Upload in Nuxeo

Requirements

When uploading files to Nuxeo, we have the following requirements :

- upload progress must be trackable client side
  - server side tracking is false because of reverse proxies
- upload should be done outside of transaction
  - otherwise transaction will timeout
- upload should be resumable
  - otherwise uploading large files over a broken connection could take days
- upload should check for duplicate
  - to avoid many users to upload the same file
- upload should avoid temporary storages when possible
  - to avoid copying the data multiple times
- upload should be cluster safe
  - in a cluster environment, the upload API should not depend on affinity aware load balancing

The first 2 requirements are currently handled by the `BatchManager` and the `BatchUpload` Endpoint.

This code was initially introduced for Android SDK and is now used by D&D, Nuxeo Drive and Automation API.

The 4 other requirements needs to be addressed.

BatchUpload improvements

Upload Resume / Chunking

Using chunking seems like a good idea since :

- this allows to manage resume with enough granularity
  - restart with chunk x
- this allows to manage multiplexing

  - upload on multiple TCP streams
- this allows to overcome the limitations of some reverse proxies
  - limits the risk of having a POST considered too big

The idea is to extend the existing API so that :

- client can upload only a chunk
- client can check if a chunk is already present

Associated tasks :

- NXP-16951
- NXP-16953

This is mainly a server side feature, but once completed, we need to use it :

- from Nuxeo Drive
- from Automation Clients (Java, JS, iOS ...)

Check for existing entry

We need to add again 2 APIs :

- `HEAD /upload/digest` : to check if binary exists
- `POST /upload` with a X-File-Digest header to "upload the file"

This second API is useful because we may actually want to create a document using a Blob that is already on the server, so we need to create an entry on the BatchManager.

This implementation should be simple.

Direct upload

How it works today The batch Manager stores the stream in temporary files.

These tmp files will be used to create Blobs that will be associated to a Document and that finally will be moved to the BinaryManager.

The final step will involve a copy of the file to write it inside the binary manager : the `storeAndDigest` will read / compute digest / write the stream to a file.

With S3 BinaryManager this read/digest/write will occur on a temporary store and then the file will be copied over S3.

What we may want to improve Client side upload is supposed to be limited by client side connectivity : S3/NAS access is likely to be faster than the http channel used by the client to upload the file.

So, we could leverage this to automatically have the batch manager write the content into the BinaryManager :

- no more file duplication
- no more slow write on S3 during end of transaction

Induced problems Doing so would create several issues

- BinaryManager would contain temporary streams
  - *this is actually not really an issue : we have GC for that*
- Chunking and Upload resume are a problem
  - BinaryManager resolve stream according to their digest : you can not find it without a digest
  - you don't have the Digest if you did not finish the upload

There are basically 2 approaches to solve this :

- Change the BinaryManager API to be able to manage chunks and temporary files
- Rely on client side logic
  - Make the client directly upload by it's own means to the backend (ex: using S3 API)
  - allow the Batch API to simply reference an existing Blob

Cluster Issue

The `BatchManager` handles a structure that is basically a list of streams that will be used within a transaction.

```
batchId => List<File>
```

If we add chunking, we have a more complex structure

```
batchId => List<List<FileChunk>>
```

This data (structure and streams) must be shared across Nuxeo nodes if we want the whole system to work across the cluster without having to enforce affinity.

The initial implementation being for JSF, this was not so much of an issue, but it is now critical since we are using this very same endpoint :

- for Nuxeo Drive
- for Rest API

This looks like a new use case for the `TransientStore` providing an abstraction API on top of :

- memory or Redis for the structure
- FS or S3 for the streams

See : NXP-17780