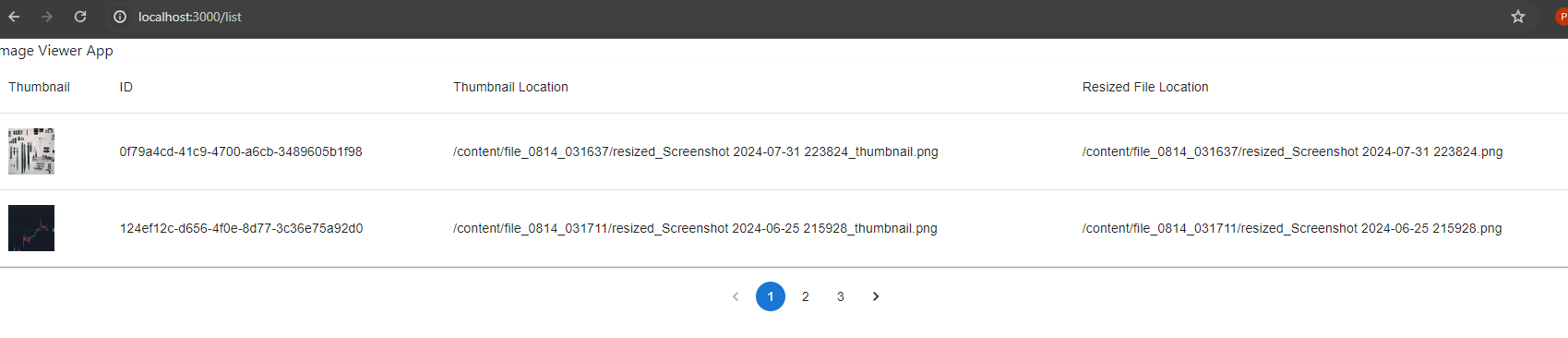
Below is the flow of the thumbnail generator , we used it to create thumbnail for each image to not let the full images to get streamed in the first go . The user can choose the thumbnail he wants to view from the paghinaged table shows below

1. **Fetch new files:** Files without thumbnails are retrieved.
2. **Lock and process:** The file is locked for processing, and the process begins.
3. **Resize:** The image is resized to 100px.
4. **Save:** The resized image is saved to a file service (S3, NFS, or NetApp).
5. **Update DB:** The database is updated with the new file location.

**Thumbnails and On-Demand Fetching:**

Creating thumbnails offers several advantages:

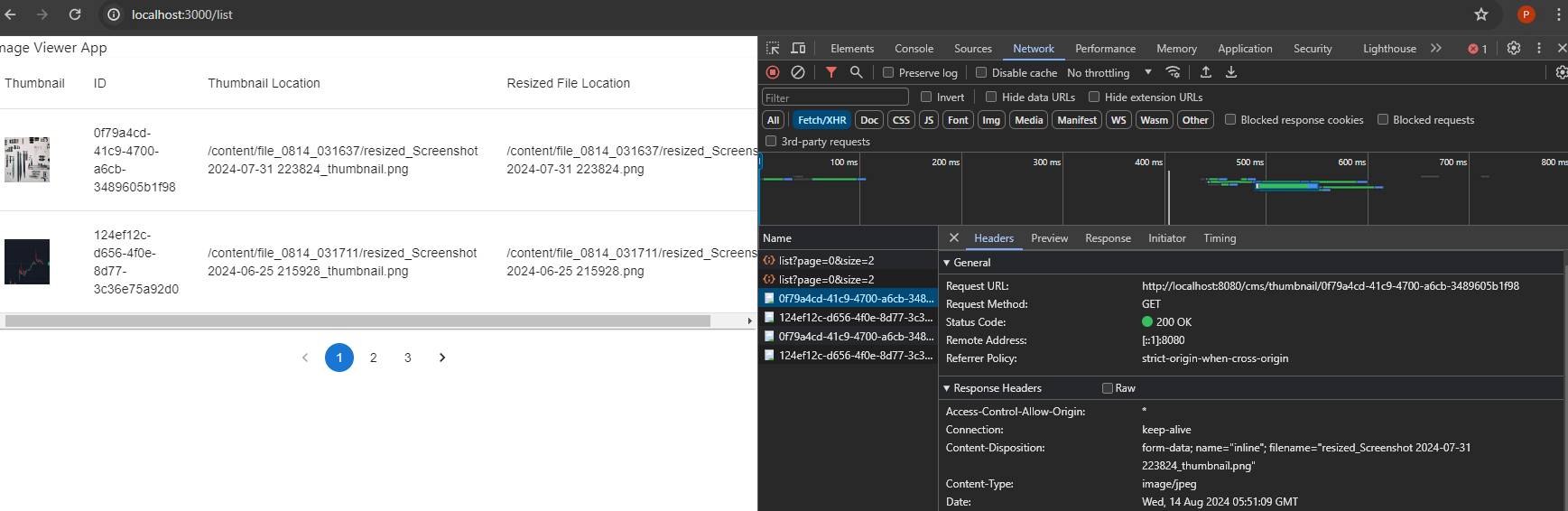
* **Faster load times:** Thumbnails are smaller, loading quicker on user devices.
* **Reduced bandwidth usage:** Smaller thumbnails consume less network data.
* **Improved performance:** Lower resource consumption on servers and networks.
* **Better user experience:** Usr can quickly scan through images in the react table and decide which to view in full size.



The above table is build from the below response from GET /list api , we retuned the **paginated** info

Additonally the ui component gives the call to fetch the thumbnails for each row in the table which is a smaller foot print on the network and backend service



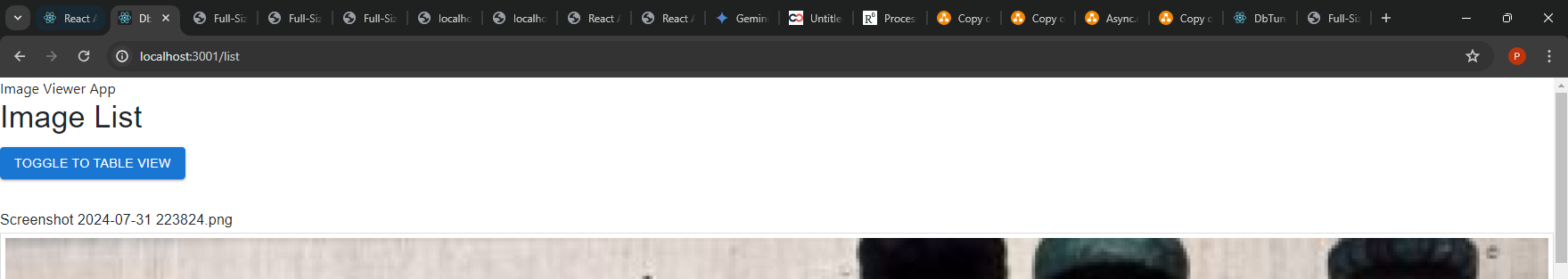


once the user clicks on the thumbnail , Another call goes to the backend to fetch the full resized image

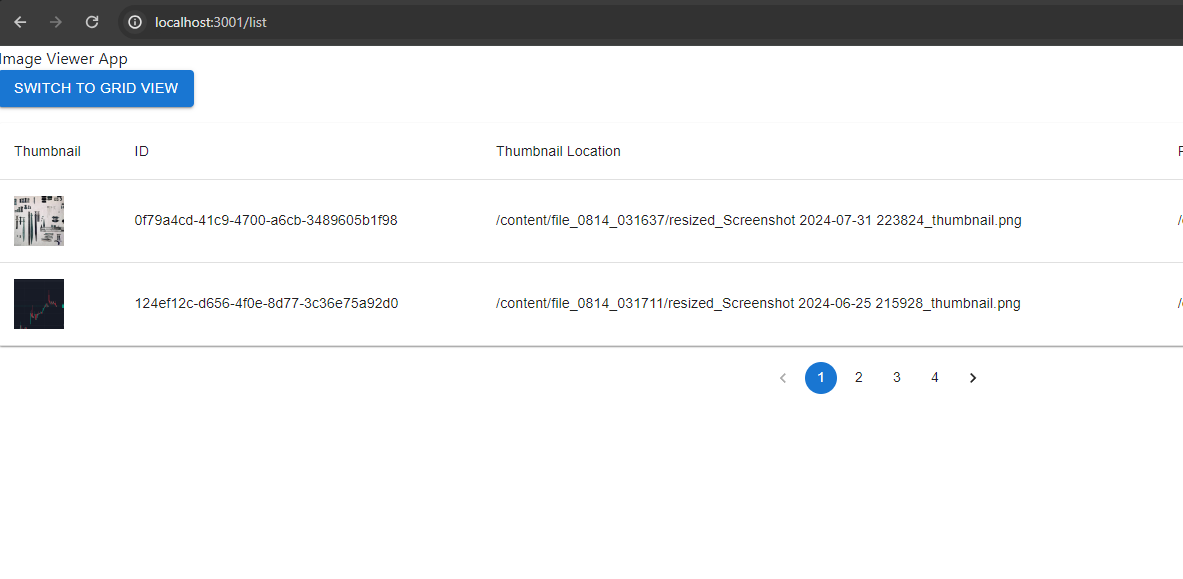


To adhere to the api asked for if we don’t want to fetch data on demand (but only paginated and not thumbnails) we show the data as Grid too where the api returns the image stream in a single response but paginated . The final ui will have a route /list but will have a toggle button if we want to show them asa grid of in a table with thumbnails first and then the user can request the image (1500 pixels on demand) but we feel that thumbnails are a better way to go in terms of scalability and keep the system light as well. So that we don’t over do as part of routine renders.  
  


Grid view



When switched to table view.



On clicking on the thumbnail, it downloads the actual & full imnage which is a resized image

