

Managing Component Dependencies: Challenges and Solutions

By Abir HSAINI, Technical Services Engineer, 4D Inc.

Technical Note 25-06

Table of Contents

Table of Contents	2
Abstract.....	3
Introduction	3
Understanding Dependency Management in 4D.....	3
Project Dependency Handling with JSON Files in 4D	3
1. Dependencies.json – Declaring Required Components	3
2. Environment4d.json – Customizing Component Paths	4
Monitoring Project Dependencies via the Project Dependencies feature in 4D	5
1. Adding a dependency	6
2. Updating GitHub dependencies	7
Differences Between Interpreted and Compiled Modes	8
Tags and versions.....	8
Common Issues and Solutions	10
Component Update Not Detected or Not Applied After Restart	10
The download archive does not contain a valid dependency	10
Unable To find the latest release for	10
Component Version Changes Without Explicit Permission	11
Demo Application	11
Publish the component in private repository.....	11
Create a release	12
Add the GitHub access token	13
Add dependency to the project	13
Conclusion	14

Abstract

Managing 4D components through project dependencies can pose several challenges during development, particularly regarding version control and update consistency. These issues can lead to unexpected behavior, and integration difficulties across projects.

This technical note outlines the most common problems developers face when working with 4D components and offers recommended best practices to mitigate them. The objective is to provide a clearer understanding of effective dependency management strategies, ensuring more stable and maintainable 4D applications.

Introduction

A **4D component** is a reusable package of 4D code designed to add specific functionality to a project. Developers can create their own components to structure code modularly or incorporate components developed by the 4D community, often shared on platforms like GitHub.

4D also offers several built-in components—such as **4D SVG**, **4D NetKit**, **4D View Pro**, and **4D Write Pro Interface**—which provide advanced features like graphics rendering, email integration, spreadsheet processing, and rich text document editing. These components help developers quickly extend their applications with powerful capabilities.

However, as projects increase in size and complexity, managing dependencies between components becomes more challenging. Common issues include strict version requirements, behavioral differences between interpreted and compiled modes, and update processes that can occasionally fail.

This technical note addresses these challenges and presents practical recommendations for achieving stable and reliable component dependency management in 4D applications.

Understanding Dependency Management in 4D

Project Dependency Handling with JSON Files in 4D

A 4D project can use external components. It is important to manage dependencies in a clear and flexible way. 4D creates two JSON files: **dependencies.json** and **environment4d.json**

The files work together to define what components the project needs, and where 4D should find them whether they're stored locally or hosted online, like on GitHub.

1. Dependencies.json – Declaring Required Components

The **dependencies.json** file is the central place for declaring all external components required by the project. It ensures 4D loads the correct components each time the project opens, whether they are stored locally or hosted remotely.

The file **must be placed** in the following path within the project package:

```
/TheProject/Project/Sources/dependencies.json
```

It can declare:

- **Local components**, stored on the machine in a standard components folder.
- **GitHub-based components**, fetched from public or private GitHub repositories.

Example:

```
{
  "dependencies": {
    "theGitHubComponent1": {
      "github" : "JohnSmith/theGitHubComponent1"
    }, // GitHub-based component
    "theLocalComponent2" : {} // Local component
  }
}
```

In the example, the project requires two components: one stored locally (theLocalComponent) and another fetched from GitHub (theGitHubComponent1). The clear declaration helps reproduce the setup across different environments.

The path for “theLocalComponent2” can be defined either here (Dependencies.json) or in the environment4d.json file which we’ll cover shortly as part of the next sub-section.

2. Environment4d.json – Customizing Component Paths

The dependencies.json file defines the required components, and the environment4d.json file specifies the location of the components. It allows developers to override the default paths of components, especially useful in team environments or multi-project setups.

The file is optional, but extremely useful when:

- Using the same components across multiple projects.
- Working with a local version of a GitHub component during development or testing.
- Keeping local paths out of version control to maintain clean and portable repository

The **Environment4d.json** file can be placed in:

- The root folder of the project
- Any parent folder above the project (4D searches upward until the file is found)

To use local components stored outside the project folder, it is possible to define the paths in the environment4d.json file. It helps 4D know where to find each component, even if it is placed in another folder on the computer.

Both relative and absolute paths are supported. Using relative paths is usually better, because it makes the project easier to move, share, or store in a source control system like Git. It also avoids problems linked to fixed file locations on different machines.

Example:

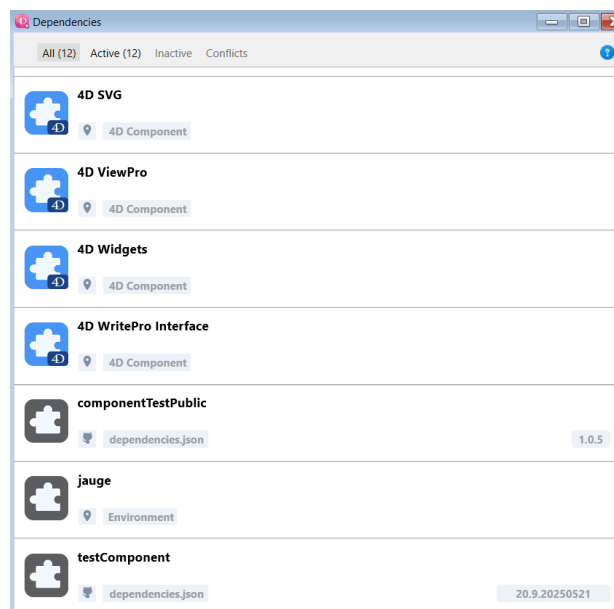
```
{
  "dependencies": {
    "theLocalComponent1": "theComponent1",
    "theLocalComponent2": "../theComponent2",
    "theLocalComponent3": "file:///Users/jean/theComponent3"
    "theGitHubComponent1": {
      "github": "JohnSmith/theGitHubComponent1"
    }
  }
}
```

4D looks for components in three main locations. First is the Components folder inside the project directory. Second are the paths listed in the dependencies.json file. Third is any custom path set in the environment4d.json file. When loading components, 4D checks the Components folder first, since it has the highest priority. Next, it uses the custom paths from environment4d.json if set, which override the dependencies file paths. If the component is not found in these places, 4D loads one of its built-in components like 4D NetKit or 4D SVG. The order ensures the correct component loads based on the environment

Monitoring Project Dependencies via the Project Dependencies feature in 4D

Another simple way to manage dependencies beyond using dependencies.json and environment4d.json files is through the Dependencies panel, accessible via the Design → Project Dependencies menu.

The graphical interface allows developers to add, remove, update, and monitor project dependencies directly within 4D, without needing to manually edit JSON files. It offers a more visual and intuitive way to handle component management, especially useful during development and debugging.

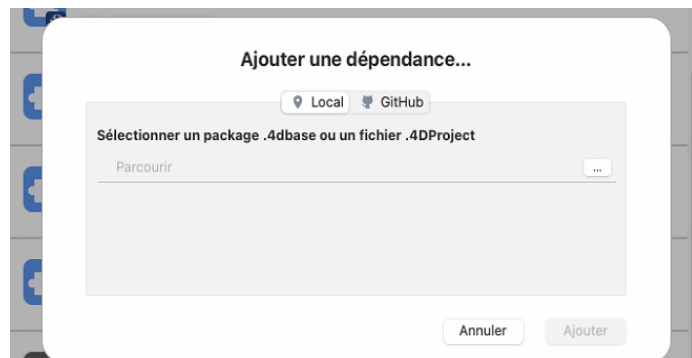


The Dependencies panel in 4D displays all the dependencies used in a project, regardless of the source. Each dependency includes an origin tag beneath its name, indicating where the component was loaded from.

Origin tag	Description
4D Component	A built-in 4D component located in the Components folder of the 4D application.
dependencies.json	A component declared in the dependencies.json file of the project.
Environment	A component defined in the environment4d.json file.
Project Component	A component physically located inside the Components folder of the project.

1. Adding a dependency

Adding a dependency starts by clicking the + button in the panel's footer. It is then possible to select the source, either from the local system or GitHub.



For a local component, if the selected item is correct and located next to the project folder, it will be declared only in the dependencies.json file. However, if it is not located next to the project folder, its file path will also be specified in the environment4d.json file.

Edit the dependency...

Local
GitHub

Address of the GitHub repository

technote-25-06/Jauge

Dependency Rule Latest

Cancel
Apply

For a GitHub component, you must **add the GitHub repository** address and define a **dependency rule** to control how the version is resolved. This is done via the component management dialog:

To declare a GitHub component, the GitHub repository address must be provided, along with a dependency rule to control version resolution. This is configured through the component management dialog:

Dependency Rule

Follow 4D version

Latest
 Up to Next Major Version
 Up to Next Minor Version
 Exact Version (Tag)
 Follow 4D version

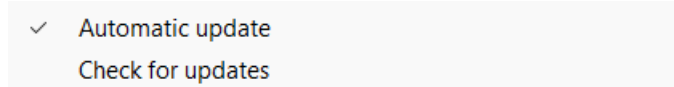
- **Repository:** Specify the GitHub path in the format owner/repository
- **Dependency Rule:**
 - Latest: Downloads the most recent release tagged as latest.
 - Up to Next Major Version: Updates the component within the same major version range.
 - Up to Next Minor Version: Restricts updates to minor version changes only.
 - Exact Version (Tag): Downloads a specific version by tag.
 - Follow 4D Version: Downloads the release that matches the 4D version in use, assuming the release follows 4D's naming conventions.

This configuration provides flexibility in managing component compatibility and versioning.

2. Updating GitHub dependencies

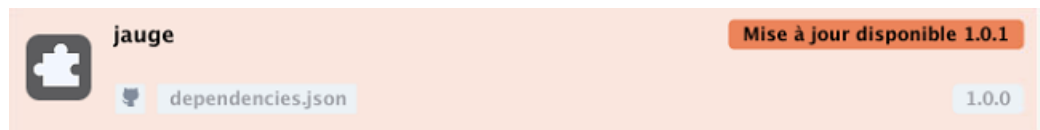
4D's Dependency Manager provides integrated support for **checking and updating GitHub components**, either **automatically** or **manually**.

- **Automatic checks** for new versions run in the background. If a GitHub access token is provided, checks are more frequent.



- **Manual checks** for updates:
 - For a **single dependency**: right-click and select *Check for updates* or *Update on next startup*.
 - For **all dependencies**: use the options menu at the bottom of the panel.

When an update is detected (based on the versioning rules), a status appears. The developer chooses to apply the update or ignore it.



Once an update is initiated, a dialog suggests restarting the project to load the new version. If postponed, the update will occur at the next startup.

Differences Between Interpreted and Compiled Modes

In 4D, a **component** can be either **interpreted** or **compiled**, and this distinction determines how it can be used within a host project.

An **interpreted component** contains uncompiled source code. It can only be used in a **4D project running in interpreted mode**. Its structure is similar to a standard 4D project folder (named, for example, theComponent.4dbase), which includes all project files such as methods, forms, and resources. To be recognized as a component, the folder must be placed in the Components folder of the host project.

On the other hand, a **compiled component** contains precompiled code in a .4DZ file. The type of component can be used in both **interpreted** and **compiled projects**, making it more flexible and suitable for deployment scenarios. If the main project runs in compiled mode, the developer must use compiled components only.

A simpler structure can be used, where the .4DZ file is placed directly at the root of the component folder, possibly along with a Resources folder and an Info.plist. However, the simpler structure is less suitable for macOS notarization requirements.

Tags and versions

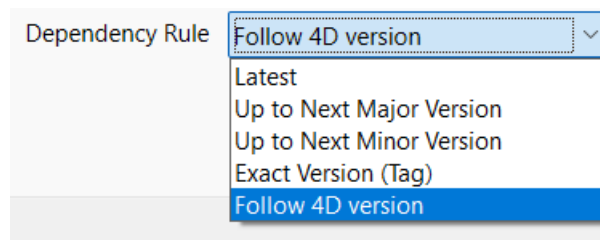
When working with GitHub components in 4D, each release is associated with a unique tag and a semantic version number. The identifiers can be used to specify which version of a component a project should use. In the dependencies.json or environment4d.json files, the

developer can define either a specific tag (e.g., "tag": "beta2") or a semantic version (e.g., "version": "2.1.3"). The versioning follows the Semantic Versioning format: major.minor.patch. He can also use version ranges and wildcards to define flexible dependencies.

For example, "1.*" targets all versions within major version 1, while ">=1.2.3" includes all versions starting from 1.2.3. Prefixes like ^ and ~ are also supported, allowing the developer to specify compatible version ranges. If neither a tag nor version is provided, the system will automatically fetch the latest available release from GitHub.

```
{
  "dependencies": {
    "theFirstGitHubComponent": {
      "github": "JohnSmith/theFirstGitHubComponent",
      "version": ">=1.2.3"
    }
  }
}
```

Selecting the "Follow 4D Version" dependency rule requires component tags on GitHub to use specific naming conventions. it ensures 4D correctly matches them with the corresponding project version.



For Long Term Support (LTS) versions, the tag must follow the pattern **x.y.p**, where **x.y** matches the main 4D version, and **p** represents optional patches. For example, specifying **20.4** directs the Dependency Manager to retrieve the latest component version matching **20.*** or a version just below 20.



For R-Releases, the tag format follows **xRy.p**, for example **20R9.1**. The system looks for the latest component version below **20R10**. If no matching release is found, 4D displays a notification. These conventions ensure dependency updates stay aligned with the project's 4D versioning strategy.

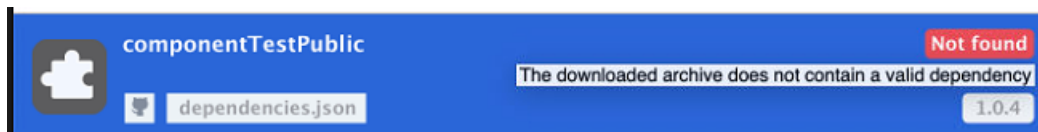
Common Issues and Solutions

While managing dependencies through the Dependency Manager, several issues may arise. The section aims to highlight common problems encountered in real-world projects, along with practical solutions and best practices to prevent or resolve them.

Component Update Not Detected or Not Applied After Restart

Sometimes, an expected update for a GitHub component does not show an "Update available" message or is not applied after restarting the project. First, if the Automatic update option is disabled. In this case, updates are not applied automatically, even if a new version exists. The developer must right-click the component, select "Update on next startup," then restart the project to apply the update. Second, if no update is detected at all, it's possible the version on GitHub is **not tagged as the latest release**. 4D relies on GitHub's release tags to detect updates, so if the version to use is not correctly published or tagged, the Dependency Manager will not recognize it as a valid update.


The download archive does not contain a valid dependency



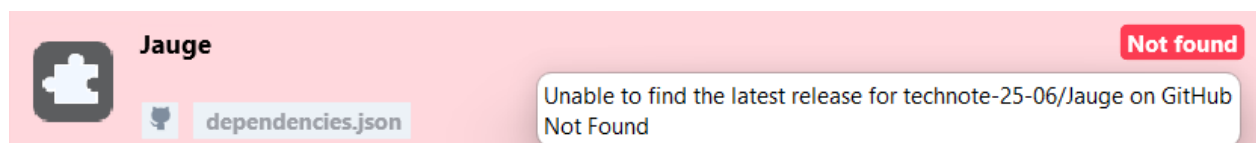
The error typically appears in the Dependency Manager when attempting to load a GitHub component, and 4D cannot recognize the structure of the downloaded archive as a valid component.

The most common reason is that the ZIP archive associated with the GitHub release is not properly structured. 4D expects the root of the archive to directly contain a valid 4D component. If the archive contains an extra top-level folder, or if the component is nested too deep, 4D cannot locate the required structure and throws the error.

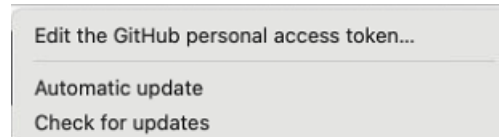
Below the valid structure of the component:

 testComponent.4dbase	89 ko	Packag...ées 4D	aujourd'hui à 18:08
 testComponent.zip	20 ko	Archive ZIP	aujourd'hui à 18:08

Unable To find the latest release for ...



After verifying the latest release exists, if the error persists, it may be caused by an incorrect access token. In that case, the developer should modify the token by clicking on "Edit the GitHub **personal access token**" from the Dependency Manager.



Component Version Changes Without Explicit Permission

When the developer runs the application in **developer mode**, 4D automatically executes a background script approximately **three minutes after the application starts** to check for component updates — even if no manual check was requested.

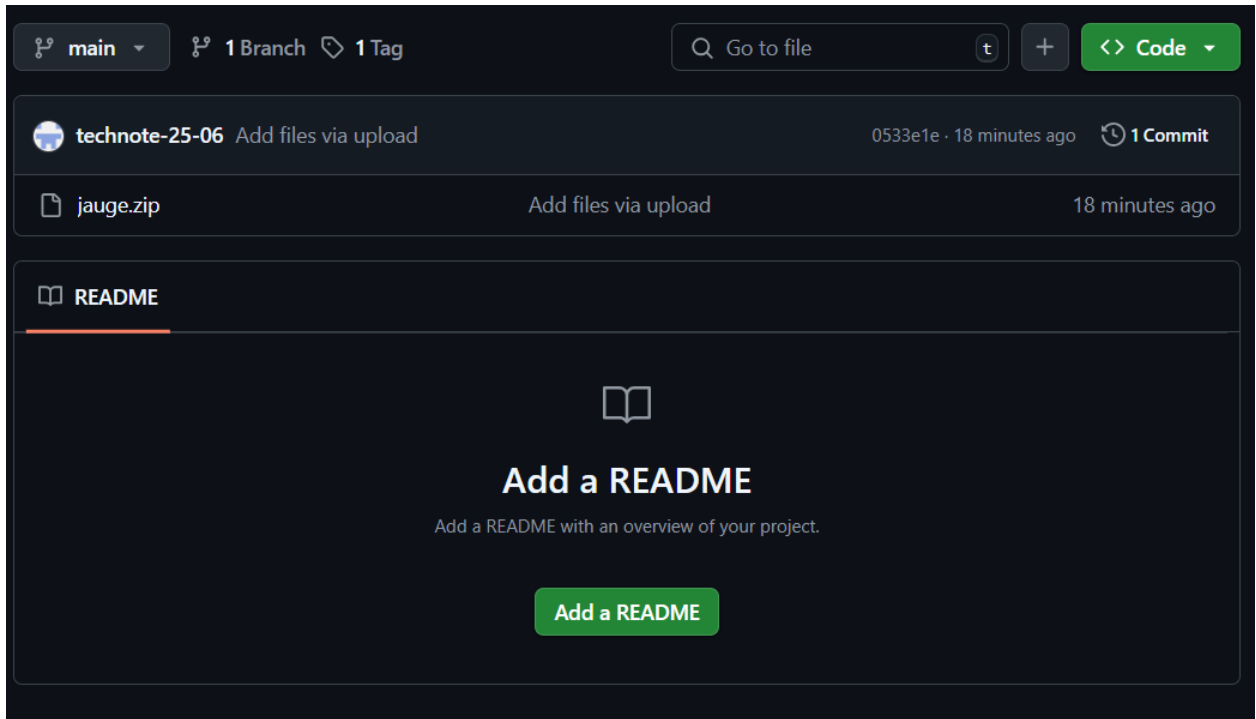
If a new version is found, 4D will notify that an update is available. However, if the developer does not pay attention to this notification and restarts the database (with **automatic updates enabled**), the new version will be automatically installed without further confirmation.

Demo Application

In the demo, a private component named "jauge" will be published. The component draws a gauge showing a percentage value. The gauge turns green when the percentage is below 70%, and red when it is 70% or higher.

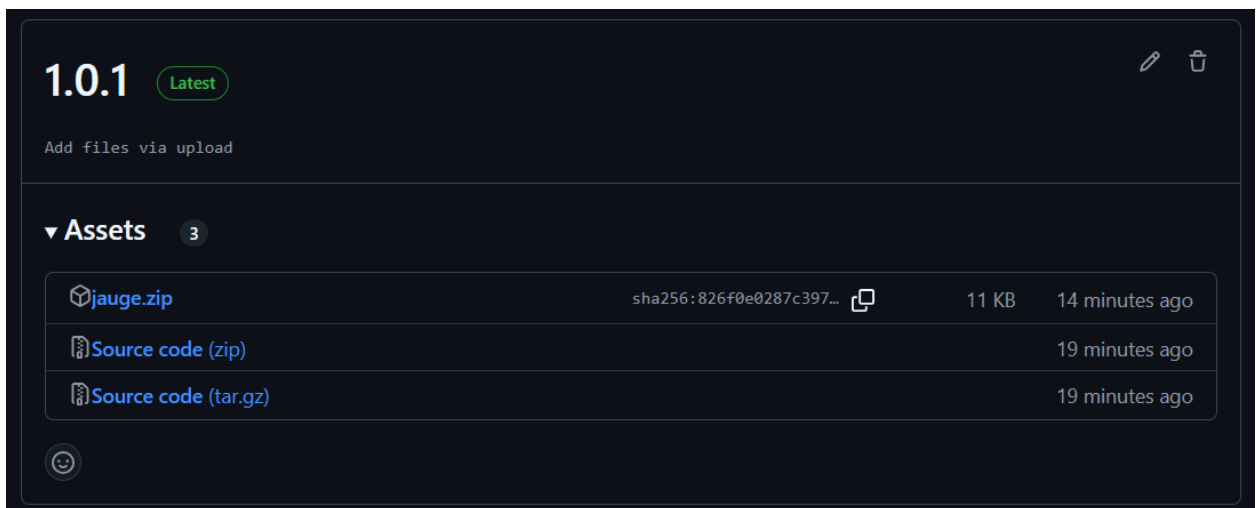
Publish the component in private repository

The component has been added to GitHub.



Create a release

A release is created using the predefined tag and version, such as 1.0.3. It makes the component available for use in the 4D Dependency Manager.

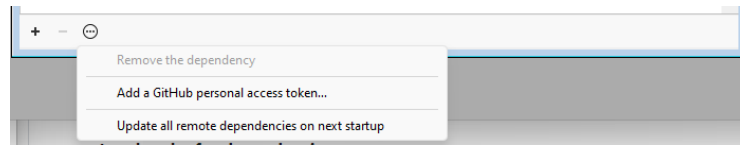


To respect the naming convention for 4D version tags, the release can be named **20.9.9385**.

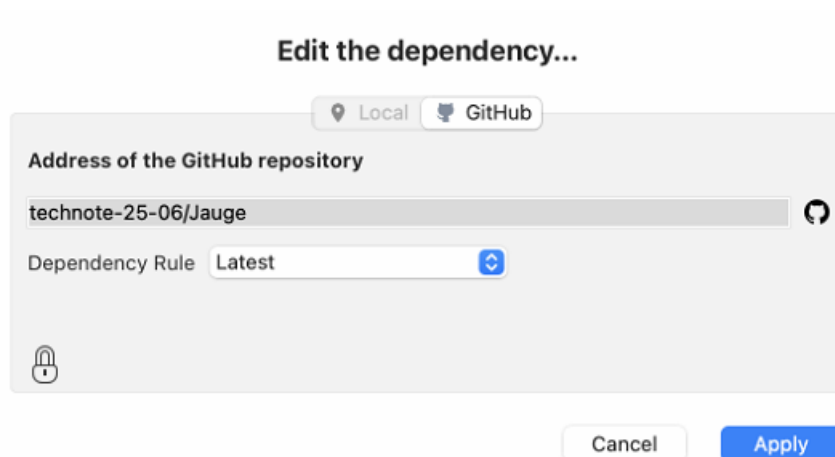


Add the GitHub access token

To use a private GitHub repo in 4D Dependency Manager, the developer should add the GitHub personal access token by Selecting **“Add a GitHub personal access token...”** from the Dependency Manager. For the test , token: ghp_XyYBLzvoQ4x2LIKNzAwTPa3S4Xri5w3PluPu

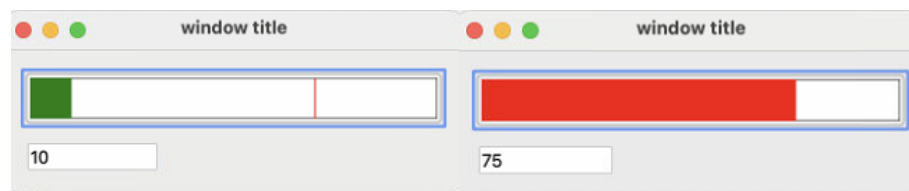


Add dependency to the project



The developer adds the GitHub dependency by entering the repository path: technote-25-06/Jauge/ and selecting the dependency rule. Then, restarting the database makes the component available.

Now, the method launchJauge is run. Here is the result:



This demo shows how to publish a private component on GitHub, add it as a dependency in 4D, and use it seamlessly in a project. By following these steps, developers can manage reusable code efficiently and ensure consistent component updates across different environments.

Conclusion

Managing dependencies in a 4D project especially when using external components from GitHub requires both structure and attention to detail. Throughout this note, we've explored how to declare dependencies using `dependencies.json` and `environment4d.json`, how to monitor and troubleshoot them using the Dependency Manager, and how to handle updates and versioning correctly.