Storage

This document aims to describe the storage solutions to adopt for projects using the BBA standard. In the following, in a function signature, BBA is a data structure following the specifications given in other folders of this repository.

# Storage solutions to consider

## Local cache

As the BBA is hashed, it is easy to check its integrity. As a result, if a user had previously downloaded the asset content, it SHOULD be kept in cache for future usage.

The platform the application runs on defines the cache’s behavior. The application developer MUST take into account that a user CAN delete it. We can mainly identify the following platforms:

* Mobile (Android / iOS) that has an app specific storage
* Browsers, that have local storage
* Standalone PC applications (Unity, Unreal Engine, etc.) where a cache can be created in a temp folder (./BBA).

The library interface MUST implement the following function, which behaves according to the platform it runs on:

* BBA QueryAssetFromCache(string \_hash)

## Centralized server

Current blockchain projects use extensively centralized servers for their asset content. The usual workflow among developers is to use the *Token\_URI* function of the ERC721 optional metadata extension in order to reference a JSON metadata file, stored on their servers. Then, paths to binary data (e.g. images) are referenced in this JSON file.

The BBA standard is compatible with this workflow. In this case, the *Token\_URI* functionMUST reference either a valid URI for the BBA file itself, or a *baseURI*. The *baseURI* will be used to query asset hashes. E.g. If I want to query an asset with the hash 0xABC123, I would send a query to the *baseURI*.

**To discuss [Implementation detail): How do we define the specs for the Request for the hash query? (is it an HTTP Post Request to the server defined in *baseURI* with the payload being the hash?**

The library interface MUST implement both functions:

* BBA QueryAssetFromCentralizedServerWithURI(string \_URI)
* BBA QueryAssetFromCentralizedServerWithHash(string \_hash)

## Distributed storage

IPFS and Swarm both offer distributed storage solutions, where a resource can directly be queried from its hash.

Using these solutions for BBA storage means directly querying the hash of the BBA, without the need for peer discovery handling or additional informations.

The library interface MUST implement both functions:

* BBA QueryAssetFromIPFS(string \_hash)
* BBA QueryAssetFromSWARM(string\_hash)

## Users’ local machines

This solution requires more work from the developers of the application. Indeed, they would need to handle Peer to Peer sharing of BBAs.

The library interface MUST contain the function (to be overloaded by the developer):

* BBA QueryAssetFromOtherUsers(string \_hash)

# Additional library consideration

## Configuration

The library MUST contain the following functions that allow developers to configure the order in which the queries are made by default:

* SetQueryPriorities(Option \_options)
* BBA QueryAsset(string \_arg)

The Option struct MUST contain:

* A list of priorities for the different storage solutions
* Timeouts parameters to define when a specific query times out (so that the next storage solution is queried instead),
* **Additional configs?**

The developer SHOULD set the local cache at the highest priority.

## BBA Registry

The library MUST take under consideration the possibility to create on-chain BBA Registries. These registries map a hash to a storage location (be it centralized, on IPFS, etc.).

As a result, the library MUST contain the following functions:

* Registry[] GetRegistryListFromMasterRegistry()
* AddRegistry(Registry \_ registry)
* BBA QueryAssetFromRegistry(Registry \_registry, string \_hash)

The Registry strut MUST contain:

* The blockchain considered (**how do we id it? Probably a hardcoded list for now?**)
* The address of the registry on that chain
* **Additional configs?**