Library System
User Manual

Ingest, store and sync media across devices in a local network.

Contents

1. Getting Started 3
- 1.1 What is LibrarySystem? 3
- 1.2 What's included 3
- 1.3 Choosing your combination 4
- 1.4 Before you start: the closed-subclass rule 4
- 1.5 Quick-start checklist 5

2. How It Works 6
- 2.1 Architecture 6
- 2.2 Separation of responsibilities 6
- 2.3 Serve root layout 7

3. Using the Core Package 8
- 3.1 Implement the data interfaces 8
- 3.2 Build the ingest pipeline 9
- 3.3 Create the settings asset 12
- 3.4 Add the managers 13
- 3.5 Connect server and client 14
- 3.6 HTTP and "Allow downloads over HTTP" 14
- 3.7 Extra artifacts (e.g. thumbnails) 14

4. Media Server Backends 16
- 4.1 Comparison 16
- 4.2 Platform support 16
- 4.3 Setup (both backends) 16
- 4.4 Nginx details 17
- 4.5 Writing your own backend 17

5. Bridges 18
- 5.1 Comparison 18
- 5.2 NGO bridge 18
- 5.3 LAN bridge 19
- 5.4 Writing your own bridge 20

6. Platform Notes 21
- 6.1 Support matrix 21
- 6.2 Android 21
- 6.3 Windows 21
- 6.4 All platforms 21

7. Troubleshooting 22

8. Reference 23
- 8.1 Packages and requirements 23
- 8.2 Closed subclasses checklist 23
- 8.3 Public extension surface 24
- 8.4 Default values 24

1. Getting Started

1.1 What is LibrarySystem?

LibrarySystem is a set of Unity packages for ingesting, storing and syncing media files between a **host (server)** and its **peers (clients)**. The server runs an ingest pipeline on uploaded files, keeps a library manifest (`library.json`), and serves the files over HTTP. Clients fetch the manifest, compare it with their local copy, and download or delete files to match.

The core package is **project-agnostic**:

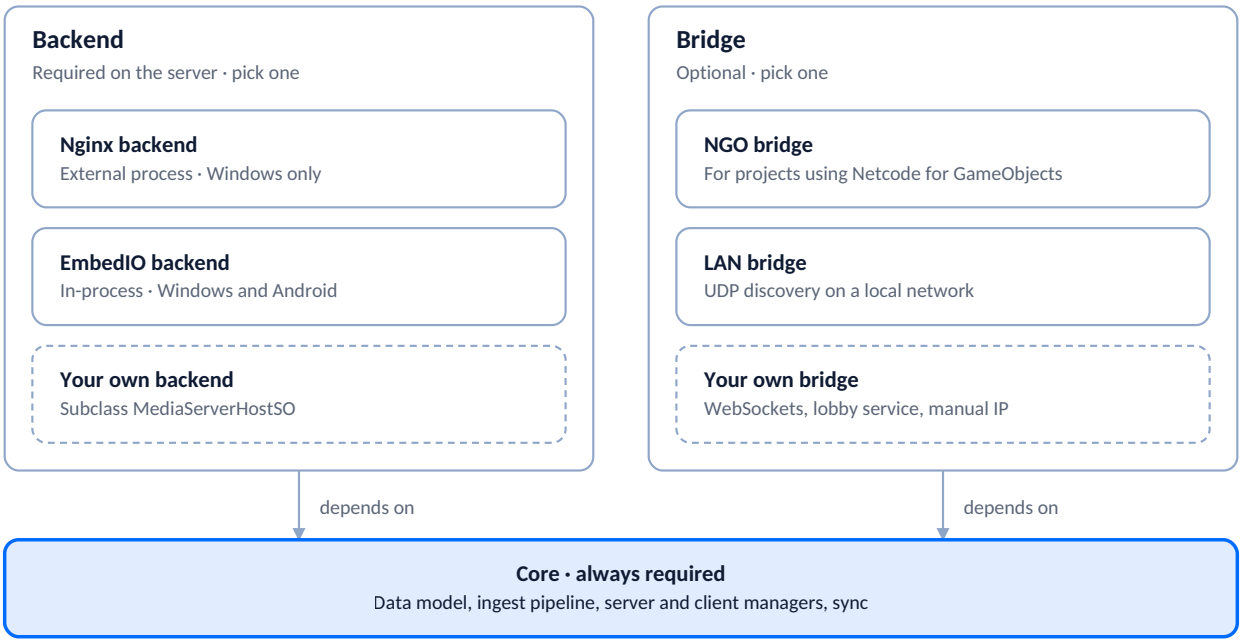
- It knows nothing about video, ffmpeg or any specific file format. Your project supplies those through a small set of interfaces and ScriptableObject assets.
- It has no required networking stack. The server and client managers work over plain HTTP and don't care how peers found each other.
- It ships no UI. You build your UI against the managers' public methods and events.

1.2 What's included

Package	ID	Role	Required?
Core	`com.genesiscreations.librarysystem.core`	Data model, ingest pipeline, server and client managers, sync	Yes
Nginx backend	`com.genesiscreations.librarysystem.backend.nginx`	Bundled nginx media server (Windows only)	One backend is required on the server
EmbedIO backend	`com.genesiscreations.librarysystem.backend.embedio`	In-process EmbedIO media server (Windows, Android)	One backend is required on the server
NGO bridge	`com.genesiscreations.librarysystem.bridge.ngo`	Connects the managers to a Netcode for GameObjects session	Optional
LAN bridge	`com.genesiscreations.librarysystem.bridge.lan`	UDP discovery and live sync on a local network, no networking stack needed	Optional

In practice, a project installs **Core + one backend + (optionally) one bridge**.

A project installs Core, one backend and, optionally, one bridge



1.3 Choosing your combination

Your situation	Backend	Bridge
Server runs on Windows only	Nginx or EmbedIO	—
Server runs on Android (or must run on both)	EmbedIO	—
Project already uses Netcode for GameObjects	—	NGO bridge
Local network, no networking stack, no internet	—	LAN bridge
Other stack (WebSockets, lobby service, manual IP entry)	—	Write your own bridge (§5.4)

1.4 Before you start: the closed-subclass rule

Everything in LibrarySystem is generic over `TMedia` (your media file type) and `TCategory` (your folder type). Unity cannot attach an open generic `MonoBehaviour` / `NetworkBehaviour` to a GameObject, and the Create Asset Menu cannot create an open generic `ScriptableObject`. So for every generic type you want to attach or create as an asset, you write a **one-line closed subclass** that fixes the type parameters:

```
public class YourServerLibraryManager : ServerLibraryManager<YourMediaFile, YourMediaFolder> { }
```

None of the packages ship pre-made closed subclasses. Each closed type must live in a file with the same name as the class. You'll see this pattern throughout the manual; §8.2 lists every one you may need.

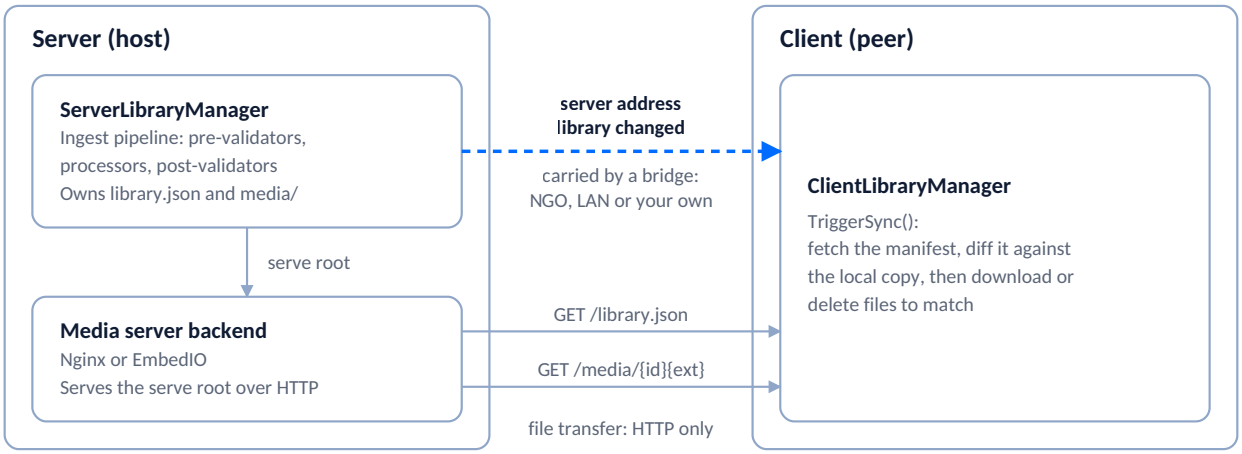
1.5 Quick-start checklist

1. Implement `TMedia` and `TCategory` (§3.1).
2. Write your validators and processors (§3.2).
3. Install a backend and create its host asset (§4).
4. Write a closed `LibrarySystemSettings` subclass, create the asset, and configure it (§3.3).
5. Write the closed server and client manager subclasses and add them to the scene (§3.4).
6. Connect server and client, with a bridge or your own code (§5).
7. Enable **"Allow downloads over HTTP"** on client builds (§3.6).
8. Build your UI against the managers' events and methods.

2. How It Works

2.1 Architecture

Files move over plain HTTP; the bridge only sends two small signals



The dashed line is the bridge: it carries the server address and a "library changed" signal. Every file travels over the solid HTTP lines.

2.2 Separation of responsibilities

Layer	Owns	Does not do
Core	Data model, ingest, manifest, folder layout under the serve root, sync logic	Serve HTTP, discover peers
Backend	Serving the serve root over HTTP (`Start(port)` / `Stop()`), deciding where the serve root lives	Know anything about the library's layout
Bridge	Telling the client the server's URL; telling clients when to re-sync	Transfer files

File transfer **never** goes through a bridge. The manifest is static JSON at `{serverBaseUrl}/library.json`, and media downloads use plain HTTP with Range support for resume at `{serverBaseUrl}/media/{id}{ext}`. Bridges carry only two small signals: "here's the server address" and "the manifest changed, re-sync".

2.3 Serve root layout

The core creates and owns this layout under the backend's serve root:

The core creates this layout under the backend’s serve root



The core creates these folders before the backend starts; the backend only serves them.

3. Using the Core Package

3.1 Implement the data interfaces

Namespace: `LibrarySystem.Core.Data`. Suggested location: `Assets/Scripts/YourLibrary/Data/`.

TMedia — `IMediaEntry<TMedia, TCategory>`

```
public interface IMediaEntry<TMedia, TCategory>
    where TMedia : IMediaEntry<TMedia, TCategory>
    where TCategory : class, ICategoryNode<TMedia, TCategory>
{
    string Id { get; }
    long SizeBytes { get; }
    string DisplayName { get; set; }
    string FileExtension { get; }   // includes the leading '.', e.g. ".mp4"
    TCategory Parent { get; set; }
}
```

- **`Id` is the on-disk filename** (`{Id}{FileExtension}`) on both server and client. It must be filesystem-safe and unique. A content hash is a natural choice, but any id that meets both rules works.
- Provide a **parameterless constructor**; manifests round-trip through Newtonsoft JSON.
- Mark `Parent` with `[JsonIgnore]`. Parents are rebuilt at load time by `AssignParentReferences()` to avoid reference cycles.
- No reflection constructor is needed. Only your own processors create `TMedia` instances.
- Implement `IHasExtraArtifacts` too if the type has extra files such as thumbnails (§3.7).

TCategory — `ICategoryNode<TMedia, TCategory>`

```
public interface ICategoryNode<TMedia, TCategory>
    where TMedia : IMediaEntry<TMedia, TCategory>
    where TCategory : class, ICategoryNode<TMedia, TCategory>
{
    string DisplayName { get; set; }
    List<TCategory> Children { get; }
    List<TMedia> Items { get; }
    TCategory Parent { get; set; }

    // Default interface methods — provided for you, do NOT re-implement:
    List<TMedia> Flatten();
    List<TCategory> FlattenCategories();
    void AssignParentReferences();
}
```

- **Do not re-implement** `Flatten`, `FlattenCategories` or `AssignParentReferences`. A class method with the same name is a second, divergent code path, not an override.

- **Required constructor:** `public YourMediaFolder(string name, YourMediaFolder parent)`. Both managers create folders with `Activator.CreateInstance(typeof(TCategory), name, parent)`. A wrong signature compiles fine but throws `MissingMethodException` at runtime, the first time a category is created.
- `Children` / `Items` are commonly explicit interface implementations backed by friendly serialized fields (e.g. `SubFolders` / `Files`).
- Mark `Parent` with `[JsonIgnore]`.

Example: a concrete `TCategory`. Note the explicit `Children` / `Items` implementations backed by `SubFolders` / `Files`, the `[JsonIgnore]` on `Parent`, and the two-argument constructor the managers call.

```
using System.Collections.Generic;
using Newtonsoft.Json;
using LibrarySystem.Core.Data;

namespace TestingLibrary.Data
{
    public class SharedFolder : ICategoryNode<SharedFile, SharedFolder>
    {
        public string DisplayName { get; set; }
        public List<SharedFile> Files;
        public List<SharedFolder> SubFolders;

        [JsonIgnore]
        public SharedFolder Parent { get; set; }

        List<SharedFolder> ICategoryNode<SharedFile, SharedFolder>.Children => SubFolders;
        List<SharedFile> ICategoryNode<SharedFile, SharedFolder>.Items => Files;

        public SharedFolder() { }

        public SharedFolder(string displayName, SharedFolder parent = null)
        {
            DisplayName = displayName;
            Files = new List<SharedFile>();
            SubFolders = new List<SharedFolder>();
            Parent = parent;
        }
    }
}
```

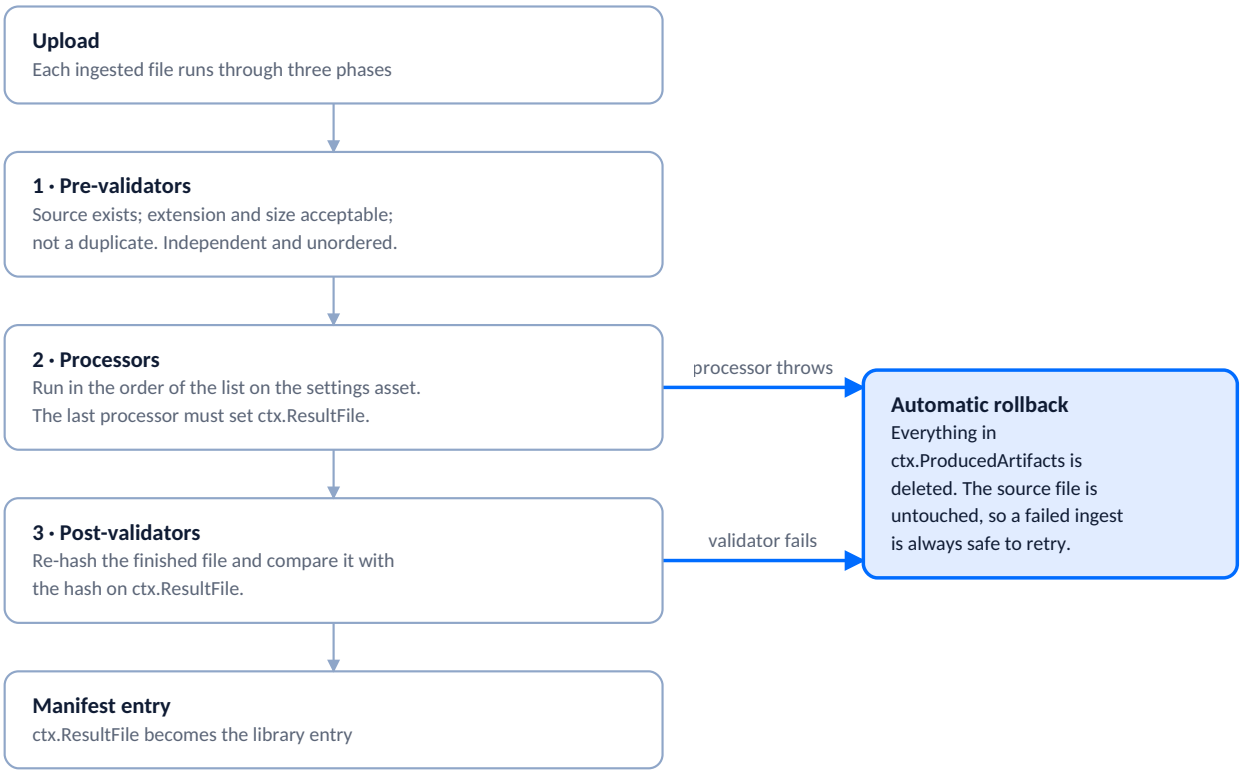
3.2 Build the ingest pipeline

Namespaces: `LibrarySystem.Core.Server.Validators`, `LibrarySystem.Core.Server.Processors`.

Every ingested file runs through three phases:

pre-validators → processors (in order) → post-validators

A failing processor or post-validator rolls back everything the run produced



- Validators within a phase are independent and unordered.
- Processor order matters and equals the order of the list on your settings asset. Reordering the pipeline is done in the Inspector, not in code.
- **Rollback is automatic:** if a processor throws or a post-validator fails, everything in `ctx.ProducedArtifacts` is deleted and the source file is untouched. A failed ingest is always safe to retry.

Validators

Subclass `IngestValidatorSO<TMedia, TCategory>` and give each one its own `[CreateAssetMenu]`:

```
[CreateAssetMenu(menuName = "Library System/Validators/File Type", fileName = "FileTypeValidator")]
public class FileTypeValidator : IngestValidatorSO<YourMediaFile, YourMediaFolder>
{
    public override UniTask<ValidationResult> ValidateAsync(IngestContext<YourMediaFile,
YourMediaFolder> ctx) { ... }
}
```

Phase	Typical checks
Pre-validators	Source exists; extension/size acceptable; not a duplicate. Compare against `ctx.Manifest.Flatten()`, which is a **live** reference, so duplicates within the same upload batch are caught.
Post-validators	Re-hash the finished file and compare with the hash on `ctx.ResultFile` to catch write/move corruption.

Processors

Subclass `IngestProcessorSO<TMedia, TCategory>` and implement `UniTask ExecuteAsync(IngestContext<TMedia, TCategory> ctx)`. **The last processor must set `ctx.ResultFile`**; that becomes the manifest entry.

Example: the bare structure of a custom processor. Like the validator above, it carries its own `[CreateAssetMenu]`:

```
[CreateAssetMenu(menuName = "Library System/Processors/My Processor", fileName = "MyProcessor")]
public class MyProcessor : IngestProcessorSO<YourMediaFile, YourMediaFolder>
{
    public override UniTask ExecuteAsync(IngestContext<YourMediaFile, YourMediaFolder> ctx) { ... }
}
```

Shipped base processors (both generic, so each needs a closed subclass carrying `[CreateAssetMenu]`):

Base	What it does	You write
`CopyToWorkingLocationProcessorBase`	Copies the source into a working temp file owned by the pipeline	Nothing but the closed subclass
`FinalizeAndMoveProcessorBase`	Hashes the final file (SHA-256), moves it to `{hash}{ext}` in the server media directory (throws "File already exists in library" if taken), calls `CreateEntry`	`CreateEntry(ctx, hash)`, and the entry's `Id` must be `hash`

```
[CreateAssetMenu(menuName = "Library System/Processors/Copy To Working Location", fileName = "CopyToWorkingLocationProcessor")]
public class CopyToWorkingLocationProcessor
    : CopyToWorkingLocationProcessorBase<YourMediaFile, YourMediaFolder> { }

[CreateAssetMenu(menuName = "Library System/Processors/Finalize And Move", fileName = "FinalizeAndMoveProcessor")]
public class FinalizeAndMoveProcessor : FinalizeAndMoveProcessorBase<YourMediaFile, YourMediaFolder>
{
    protected override YourMediaFile CreateEntry(IngestContext<YourMediaFile, YourMediaFolder> ctx, string hash)
        => new YourMediaFile { Id = hash, SizeBytes = new FileInfo(ctx.CurrentFilePath).Length, /* ... */ };
}
```

Optional overrides on `FinalizeAndMoveProcessorBase`:

- `ComputeHashAsync(ctx)` — defaults to hashing `ctx.CurrentFilePath` off the main thread. If no processor changes the bytes and a validator already hashed the source, return `ctx.SourceHash` to skip a second pass.
- `GetFinalExtension(ctx)` — defaults to the source extension. Override it if a processor converts the file (e.g. to `.mp4`).

Ids that aren't hashes (names, keys, slugs): write your own last processor instead (about fifteen lines):

1. Compute the id.

2. `File.Move(ctx.CurrentFilePath, {ServerMediaDirectory}/{id}{extension})`.
3. Keep rollback bookkeeping right: `ctx.ProducedArtifacts.Remove(oldPath)`, set `ctx.CurrentFilePath` to the new path, `ctx.ProducedArtifacts.Add(newPath)`.
4. Set `ctx.ResultFile`.

Renaming an entry later (`RequestRenameFile`) changes only `DisplayName`; the id and filename never change.

Pipeline helpers

Helper	Use
`FileHash.ComputeSha256Async(path)` (`LibrarySystem.Core.Shared`)	Off-thread SHA-256, e.g. in a validator that stores `ctx.SourceHash`
`ctx.GetOrCreateState<T>()`	One custom scratch-state object per ingest run, shared by all processors. Requesting a second, *different* `T` on the same run throws `InvalidOperationException`, so design one state class (e.g. `VideoIngestState { IsVideo, DurationSeconds }`).
`ctx.ReportProgress(p)`	Report **your own** processor's 0..1 progress. The pipeline converts it into overall progress.
`ctx.ProducedArtifacts`	Paths to delete on rollback
`ctx.ExtraArtifacts`	References to record on the entry (§3.7)

Don't create directories in processors. `ServerLibraryManager.Initialize()` already creates the serve root, the media directory and every registered extra-artifact directory.

Typical pipeline

1. **Copy** — closed `CopyToWorkingLocationProcessorBase`.
2. **Classify** — inspect the file and store its kind in scratch state.
3. **Extract before destroying** — e.g. generate a thumbnail from the original, so a later encode failure can't leave the file without one.
4. **Transform** — e.g. transcode; discard the pre-transform copy on success.
5. **Finalize** — closed `FinalizeAndMoveProcessorBase`, sets `ctx.ResultFile`.

3.3 Create the settings asset

`LibrarySystemSettings<TMedia, TCategory>` is generic and has no menu entry. Write a closed subclass, then create the asset from its menu:

```
[CreateAssetMenu(menuName = "Library System/Library System Settings", fileName =
"LibrarySystemSettings")]
public class YourLibrarySettings : LibrarySystemSettings<YourMediaFile, YourMediaFolder> { }
```

Field	Notes
`port`	Media server port (default **8080**). Passed to the backend's `Start(port)` and used by clients to build the URL.

Field	Notes
`serverHost`	**Required on the server.** The host asset from your backend (§4). Also decides where the serve root lives.
`clientLocation`	`StorageLocation` for client data. Default `PersistentDataPath`.
`preValidators` / `processors` / `postValidators`	Your validator/processor assets. Processor order matters.
`extraArtifactDirectories`	`NamedStoragePath` entries `{ Key, ClientDownloadable }` (§3.7).

Everything else (`ServeDirectory`, `ServerMediaDirectory`, `ClientMediaDirectory`, …) is derived. Server folders are created by `Initialize()` and client folders on first access. Assign **one** settings asset to both manager components (and to the NGO client bridge, if used).

Storage locations

`StorageLocation` is used for the client folder and for each backend's serve root:

Kind	Resolves to
`PersistentDataPath` (default)	`Application.persistentDataPath/{server|client}`
`Documents`	`Documents/{Folder Name}` — on Android, `/storage/emulated/0/Documents/{Folder Name}`
`Custom`	An absolute path you type. A relative path throws.

On Android, `Documents` and `Custom` need extra permissions — see §6.2.

3.4 Add the managers

`ServerLibraryManager` and `ClientLibraryManager` are plain `MonoBehaviour`s with no knowledge of networking. Create closed subclasses, attach those, and assign the settings asset:

```
public class YourServerLibraryManager : ServerLibraryManager<YourMediaFile, YourMediaFolder> { }
public class YourClientLibraryManager : ClientLibraryManager<YourMediaFile, YourMediaFolder> { }
```

Server manager

Item	Details
`Initialize()`	**Call explicitly** once this peer is confirmed as the server (e.g. Netcode's `IsServer`). It isn't called in `Start()` by default because the manager can't know its role. If there's no role to check (e.g. a LAN app where this scene is always the server), tick **Initialize On Start**. It creates folders, starts the backend, loads/creates the manifest, prunes missing files and fires `OnManifestLoaded`.
`Shutdown()`	Call explicitly; also auto-registered on `Application.quitting`. Stops the backend.
Write API	`RequestUpload`, `RequestNewFolder`, `RequestDeleteFile`, `RequestDeleteFolder`, `RequestRenameFile`, `RequestRenameFolder`, `RequestMoveFile`, `RequestMoveFolder`
Read API	`LibraryManifest`, `GetFileById(id)`, `GetFilePath(id|file)`, `GetExtraArtifactPath(id|file, key)` — return `null` for unknown ids/keys, don't check existence

Item	Details
Static events	`OnManifestLoaded`, `OnManifestPersisted` (passes new `LastUpdated`; local only), `OnUploadProgress`, `OnBatchUploadComplete`, `OnNotification`

Client manager

Item	Details
`Initialize(serverBaseUrl)`	Call once the server's URL (`http://{ip}:{port}`) is known. The NGO and LAN bridges do this for you.
`TriggerSync()`	Fetch manifest, diff, download/delete. Safe to call repeatedly; concurrent calls collapse into a no-op with a warning.
Read API	`LibraryManifest`, `GetFileById(id)`, `GetFilePath(id|file)` (adds `file://` on Android), `GetExtraArtifactPath(id|file, key)`
Static events	`OnSyncStarted`, `OnSyncCompleted`, `OnSyncFailed(string)`, `OnDownloadProgress`

`OnSyncFailed` fires when the server can't be reached (or the request is blocked), the device is out of storage, or downloads fail. A sync with nothing to do raises `OnSyncCompleted` **without** a preceding `OnSyncStarted`.

3.5 Connect server and client

Whatever you use to connect, the managers need only three things:

- **Server:** know when this peer is the server → call `Initialize()`.
- **Client:** know the server's base URL → call `Initialize(serverBaseUrl)`.
- **Client:** know when the library changed → call `TriggerSync()`.

The bridges in §5 do this for you.

3.6 HTTP and "Allow downloads over HTTP"

All backends serve plain **HTTP**, and clients download with `UnityWebRequest`, which Unity blocks in player builds by default.

> **Client builds must set Player Settings ▸ Other Settings ▸ Configuration ▸ "Allow downloads over HTTP" to "Always allowed"** (or "Allowed in development builds" for dev builds).

Without it, the request throws `InvalidOperationException` ("Insecure connection not allowed"), the error is logged, and the sync ends with `OnSyncFailed("Could not reach server.")`. The server doesn't need it, but a build that runs both roles does. If you put the host behind HTTPS (e.g. a reverse proxy), the setting isn't needed.

3.7 Extra artifacts (e.g. thumbnails)

Optional. For media with extra files beyond the primary file, implement `IHasExtraArtifacts`:

```
public interface IHasExtraArtifacts
{
    IReadOnlyList<ExtraArtifactRef> ExtraArtifacts { get; }
}

public struct ExtraArtifactRef
{
    public string DirectoryKey;  // resolved via LibrarySystemSettings.Get{Server,Client}Directory
    public string FileName;
}
```

The interface is read-only, so back it with a mutable list and implement it explicitly:

```
public class YourMediaFile : IMediaEntry<YourMediaFile, YourMediaFolder>, IHasExtraArtifacts
{
    // ... Id, SizeBytes, DisplayName, FileExtension, Parent ...
    public List<ExtraArtifactRef> ExtraArtifacts { get; set; } = new();
    IReadOnlyList<ExtraArtifactRef> IHasExtraArtifacts.ExtraArtifacts => ExtraArtifacts;
}
```

Workflow:

1. **Register the directory** on the settings asset: `{ Key = "thumbnails", ClientDownloadable = true }`. The key is both the URL segment and the local folder name. Always resolve paths with `settings.GetServerDirectory(key)` / `settings.GetClientDirectory(key)`; both return `null` for an unregistered (or, on the client, non-downloadable) key, meaning "nothing to do".
2. **Produce the file** in a processor, writing into `settings.GetServerDirectory("thumbnails")`, and record it twice: `ctx.ProducedArtifacts.Add(path)` for rollback, and `ctx.ExtraArtifacts.Add(new ExtraArtifactRef("thumbnails", fileName))` to record on the entry. Name it after something stable, such as the source hash.
3. **Copy the references onto the entry**, e.g. in `CreateEntry`: `entry.ExtraArtifacts = new List<ExtraArtifactRef>(ctx.ExtraArtifacts);` (copy, don't share). The entry is the only source of truth; core assumes no naming convention.
4. **Deletion and download are automatic.** Client downloads of artifacts are best-effort: a failed thumbnail doesn't fail the file's sync.
5. **Read** with `GetExtraArtifactPath(...)` on either manager, or `entry.FindExtraArtifact(key)` on an entry. None of the lookups check that the file exists, so call `File.Exists` before loading. `GetFileById` is a linear search; cache entries if you call it per row of a large list.

Keep `ctx.ExtraArtifacts` (files produced) separate from your scratch state (facts about the file). This lets processors from different packages contribute artifacts without sharing a state class.

Worked example: the **"Library System - Core Samples"** sample (Package Manager) shows an `ArtifactProcessor` producing a server-only JSON `metadata` sidecar for every file and a client-downloadable PNG `thumbnails` entry for image sources only, plus `SharedFile`, `FinalizeProcessor`, `LibrarySettings` and a `ServerUI` that shows the newest thumbnail. No external tools needed.

4. Media Server Backends

The core ships no server. The server needs exactly **one** backend, whose host asset goes into `serverHost` on the settings asset.

4.1 Comparison

	Nginx	EmbedIO
Package	`…backend.nginx`	`…backend.embedio`
Type	External `nginx.exe` process	In-process server (inside the Unity player)
Platforms	**Windows only** (Editor + Windows Standalone)	**Cross-platform** — tested on Windows (IL2CPP) and Android
Create asset	Assets ▸ Create ▸ Library System ▸ Media Server ▸ **Nginx Host**	Assets ▸ Create ▸ Library System ▸ Media Server ▸ **EmbedIO Host**
Serve root	`PersistentDataPath` , `Documents` or `Custom`	`PersistentDataPath` , `Documents` or `Custom`
Bundled files	`nginx.exe` , `nginx.conf.temp` , `mime.types` in `Nginx~/` (with nginx, OpenSSL, PCRE, zlib licenses)	`EmbedIO.dll` 3.5.2, `Swan.Lite.dll` 3.1.0 (MIT)
Runtime files	`{persistentDataPath}/nginx-runtime/` (conf, `logs/` , `temp/` , pid). Not configurable.	None outside the serve root
Build step	Post-build copies `Nginx~/` to `<Product>_Data/nginx`	None
Protocol	Plain HTTP	Plain HTTP
Watch out for	Non-Windows targets get a warning on the asset and no nginx files in the build	Don't also install EmbedIO/Swan via NuGetForUnity (duplicate assemblies)

4.2 Platform support

Server platform	Nginx	EmbedIO
Windows Editor	✔	✔
Windows Standalone	✔	✔ (tested IL2CPP)
Android	✘	✔ (tested)

Rule of thumb: if the server will ever run on Android, use EmbedIO. If the server is Windows-only, either works; nginx runs as a separate process, while EmbedIO keeps everything inside the player.

Clients don't need a backend; they only download over HTTP.

4.3 Setup (both backends)

1. Install the backend package.
2. Create the host asset from the menu in the table above.

3. Choose its serve root (`PersistentDataPath` , `Documents` or `Custom`).
4. Assign it to `serverHost` on your settings asset.

On Android, a `Documents` / `Custom` serve root needs all-files access (§6.2).

4.4 Nginx details

- Started as `nginx -p <prefix> -c <generated conf>` , prefix `{Application.persistentDataPath}/nginx-runtime` .
- In the Editor, the exe runs straight from the package folder. Nothing is written into `Nginx~/` .
- **Updating nginx:** replace the files in `Nginx~/` and bump the package version.

4.5 Writing your own backend

Subclass `MediaServerHostSO` (with `[CreateAssetMenu]`) and return an `IMediaServerHost` from `CreateHost()` . A host only:

- answers "where is the serve root" (`MediaServerHostSO.ServeRoot`), and
- runs `Start(port)` / `Stop()` .

The core owns everything under the serve root and creates the folders before `Start` .

5. Bridges

A bridge tells the client **where the server is** and **when to re-sync**. Bridges are optional: you can call the managers yourself.

5.1 Comparison

	NGO bridge	LAN bridge
Package	`…bridge.ngo`	`…bridge.lan`
For	Projects already using Netcode for GameObjects	Projects with no networking stack (or avoiding NGO) on a local network without internet
Dependencies	Core 1.0.1 + `com.unity.netcode.gameobjects` 2.9.2	Core 1.0.1 only
Discovery	Reads server address from `UnityTransport.ConnectionData.Address`	Server broadcasts a UDP beacon; client locks to the **first** server it hears
Re-sync signal	`ClientRpc` on every `OnManifestPersisted`	Manifest revision carried on the beacon (optional `LanLiveSync`)
Re-sync is	Always on	Optional
Closed subclasses	2	2 (+1 optional)
Extra components	Needs `NetworkManager` + `UnityTransport` + a `NetworkObject`	`LanBeacon` (server), `LanBrowser` (client), added automatically
Shared config	Your `LibrarySystemSettings` (port)	A `LanSettings` asset (app id, broadcast port, beacon options)
Network scope	Whatever your NGO session reaches	Same subnet only; broadcasts don't cross routers
Platform notes	None beyond NGO's own	Android multicast permission; Windows firewall prompt (§6)
Not supported	—	WebSockets/persistent connections, server lists, authentication

Both bridges: file transfer is always HTTP from the backend, never through the bridge, and clients still need "Allow downloads over HTTP".

5.2 NGO bridge

This is the only package that references `Unity.Netcode`. Projects that don't use NGO never install it.

Components

Component	Type	Behavior
`ClientLibraryNetworkBridge<TMedia, TCategory>`	`NetworkBehaviour`	On `OnNetworkSpawn`, if this peer is a client, reads the server address from `UnityTransport`, builds `http://<address>:<settings.Port>`, calls `ClientLibraryManager.Initialize`. Defines no RPCs.

Component	Type	Behavior
`ServerLibraryNetworkBridge<TMedia, TCategory>`	`NetworkBehaviour`	On the server, listens for `OnManifestPersisted` and sends a `ClientRpc`; clients respond with `TriggerSync()`. Send and receive live in one class, gated by `IsServer` / `IsClient`.

Setup

```
public class YourServerLibraryNetworkBridge : ServerLibraryNetworkBridge<YourMediaFile, YourMediaFolder> { }
public class YourClientLibraryNetworkBridge : ClientLibraryNetworkBridge<YourMediaFile, YourMediaFolder> { }
```

1. Attach both bridges to a `NetworkObject` that's present/spawned for **every** peer. The server bridge must exist on **both** server/host and clients, because the RPC runs on every peer.
2. Assign your settings asset to the client bridge's `settings` field (empty by default). It reads the port from it.
3. Make sure `NetworkManager` uses a `UnityTransport` component.
4. Make sure your client manager is active in the scene (its `Awake()` has run) before Netcode connects.
5. Still call `ServerLibraryManager.Initialize()` yourself once `IsServer` is confirmed.

No URL is ever entered by hand.

5.3 LAN bridge

Components

Component	Where	Role
`LanBeacon`	Server	Announces the server over UDP. Knows nothing about the library.
`LanBrowser`	Client	Listens for beacons, locks to the first server. Knows nothing about the library.
`LanServerBridge<TMedia, TCategory>`	Server	Starts the beacon once the server is serving; stops it when the server stops.
`LanClientBridge<TMedia, TCategory>`	Client	Passes the found server's URL to `ClientLibraryManager.Initialize`, which starts the first sync.
`LanLiveSync<TMedia, TCategory>`	Both, **optional**	Publishes the manifest revision on the server; re-syncs clients when it changes.

Setup

```
public class YourLanServerBridge : LanServerBridge<YourMediaFile, YourMediaFolder> { }
public class YourLanClientBridge : LanClientBridge<YourMediaFile, YourMediaFolder> { }
public class YourLanLiveSync     : LanLiveSync<YourMediaFile, YourMediaFolder> { }   // optional
```

Shared: create one **LanSettings** asset (Create ▸ Library System ▸ LAN Settings) and assign it to both `LanBeacon` and `LanBrowser`. App id and broadcast port must match on both sides. It also holds the server name, beacon interval, interface logging and the browser's initial `KeepListening`. Without an asset, both use the defaults: app id `library-system`, port **47778**. The app id only keeps two apps on one network apart; **it is not security.**

Server:

1. Add `YourLanServerBridge` to a GameObject (Unity adds and assigns `LanBeacon`).
2. Call `ServerLibraryManager.Initialize()`, or tick **Initialize On Start**. The beacon starts when the host is serving and advertises `LibrarySystemSettings.Port`.

Client:

1. Add `YourLanClientBridge` to a GameObject (Unity adds and assigns `LanBrowser`).
2. Keep your client manager enabled. With `discoverOnEnable` (default), the bridge searches on its own and calls `Initialize` when it finds the server.
3. For a "search again" button, call `YourLanClientBridge.Search()`.

Live sync (optional): add `YourLanLiveSync` and assign what's available:

- Server: assign `beacon`. The revision rides on every beacon, and a beacon goes out immediately when the manifest changes.
- Client: assign `browser`. The browser is forced to keep listening; a new revision triggers `TriggerSync()`. Failed syncs retry after `retryDelaySeconds`.

One component serves both roles, and either reference may be left empty.

5.4 Writing your own bridge

For WebSocket signaling, a lobby service, manual IP entry, etc., write a small component that:

1. Calls `ServerLibraryManager.Initialize()` when your stack says this peer is the host (and `Shutdown()` on teardown).
2. Calls `ClientLibraryManager.Initialize($"http://{serverIp}:{settings.Port}")` when the client knows the host address.
3. Listens to `ServerLibraryManager.OnManifestPersisted` on the server, sends a signal over your stack, and calls `TriggerSync()` on clients when it arrives.

6. Platform Notes

6.1 Support matrix

Package	Windows	Android
Core	✔	✔ (see storage notes)
Nginx backend	✔ Editor + Standalone	✘
EmbedIO backend	✔ (tested IL2CPP)	✔ (tested)
NGO bridge	Follows NGO	Follows NGO
LAN bridge	✔ (firewall prompt)	✔ (multicast permission)

6.2 Android

Storage: `Documents` and `Custom` locations need all-files access.

- The package never prompts by itself. Call `AndroidStoragePermission.IsGranted()` / `Request()` from your app when it suits you.
- Until access is granted, resolving such a location throws `InvalidOperationException`.
- Add `MANAGE_EXTERNAL_STORAGE` to your own `Assets/Plugins/Android/AndroidManifest.xml` (keeping Unity's generated `<application>` block). The package doesn't declare it, because package manifests merge into every app and Google Play restricts that permission.

LAN bridge: `LanBrowser` holds a Wi-Fi multicast lock, which needs `CHANGE_WIFI_MULTICAST_STATE`. Add it to your own `AndroidManifest.xml`. Test on real hardware. The beacon may fall back to broadcasting on `255.255.255.255` when the subnet mask isn't available (possible under Mono/Android).

Paths: client `GetFilePath` / `GetExtraArtifactPath` add a `file://` prefix on Android.

Server on Android: use EmbedIO.

6.3 Windows

- **LAN bridge:** the first run of a listener triggers a Windows Firewall prompt.
- **Nginx:** runtime files go to `{persistentDataPath}/nginx-runtime`; builds include `<Product>_Data/nginx`.

6.4 All platforms

- Client builds need **"Allow downloads over HTTP"** (§3.6).
- **Running server and client in one app:** disable one of the two manager objects per build. If both run and their storage resolves to the same folder, they corrupt each other's `library.json` / `media/`. No overlap check is done.

7. Troubleshooting

Symptom	Likely cause	Fix
Client fails immediately with `OnSyncFailed("Could not reach server.")`	"Allow downloads over HTTP" not enabled	Enable it in client Player Settings (§3.6)
`MissingMethodException` when creating a folder	`TCategory` lacks a public `(string name, TCategory parent)` constructor	Add the exact constructor
`InvalidOperationException` from `GetOrCreateState<T>()`	Two different state types requested in one ingest run	Use one shared state class
"File already exists in library" on ingest	Identical content already ingested (hash collision on name)	Expected for duplicates; add a duplicate pre-validator
`InvalidOperationException` resolving a storage location on Android	All-files access not granted	Call `AndroidStoragePermission.Request()`, add `MANAGE_EXTERNAL_STORAGE`
Can't add a component / create an asset from the menu	Trying to use an open generic type	Create a closed subclass (§1.4)
Every peer starts a server and writes a manifest	`Initialize()` called unconditionally	Call it only once the server role is confirmed
LAN client never finds the server	Different subnet, router between them, guest Wi-Fi isolation, mismatched `LanSettings`, missing Android permission, or blocked firewall	Same subnet, shared `LanSettings` asset, add permission, allow firewall
LAN client connected to the wrong server	Browser locks to the first beacon heard	Use a distinct app id per app; call `Search()` again
NGO clients don't re-sync	`ServerLibraryNetworkBridge` missing on clients	It must exist on every peer
NGO client never initializes	`settings` not assigned on the client bridge, no `UnityTransport`, or client manager inactive at connect time	Check all three
Duplicate assembly errors	EmbedIO/Swan also installed via NuGetForUnity	Remove the NuGet copies
Nginx backend does nothing in a non-Windows build	Nginx is Windows-only	Use EmbedIO
Thumbnail missing on client but file synced	Artifact downloads are best-effort, or key not `ClientDownloadable`	Check the key settings; `File.Exists` before loading

8. Reference

8.1 Packages and requirements

Package	ID	Depends on
Core	`com.genesiscreations.librarysystem.core`	Unity 6000.0
Nginx backend	`com.genesiscreations.librarysystem.backend.nginx`	Core
EmbedIO backend	`com.genesiscreations.librarysystem.backend.embedio`	Core
NGO bridge	`com.genesiscreations.librarysystem.bridge.ngo`	Core 1.0.1, `com.unity.netcode.gameobjects` 2.9.2
LAN bridge	`com.genesiscreations.librarysystem.bridge.lan`	Core 1.0.1

Assemblies: `GenesisCreations.LibrarySystem.Core` (Shared/Server/Client/Data), `GenesisCreations.LibrarySystem.Core.Editor` (editor tooling such as the `StorageLocation` drawer), `GenesisCreations.LibrarySystem.Bridge.NGO`, `GenesisCreations.LibrarySystem.Bridge.LAN`. Bridge assemblies are auto-referenced.

8.2 Closed subclasses checklist

Closed subclass of	Needed when	Needs `[CreateAssetMenu]`
`LibrarySystemSettings<,>`	Always	Yes
`ServerLibraryManager<,>`	Server	—
`ClientLibraryManager<,>`	Client	—
`IngestValidatorSO<,>` (each validator)	Per validator	Yes
`IngestProcessorSO<,>` (each processor)	Per processor	Yes
`CopyToWorkingLocationProcessorBase<,>`	Using the shipped copy step	Yes
`FinalizeAndMoveProcessorBase<,>`	Hash-named ids	Yes
`ServerLibraryNetworkBridge<,>`	NGO bridge	—
`ClientLibraryNetworkBridge<,>`	NGO bridge	—
`LanServerBridge<,>`	LAN bridge	—
`LanClientBridge<,>`	LAN bridge	—
`LanLiveSync<,>`	LAN live sync	—
`MediaServerHostSO`	Custom backend	Yes

8.3 Public extension surface

A project interacts only through the manager classes, the `*SO` bases (including `MediaServerHostSO` / `IMediaServerHost`), and the `Shared` / `Data` types. `IngestPipeline`, `ManifestTreeEditor`, `MediaFileStore`, `ClientMediaService`, `ManifestFetcher` and `ManifestDiffEngine` are `internal`.

8.4 Default values

Setting	Default
Media server port	8080
Client / serve root location	`PersistentDataPath`
LAN app id	`library-system`
LAN broadcast port	47778
LAN `discoverOnEnable`	On