Charon Documentation

version 2026.1.1

Denis Zykov

January 21, 2026

Contents

Overview

Charon is a powerful data-driven game development tool designed to streamline the creation and management of static game data within your game. It allows both developers and game designers to efficiently model and edit game entities such as characters, items, missions, quests, and more, directly within the Unity/Unreal Engine/Browser environment. Charon simplifies the process of data manipulation, offering a user-friendly interface and automatic source code generation, which significantly reduces development time and minimizes manual coding errors. Charon also offers support for working with text in multiple languages, with easy loading and unloading of translated text.

With Charon, game developers can focus on creating engaging gameplay experiences without worrying about the technical details of managing game data. It is available in three deployment variants, including a standalone application, web application, Unity plugin and Unreal Engine plugin.

TLDR Charon is an in-game database for your game, replacing spreadsheets or config files.

Why Choose Charon?

Charon replaces traditional spreadsheets or config files with an in-game database, offering a structured and efficient way to manage game data. It allows developers to focus on creating engaging gameplay experiences without worrying about the technical details of data management.

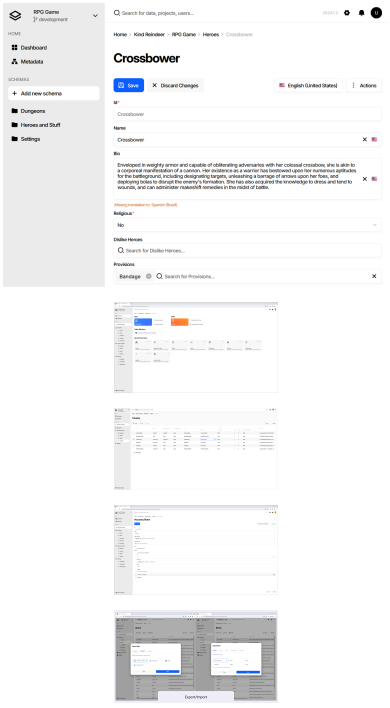
Is It Free?

The offline version, CLI and plugins are completely free and have no restrictions. They are distributed under a free license and allow you to distribute tools along with the game for modding games.

The online version, which allows working in large teams on shared game data, requires a subscription.

What is Charon

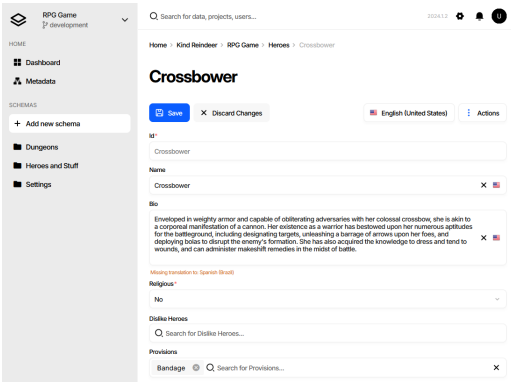
It is a .NET 8 console application that can be used as a command-line tool for performing CRUD operations with your game data, or as an HTTP Server to provide a UI for modeling and editing your game data. There are plugins for Unity and Unreal Engine that provide a more integrated experience while using Charon. As with any modern .NET application, it can be launched as is on Windows, macOS and Linux and via *dotnet* tool.



Further reading

Unreal Engine Plugin Overview

For Unreal Engine developers, managing complex game data shouldn't mean fighting with restricted DataTables or bloated Blueprints. The Charon Unreal Plugin provides a robust, professional-grade database workflow designed to handle the scale and performance requirements of modern UE projects.



---

1. What is it?

The **Charon Unreal Plugin** is a high-performance data management ecosystem integrated directly into the Unreal Editor. It replaces traditional, flat DataTables with a **relational game database** that supports complex nesting, cross-references, and multi-language text—all accessible via a native-feeling UI and backed by an automated C++ code generator.

2. Which problem does it solve?

While Unreal Engine is powerful, its default data tools often fall short in complex productions. Charon solves these specific pain points:

- **DataTable Limitations:** Unlike standard DataTables, Charon handles deep hierarchies and complex relationships (like branching quest lines or intricate skill trees) without becoming a visual mess.
- **Boilerplate Fatigue:** It automatically generates the **C++ and Blueprint** code required to access your data, eliminating the need to manually write struct definitions or parsing logic.
- **Data Fragility:** With built-in validation checks, it catches errors (like broken links or out-of-range values) before they cause a crash in your build.
- **Content "Locked" in the Engine:** It allows for easy import/export for spreadsheets and localization, making it easier to collaborate with external writers or translation agencies.

3. For whom?

- **Technical Directors & Programmers:** Who need type-safe, performant C++ access to game data and a reliable pipeline for CI/CD and modding support.
- **Game & Narrative Designers:** Who require a structured environment to build massive game worlds, item economies, and dialogue systems without needing to know C++.
- **Live-Ops Teams:** Who need the ability to dynamically load data updates or perform A/B testing without submitting a new build to storefronts.

Key Features

Feature	Benefit to Your Workflow
C++/Blueprint Generation	Access your data instantly with full IDE auto-complete and Blueprint nodes.
Relational Modeling	Interconnect tables (Items, Quests, NPCs) with ease within a single UI.
Hot Updates	Support for dynamic loading of game data for live-tuning and hotfixes.
Modding Support	Give your community the same professional tools to create game mods.
Localization	Seamlessly export and import translation keys for global releases.

Getting Started

To begin using this plugin, the initial step involves installing the plugin from the Unreal Engine Marketplace. Once installed, you'll need to enable the plugin for your project through the project settings. Following this, a rebuild of your project's C++ code is necessary. The final step in the setup process is the creation of your first game data file.

Prerequisites

The Unreal Engine plugin is written in C++ but relies on `dotnet charon`, a .NET Core application which runs on .NET 8.

Windows

1. Download and install SDK NET 8+.
2. Make sure you have write access to `%PROGRAMDATA%/Charon`.

MacOS

1. Download and install SDK NET 8+.
2. Make sure you have write access to `/Users/<username>/.config/Charon`.
3. Make sure `dotnet` is available from `$PATH`.

Linux

1. Download and install NET SDK 8+.
2. Make sure you have write access to `/home/<username>/.config/Charon`.
3. Make sure `dotnet` is available from `$PATH`.

Checking Available .NET SDK Versions

```
# check for dotnet already installed
dotnet --list-sdks

# output for dotnet --list-sdks
7.0.120 [C:\Program Files\dotnet\sdk]
8.0.206 [C:\Program Files\dotnet\sdk]
8.0.414 [C:\Program Files\dotnet\sdk]  # <- this one is fine
9.0.300 [C:\Program Files\dotnet\sdk]  # <- this one too
```

Installation from Marketplace

1. Add to cart Charon plugin [Epic Launcher] / [Fab.com Asset Page] in the Unreal Engine Marketplace.
2. Follow the instruction on installing plugin into your project:
    a. Click **Install to Engine** and select the engine version.
    b. Open your project and go to **Edit** → **Plugins…** window.
    c. Type **Charon** in the **Search** bar.

d. Check the checkbox near the plugin's name to enable it.

3. Rebuild project C++ code.

Building from Source Code

1. Clone or download the plugin source code from the GitHub repository.
2. Create a `<project-dir>/Plugins/Charon` directory.
3. Copy the plugin files into this directory. Ensure **Charon.uplugin** is located at the path `<project-dir>/Plugins/Charon/Charon.uplugin` after copying.
4. Remove the `"EngineVersion"` attribute if your engine doesn't match the plugin's engine version.
5. Rebuild the project's C++ code.
6. Enable the plugin in **Edit** → **Plugins…** if needed.

Core Concepts

Data-Driven Design Principles

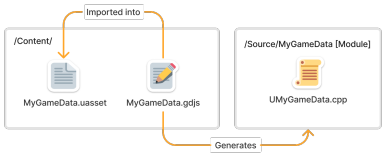
Data-driven design emphasizes the control of gameplay through data, rather than source code/blueprints, with game mechanics and processes determined by structured data files. For instance, rather than embedding damage calculations directly in the game's source code, these are defined by data specifying weapon effects and the rules for their application. Or for example, mission progression is not hardcoded; it's outlined in editable text files, making these aspects of game design highly flexible. This approach not only facilitates quick adjustments during development but also simplifies adding modding support post-release.

- Data Driven Gameplay Elements (UE Documentation)
- Modify Everything! Data-Driven Dynamic Gameplay Effects on 'For Honor' (Video)
- Data-driven Design in Unreal (Article)

Understanding the Plugin's Architecture

Plugin Assets

Working with data in this plugin is akin to how the built-in *DataTable* functions. There is a data source file, a module containing the code required to load the data, and an asset that will be utilized in the game. Whenever you edit a data source file, you need to re-import this data into the asset. Should the data structure in the source file change, then the C++ code must be regenerated.



For scenarios requiring dynamic loading of game data, this can be accomplished through the `TryLoad` method on the game data class, which accepts the source JSON file.

Plugin Modules

The Charon plugin is structured into two modules:

- `CharonEditor` module acts as an Unreal Engine Editor extension. Extension points for the module are declared in the `ICharonEditorModule` class, and automation of game data processing is facilitated through the `FCharonCli` class.
- `Charon` module, houses the core logic and shared code crucial for handling game data files.

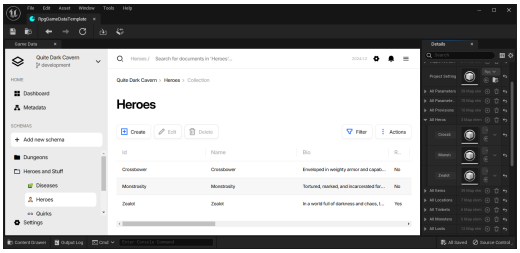
Working with the Plugin

Creating Game Data

To create a new game data file within the Unreal Engine Editor, open the **Content Drawer**, right-click in the desired folder, and select in the **Create Advanced Assets** section **Miscellaneous** → **Game Data** menu option. Name your game data file and proceed according to the instructions in the dialog window that appears.

Detailed guide on how to create game data.

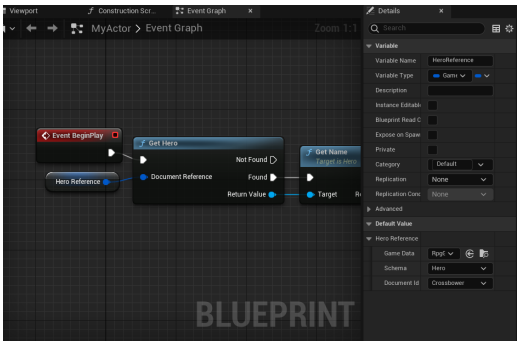
Editing Game Data



To edit a game data file in the Unreal Engine Editor, navigate to the **Content Drawer**, find the corresponding .uasset file, and double-click it. This action opens a new window featuring a user interface for editing the game data. Remember to reimport and, if necessary, regenerate the source code after completing your edits.

Refencing Game Data in Blueprints

Similar to the DataTable's `FDataTableRowHandle`, the Charon plugin introduces a specific type for referencing documents within Blueprints, named `FGameDataDocumentReference`. This type is housed within the Charon module. Here is example of **Game Data Document Reference** used to resolve *Hero* document:



Advanced Features

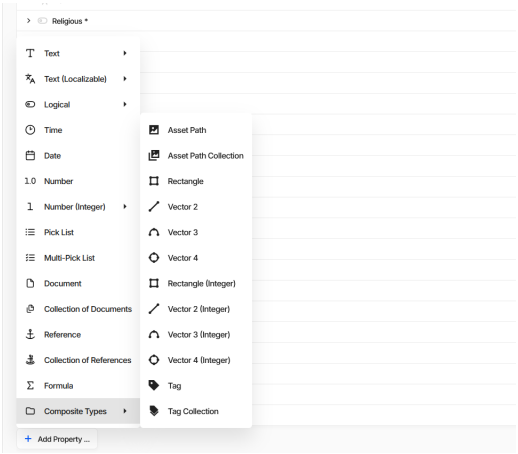
Localization and Multi-Language Support

Charon facilitates multi-language text support through the `Localizable Text` data type. When creating a *Schema*, properties can be defined with various data types, including `Localizable Text`. Initially, all localizable text defaults to `EN-us` (US English). Additional languages can be added via **Project Settings** → **Internationalization** → **Translation Languages** in the Charon UI.

Exporting/importing localizable data.

Referencing Unreal Engine Assets

To reference assets within the game, you can use a special Asset composite type. Create a property via `Composite Types` -> `Asset Path` menu options.



Feedback

We welcome and encourage feedback, particularly bug reports and suggestions, to help improve our tool. If you have any questions or would like to share your thoughts, please join our Discord community or reach out to us via email at support@gamedevware.com.

See also

- Basic Navigation and User Interface Overview
- Creating Document Type (Schema)
- Filling Documents
- Frequently Asked Questions (FAQ)
- Glossary

How to Create Game Data File

To create a new game data file within the Unreal Engine Editor, open the **Content Drawer**, right-click in the desired folder, and select in the **Create Advanced Assets** section **Miscellaneous** → **Game Data** menu option. Name your game data file and proceed according to the instructions in the dialog window that appears.

Step By Step

1. **Open Content Drawer:** Open the *Content Drawer* window in the Unreal Engine Editor.
2. **Select Folder:** Right-click in the desired folder where you want to create the game data file.
3. **Create Game Data:** Navigate to **Create Advanced Assets** → **Miscellaneous** → **Game Data** from the context menu.
4. **Name the File:** In the *Content Drawer* window that appears, enter a name for your game data file.
5. **Check for Errors:** Ensure there are no error messages in the dialog window that opens, then press *Next*.
6. **Wait for Module Generation:** Allow time for the new module to be generated, watching the wizard in the dialog proceed to the next step automatically.
7. **Review Summary:** Check the summary and verify there are no suspicious errors in the *Output Log* window.
8. **Recompile C++ Code:** Use your IDE of choice to recompile the C++ code. Restart Unreal Engine Editor if needed.
9. **Import Game Data:** Reopen the *Content Drawer* window and click the **Import** button.
10. **Select .gdjs File:** Locate and select the *.gdjs* game data file you named in step 4, then click *Ok*.

11. **Choose Game Data Class:** Select the *Game Data* class, which should match the game data file name. If it's not listed, return to step 7.
12. **Save .uasset File:** Save the newly created *.uasset* file after completing the import process.

Troubleshooting

Game data creation or code generation/compilation may encounter issues under certain circumstances:

Insufficient File System Rights or File Creation Errors - **Problem**: Lack of sufficient rights to the OS file system, or errors during file creation (e.g., file name too long, antivirus block). - **Solution**: Check the *Output Log* window for errors or the most recent log file in `<project-dir>\Intermediate\Charon\logs` and attempt to resolve them.

Class Name Collision Within Project - **Solution 1** (Game Data Class Name Collision): Delete the newly created **.gdjs** game data file and the generated module. Then, start over with a new name and clean your `.Target.cs` files from the generated module name. - **Solution 2** (Schema Class Name Collision): Open the game data in another editor (Online, Standalone), rename the schema, and try again.

No Game Data Class in Import Window - **Problem**: The generated game data module is not being compiled. - **Solution**: Ensure it's added to your `<project-name>.Target.cs` and `<project-name>Editor.Target.cs` files as an extra module. If missing, include following expression in both target files:

```
ExtraModuleNames.Add("<module-name>");
```

Additionally, verify that your **.uproject** file includes the generated module definition. If it's absent, add the following module definition to the **Modules** list:

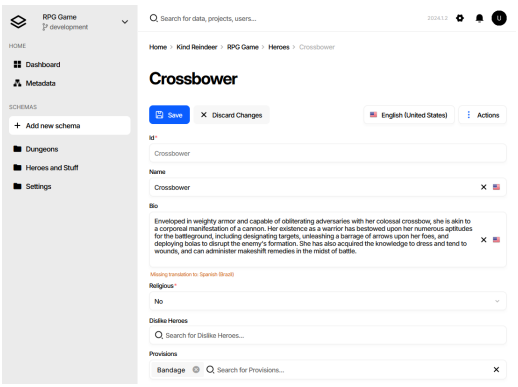
```
{
    "Name": "<module-name>",
    "Type": "Runtime",
    "LoadingPhase": "Default"
}
```

See also

- Basic Navigation and User Interface Overview
- Creating Document Type (Schema)
- Filling Documents
- Frequently Asked Questions (FAQ)
- Glossary

Unity Plugin Overview

If you are building a data-heavy game in Unity, managing that data through Inspector overrides or massive ScriptableObjects can quickly become unmanageable. The Charon Unity Plugin provides a professional-grade database workflow directly within your Unity project.



---

1. What is it?

The **Charon Unity Plugin** is an integrated extension for the Unity Editor that bridges the gap between game design and implementation. It combines a **structured data editor** (accessible via an embedded or external browser) with a **native code generator**.

Instead of manual data entry, it provides a centralized “Game Database” inside your project that behaves like a professional CMS but lives within your local assets.

2. Which problem does it solve?

Unity’s built-in tools are often insufficient for complex game data. Charon solves the most common “data-driven” headaches:

- **ScriptableObject Sprawl:** No more navigating hundreds of individual `.asset` files. All your items, NPCs, and quests are managed in one organized, searchable database.
- **Lack of Type Safety:** Charon automatically generates the C# classes needed to load your data. This means you get **full IntelliSense support**—no more searching for strings or worrying about typos in your code.
- **The “Designer-to-Developer” Gap:** It provides a user-friendly, non-technical UI for designers to balance numbers and write dialogue, while generating the clean, performant code that programmers expect.
- **Localization Bottlenecks:** It includes built-in support for multi-language text, allowing you to swap languages or export translation keys without leaving the editor.

3. For whom?

- **Unity Developers:** Who want to stop writing boilerplate data-loading code and start using type-safe, optimized data structures.
- **Game & Narrative Designers:** Who need a powerful, visual environment to model complex systems (like skill trees or quest branches) without touching a line of code.
- **Production Teams:** Working on RPGs, CCGs, Strategy games, or any project where thousands of balanced data points are the core of the experience.

Key Features

Feature	Description
In-Editor Server	Launch the data editor directly from the Unity menu bar.
Auto-Generation	C# code is updated automatically whenever you change your data schema.
Asset Integration	Link your game data directly to Unity Assets (like Prefabs, Sprites, etc.).

Memory Efficient	Optimized loading routines designed specifically for mobile/console performance.

Getting Started

Prerequisites

Unity plugin uses `dotnet charon` tool, which is a .NET Core application built for .NET 8.

Windows

1. Download and install NET SDK 8+.
2. Make sure you have write access to `%PROGRAMDATA%/Charon`.

MacOS

1. Download and install NET SDK 8+.
2. Make sure you have write access to `/Users/<username>/.config/Charon`.
3. Make sure `dotnet` is available from `$PATH`.

Linux

1. Download and install NET SDK 8+.
2. Make sure you have write access to `/usr/share/Charon`.
3. Make sure `dotnet` is available from `$PATH`.

Checking Available .NET SDK Versions

```
# check for dotnet already installed
dotnet --list-sdks

# output for dotnet --list-sdks
7.0.120 [C:\Program Files\dotnet\sdk]
8.0.206 [C:\Program Files\dotnet\sdk]
8.0.414 [C:\Program Files\dotnet\sdk]  # <- this one is fine
9.0.300 [C:\Program Files\dotnet\sdk]  # <- this one too
```

Microsoft.WindowsDesktop.App 9.0.0 [C:Program FilesdotnetsharedMicrosoft.WindowsDesktop.App]

Installation from OpenUPM (recommended)

1. Install the required software for your operating system.
2. Ensure your Unity version is 2021.3 or later.
3. Open the OpenUPM page for the plugin.
4. Click the **Manual Installation** button in the upper right corner and follow the instructions.

Installation from Unity Asset Store

1. Install the required software for your operating system.
2. Ensure your Unity version is 2021.3 or later.
3. Open the Charon plugin in the Unity Asset Store.
4. Click **Add To My Assets**.
5. Open the Unity Package Manager by navigating to **Window** → **Package Manager**.
6. Wait for the package manager to populate the list.
7. Select **My Assets** from the dropdown in the top left corner.
8. Select **Charon** from the list and click **Download**. If it's already downloaded, you will see an **Import** option.

Installation from GitHub

1. Install the required software for your operating system.
2. Clone or download the plugin source code from the GitHub repository.
3. Create a `<project-dir>/Packages/com.gamedevware.charon` directory.
4. Copy the plugin files from `src/GameDevWare.Charon.Unity/Packages/com.gamedevware.charon` into this directory.
5. Restart Unity if necessary.

Core Concepts

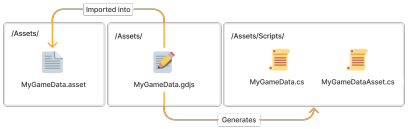
Data-Driven Design Principles

Data-driven design emphasizes controlling gameplay through data rather than source code or blueprints. Game mechanics and processes are determined by structured data files. For example, instead of embedding damage calculations directly in the game's source code, these are defined by data specifying weapon effects and the rules for their application. Similarly, mission progression is not hardcoded; it is outlined in editable text files, making these aspects of game design highly flexible. This approach not only facilitates quick adjustments during development but also simplifies adding modding support post-release.

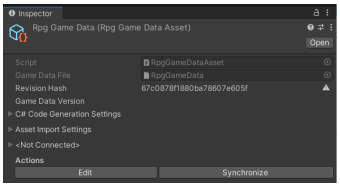
- Modify Everything! Data-Driven Dynamic Gameplay Effects in 'For Honor' (Video)
- Data-driven Design in Unreal (Article)

Understanding the Plugin's Architecture

Plugin Assets



All game data information is stored in a JSON file within your project. The generated source code is used to load this data into the game. Additionally, a `ScriptableObject` asset will be created, which can be used to access game data from your scenes.



Whenever there is a modification in the data structure within a JSON file, it is necessary to regenerate the C# source code and reimport the *.asset* file. To do this, select the *.asset* file and press the **Synchronize** button.

Working with the Plugin

Creating Game Data

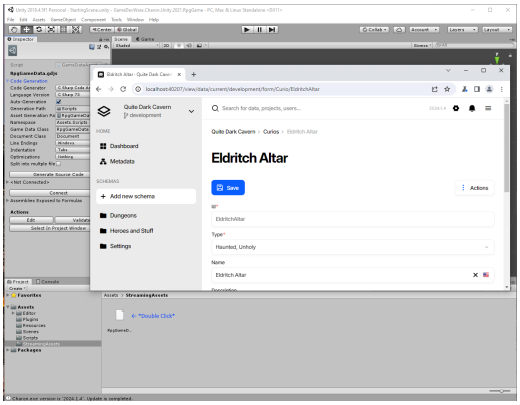
To create a new game data file within the Unity Editor, open the **Project** window, right-click in the desired folder, and select the **Create** → **Game Data** menu option.

1. Open the **Project** window and navigate to the desired folder.
2. Right-click in the **Project** window and select **Create** → **Game Data**.

3. Name your game data file and click the **Create** button.
4. Wait for the source code and assets to be created in the specified folder and for the editor to recompile the scripts.
5. Double-click the created *.asset* or *.gdjs* file to start editing.

Editing Game Data

To edit a game data file in the Unity Editor, open the **Project** window, find the corresponding *.gdjs*, *.gdmp*, or *.asset* file, and double-click it. This action opens a new web browser window featuring a user interface for editing the game data. Remember to **Synchronize** assets from the Inspector window after completing your edits.



Advanced Features

Localization and Multi-Language Support

Charon facilitates multi-language text support through the `Localizable Text` data type. When creating a *Schema*, properties can be defined with various data types, including `Localizable Text`. Initially, all localizable text defaults to `EN-us` (US English). Additional languages can be added via **Project Settings** → **Internationalization** → **Translation Languages** in the Charon UI.

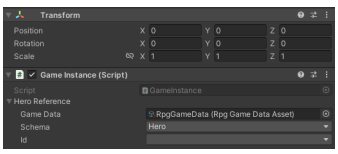
Exporting/importing localizable data.

Referencing Game Data in Scenes

The Charon plugin introduces a specific type for referencing documents within scenes, named `GameDataDocumentReference`. This type is part of the Charon package. To create such a reference, add a field with the `GameDataDocumentReference` type to your component class.

```
public class HeroComponent : MonoBehaviour
{
  public GameDataDocumentReference heroReference;
}
```

You can then configure it in the Inspector. Here is an example of a **Game Data Document Reference** used to point to a *Hero* document:



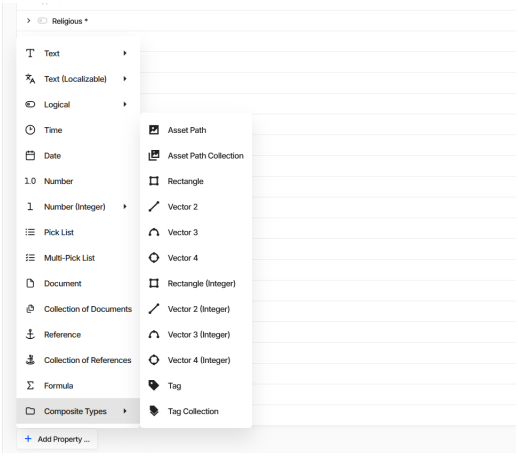
To get an instance of a document in your game code, call the `GameDataDocumentReference.GetReferencedDocument<Hero>()` method.

```
private void OnEnable()
{
  var hero = this.heroReference.GetReferencedDocument<Hero>();
```

```
    Debug.Log(hero.Name);
}
```

Referencing Unity Assets

To reference assets within the game, you can use a special Asset composite type. Create a property via `Composite Types` -> `Asset Path` menu options.



Work & Build Automation

To facilitate automation of work or builds, a programmatic interface for working with game data is provided. You can read more about it on the CharonCli class documentation page.

Feedback

We welcome and encourage feedback, particularly bug reports and suggestions, to help improve our tool. If you have any questions or would like to share your thoughts, please join our Discord community or reach out to us via email at support@gamedevware.com.

See also

- Basic Navigation and User Interface Overview
- Creating Document Type (Schema)
- Filling Documents
- Frequently Asked Questions (FAQ)
- Glossary

CharonCli Overview

The CharonCli class provides a convenient interface for running `dotnet charon` command-line operations. It simplifies interactions with the Charon tool, enabling developers to manage game data, automate workflows, and integrate with Unity projects. Below is an overview of its methods grouped by purpose.

Game Data Management

- **InitGameDataAsync**: Initializes a GameData file.
- **CreateDocumentAsync**: Creates a document in the specified GameData URL.
- **UpdateDocumentAsync**: Updates a document in the specified GameData URL.
- **DeleteDocumentAsync**: Deletes a document in the specified GameData URL (by document or ID).
- **FindDocumentAsync**: Finds a document in the specified GameData URL by ID.

- **ListDocumentsAsync**: Lists documents in the specified GameData URL with optional filters and sorting.

Import and Export

- **ImportAsync**: Imports documents grouped by schema into a specified GameData URL.
- **ImportFromFileAsync**: Imports documents from a file into a specified GameData URL.
- **ExportAsync**: Exports documents from a GameData URL.
- **ExportToFileAsync**: Exports documents from a GameData URL to a file.

Localization (I18N)

- **I18NImportAsync**: Imports translated documents grouped by schema into a specified GameData URL.
- **I18NImportFromFileAsync**: Imports translated documents from a file into a specified GameData URL.
- **I18NExportAsync**: Exports documents for localization from a GameData URL.
- **I18NExportToFileAsync**: Exports documents for localization from a GameData URL to a file.
- **I18NAddLanguageAsync**: Adds translation languages to a GameData URL.

Patching and Backup

- **CreatePatchAsync**: Compares documents in two GameData URLs and creates a patch representing the difference.
- **CreatePatchToFileAsync**: Compares documents in two GameData URLs and saves the patch to a file.
- **ApplyPatchAsync**: Applies a patch to a specified GameData URL.
- **ApplyPatchFromFileAsync**: Applies a patch from a file to a specified GameData URL.
- **BackupAsync**: Backs up game data with all documents and metadata.
- **BackupToFileAsync**: Backs up game data to a file with all documents and metadata.
- **RestoreAsync**: Restores game data from a backup.
- **RestoreFromFileAsync**: Restores game data from a backup file.

Validation and Code Generation

- **ValidateAsync**: Validates all documents in a GameData URL and returns a report with issues.
- **GenerateCSharpCodeAsync**: Generates C# source code for loading game data into a game's runtime.
- **DumpTemplatesAsync**: Dumps T4 code generation templates into a specified directory.

Tool Utilities

- **GetVersionAsync**: Gets the version number of the Charon tool executable.
- **GetGameDataToolVersionAsync**: Gets the version of the Charon tool used to create a GameData URL.
- **RunCharonAsync**: Runs a specified command with the Charon tool.
- **RunT4Async**: Processes T4 templates using the `dotnet-t4` command-line tool.

See also

- Unity Plugin Overview
- CharonCli class
- Examples of CharonCli class

Migration from Legacy Version (Before 2025.1.*)

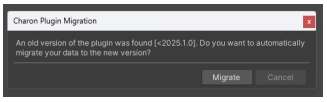
Warning

Before proceeding with the migration, ensure your project is under a source control system (e.g., Git) or that a full backup of your project has been created. Migration involves modifying and deleting files, and having a backup or version control ensures you can recover in case of unexpected issues.

Install the package with the new version of the plugin via the Unity Asset Store or using OpenUPM (recommended). After installing plugin package you have two options:

Automated Migration

A window will appear offering to perform the migration automatically.



1. Click the **Migrate** button and wait for the process to complete.
2. Once the migration is finished, close the window if everything is successful.
3. If an error occurs, check the **Console** window for details and consider using the *Manual Migration* approach.

Manual Migration

To migrate manually, you will need to remove the old plugin, convert, and configure the game data files:

1. Navigate to the `Assets/Editor/GameDevWare.Charon` folder and delete it.
2. Temporarily move all **.gdjs** and **.gdmp** files from `Assets/StreamingAssets/` to `Assets/`.
3. Select each **.gdjs** or **.gdmp** file and click the **Reimport** button in the **Inspector** window.
4. Replace the old **.asset** file with the newly generated one. If the file did not exist previously, place it anywhere within the boundaries of the **.asmdef** file.
5. Replace the old source code files (**.cs**) with the newly generated ones.

Warning

Preserve the original **.meta** files for **.cs** and **.asset** assets to maintain Unity resource associations and links.

See also

- Unity Plugin Overview

Migrating to Web Application

To migrate to the https://charon.live, you can do it through a *backup <../web/migrating_to_web>* or through the “Connection” mechanism.

In short: you need to create an empty project in at https://charon.live, in Unity Editor in Inspector window click **Connect**, and specify that you want to upload data to the https://charon.live.

Migration with Connection

Be sure to back up your local data before making any connections.

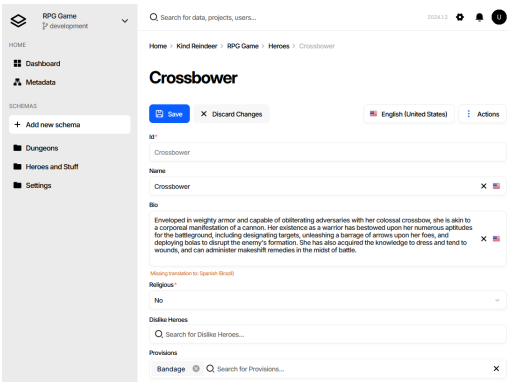
1. At https://charon.live: on the Home screen, click on **Create Project**.
2. Specify the project name, tags, and script language.
3. Click the **Create** button.
4. In the Unity Editor: select your game data **.asset** file in the **Project** window.
5. In the **Inspector** window, expand **<Not Connected>** foldout, click **Connect** button.
6. In the dialog that opens, click on the **Profile** → **API Keys** link.
7. At https://charon.live: a page of your profile on the *API Keys* management page should have opened in your browser.
8. Click **Create API Key…** button.
9. Fill in the name and expiration time, then click the **Create** button.
10. Click the **Copy** button in the list of keys next to the newly created key labeled “New!”.
11. In the Unity Editor: paste the *API Key* into the corresponding field in the **Connect Game Data** window.
12. Check the *Upload local data…* checkbox, it is only available when the selected *Project* is empty and does not contain any data.
13. Click the **Upload** button”.
14. Close **Connect Game Data** window

See also

- **`Basics`**
- Charon Website

Standalone & CLI Overview

For developers who require maximum control over their environment, Charon offers a flexible standalone distribution. This version is built for those who prefer working outside of a specific game engine or need to automate their data workflows.



---

1. What is it?

The standalone version is a **cross-platform .NET 8 CLI application**. It operates as a dual-purpose tool:

- **Headless CLI:** A powerful command-line interface for data manipulation and code generation.
- **Local HTTP Server:** By launching a local server, it provides a high-performance **browser-based UI** for visual data modeling and editing without requiring an internet connection.

2. Which problem does it solve?

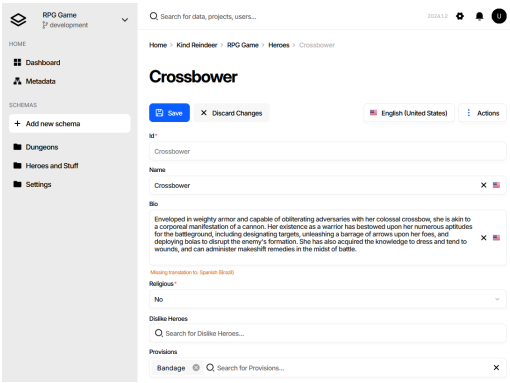
While the engine plugins are great for standard development, the Standalone/CLI version addresses specialized technical requirements:

- **Custom Pipeline Integration:** Easily integrate game data validation and code generation into custom build scripts or tools.
- **CI/CD Automation:** Run data integrity checks and generate source code automatically on build servers (GitHub Actions, Jenkins, etc.).
- **Modding Toolsets:** Distribute a lightweight, engine-agnostic tool to your players, enabling them to create and edit game content safely.
- **Engine-Agnostic Workflows:** Work on game data for custom engines or projects that don't use Unity or Unreal.

3. For whom?

This version is tailored for **advanced users and technical leads** who are comfortable with the terminal. It is ideal for:

- **Tools Engineers** building custom development pipelines.
- **Technical Designers** who need a standalone environment for content creation.
- **Build Engineers** automating game data processing in the cloud.



Prerequisites

Standalone application uses *dotnet-charon <https://www.nuget.org/packages/dotnet-charon>* tool, which is a .NET Core application built for .NET 8.

Windows

1. Download and install NET 8+.
2. Make sure you have write access to `%PROGRAMDATA%/Charon`.

MacOS

1. Download and install NET 8+.
2. Make sure you have write access to `/Users/<username>/.config/Charon`.
3. Make sure `dotnet` is available from `$PATH`.

Linux

1. Download and install NET 8+.
2. Make sure you have write access to `/home/<username>/.config/Charon`.

3. Make sure `dotnet` is available from `$PATH`.

Checking Available .NET Versions

```
# check for dotnet already installed
dotnet --list-runtimes

# output for dotnet --list-runtimes
Microsoft.AspNetCore.App 6.0.36 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 7.0.20 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 8.0.6 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 9.0.0 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.NETCore.App 6.0.36 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 7.0.20 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 8.0.6 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App] # <- this
Microsoft.NETCore.App 9.0.0 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App] # <- this
```

Installation and Updates

You can use just two commands to install the command line tool, or use a bootstrap script that will check dependencies and installed software, and then download and run the tool for you.

dotnet tool (recommended)

```
# install charon globally (run it once)
dotnet tool install -g dotnet-charon

# update global tool
dotnet tool update -g dotnet-charon

# make empty game data file
 charon INIT ./gamedata.json

# run editor
 charon ./gamedata.json
```

Bootstrap Script

Two bootstrap scripts which download and run latest version of Charon on your PC:

- `RunCharon.bat` for Windows
- `RunCharon.sh` for Linux or MacOS

> **Warning**
>
> Bootstrap scripts require **.NET SDK■** to run, not only bare .NET Runtime.

Both scripts require the dotnet tool to be available in `PATH`.

1. **Download one of the scripts into a local folder** `charon`**.**
    a. RunCharon.bat (Windows)
    b. RunCharon.sh (Linux, MacOS)
2. Navigate to the local folder `cd charon`.
3. Run `RunCharon.bat` or `RunCharon.sh` depending on your OS.
4. Wait for the script to automatically download and upgrade `dotnet-charon tool`, and display help text.

5. Create an empty file named `RunCharon.bat INIT gamedata.json`
6. Run in standalone mode: `RunCharon.bat gamedata.json`

Or use following bootstrap script:

Windows

```
rem ##### Load and run bootstrap script #####

@echo off
mkdir Charon
cd Charon
curl -O https://raw.githubusercontent.com/gamedevware/charon/main/scripts/bootstrap/RunCharo
./RunCharon.bat INIT ./gamedata.json

rem ##### Start editor #####

./RunCharon.bat ./gamedata.json --log out
```

Linux, MacOS

```
##### Load and run bootstrap script #####

mkdir Charon
cd Charon
curl -O https://raw.githubusercontent.com/gamedevware/charon/main/scripts/bootstrap/RunCharo
chmod +x RunCharon.sh
./RunCharon.sh INIT ./gamedata.json

##### Start editor #####

./RunCharon.sh ./gamedata.json --log out
```

Creating and Editing Game Data

Any empty **gamedata.json** file could be used as starting point for standalone application launch. The editor will automatically fill the empty file with the initial data.

Windows

```
# charon INIT .\gamedata.json
# or
# copy /y NUL .\gamedata.json >NUL

charon .\gamedata.json --log out
```

Linux, MacOS

```
# charon INIT ./gamedata.json
# or
# touch ./gamedata.json

charon ./gamedata.json --log out
```

After finishing your work, you could just terminate the process with `CTRL+C` keyboard shortcut or close terminal window.

See also

- Nuget Package
- Bootstrap Scripts
- Basic Navigation and User Interface Overview

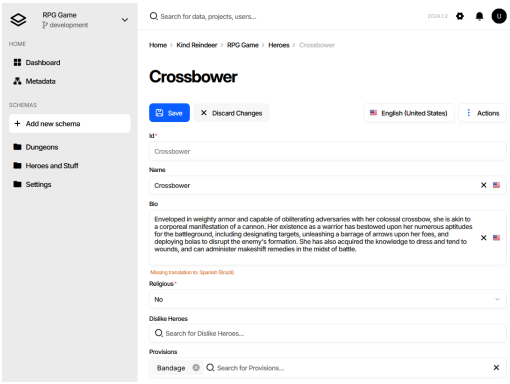
- Creating Document Type (Schema)
- Filling Documents
- Publication of Game Data
- Generating Source Code
- Frequently Asked Questions (FAQ)
- Glossary

Web Application Overview

The web version of the Charon provides a collaborative work environment where game designers can work together to create engaging gameplay experiences. The core concepts of collaborative work include workspaces and projects. A workspace is a virtual location where projects are located. The subscription and all limitations are bound to the workspace, meaning that all projects within the workspace are subject to the same subscription and limitations.

A project is a virtual location for storing game data, localization settings, backups, branches, and members. When a project is created,

the user becomes its owner and can invite other members to join.



Starting with a new Project

1. Visit the charon.live website and click on the *Register* button to create a new account.
2. Fill out the registration form with your desired username and password, and click on the “Create” button to create your account.
3. After successfully logging in, you will be directed to the workspace page.
4. If you’re a new user, the workspace page will be empty, with no projects listed. Click on the *Create project* button to create your first project.
5. On the “Create Project” page, fill in the name of your project and any other basic information you want to include, and click on the “Create” button to create your project.
6. After creating the project, you’ll be redirected to the project’s dashboard page, which provides an overview of the project and allows you to start modelling game data.

See also

- Charon Website
- CLI Access to Web Project
- Migrating to Web Application
- Basic Navigation and User Interface Overview
- Creating Document Type (Schema)
- Filling Documents

- Publication of Game Data
- Generating Source Code
- Working with Source Code (C# 4.0)
- Working with Source Code (C# 7.3)
- Working with Source Code (TypeScript)
- Frequently Asked Questions (FAQ)
- Glossary

CLI Access to charon.live

The web version of the Charon provides a REST API and CLI for accessing and modifying game data. To access the API, users need to generate an `API Key` in the *API Keys* section of their *User Profile*.

With the API Key, Charon can easily be integrated into existing game development workflows. For example, the API Key can be used to export game data from the web into a local GIT repository

Step By Step

To generate an API Key:

1. Navigate to the *API Keys* section and click on the *Generate API Key…* button.
2. Copy the generated `API Key`.
3. Use the `API Key` in *the `Authenticate`* header to access the REST API or in the `--credentials` parameter of the CLI.

See also

- Command Line Interface (CLI)
- Migrating to Web
- REST API
- DATA EXPORT Command
- DATA IMPORT Command
- GENERATE CSHARPCODE Command

Migrating to Web Application

To migrate to the web application, you only need to backup your game data in your current editor and restore it in the web application.

In short: project settings include a backup and restore feature. You make a backup in one place, and you can restore it in another. This is how you can transfer game data to another project.

Backup Data Step by Step

1. Open your game data in the editor (Standalone, Unity …)
2. In the bottom left corner, click on the gear icon “Settings”
3. In the left menu, select “Backup”
4. On the backup management page, click the “Backup” button
5. Choose “File” as the destination to save the backup and click “Next”
6. On the “Format” step, select “JSON” and click “Backup”

7. On the “Summary” step, click the link with a file name like `backup_2023_11_10_11_27.json` and save it to your computer.

Restoring Backup in the Web Application

1. On the Home screen, click on “Create Project”
2. Specify the project name, tags, and script language
3. Click the “Create” button
4. In the left menu, select “Backup”
5. On the backup management page, click the “Restore” button
6. Choose “File” as the source of the restore and click “Next”
7. Select the input file, the one you have after the backup, such as `backup_2023_11_10_11_27.json`, and click “Restore”
8. Done

Any data that was in this project at the time before the restore will be lost.

See also

- Basic Navigation and User Interface Overview
- Publication of Game Data
- Generating Source Code
- Frequently Asked Questions (FAQ)
- Glossary

Roles and Permissions

The web version of the Charon provides a system of roles and permissions to manage access and control over your game development projects. Each role is designed to provide specific levels of access and functionality within the platform. Here are the key roles , along with their respective permissions:

Viewer Role:

- View Documents: Users with the Viewer role can access and view documents within projects.
- Export Data: Viewers can export data from the platform.
- Access Project Settings: They can access and view project settings.

Editor Role:

- View Documents: Editors can view documents.
- Edit Documents: Editors have the ability to edit documents within projects.
- Import Data: They can import data into the platform.
- Access Project Settings: Similar to Viewers, Editors can access and view project settings.

Designer Role:

- Change Document Structure: Designers have the privilege to modify the structure of documents.
- View Documents: Designers can view documents.
- Edit Documents: They can edit documents.
- Import Data: Designers can import data.
- Access Project Settings: They can access and view project settings.

Administrator Role:

- Make and Restore Backups: Administrators have the authority to create and restore backups of project data.
- Grant or Revoke Permissions: They can grant or revoke permissions for users within the project.
- Change Project Settings: Administrators can modify various project settings to tailor the environment to their needs.
- View Documents: Administrators can view documents.
- Edit Documents: They can edit documents.
- Import Data: Administrators can import data.

Workspace Administrator and Workspace Owner Role:

- They have the same permissions as Administrators, and they also have the ability to delete and transfer projects.

See also

- Overview
- Web-based Application

REST API

The web version of the Charon provides a experimental REST API feature.

Testing REST API

You can utilize the Swagger UI to perform test requests. In the *Swagger UI*, click on the *Authorize* button and paste your API Key for authentication.

Working with REST API

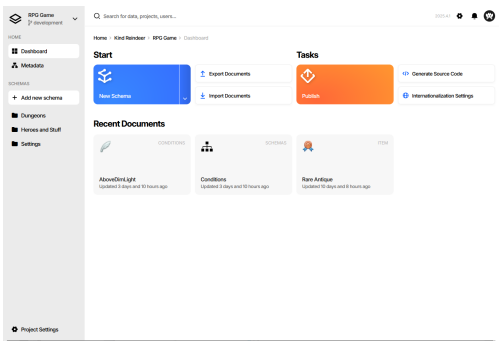
To make requests, you will need an API Key obtained from your profile page. Add the Authorization: `Basic <api-key>` header to all of your HTTP requests. Also is recommeded to provide correct User-Agent header.

Basic Navigation and User Interface Overview

The UI consists of a left-side menu displaying all schemas of the game data, a middle working area with a dashboard/document list or document form, and a headline on the top with the project name and settings button. Depending on the installation, the UI may also include a user menu.

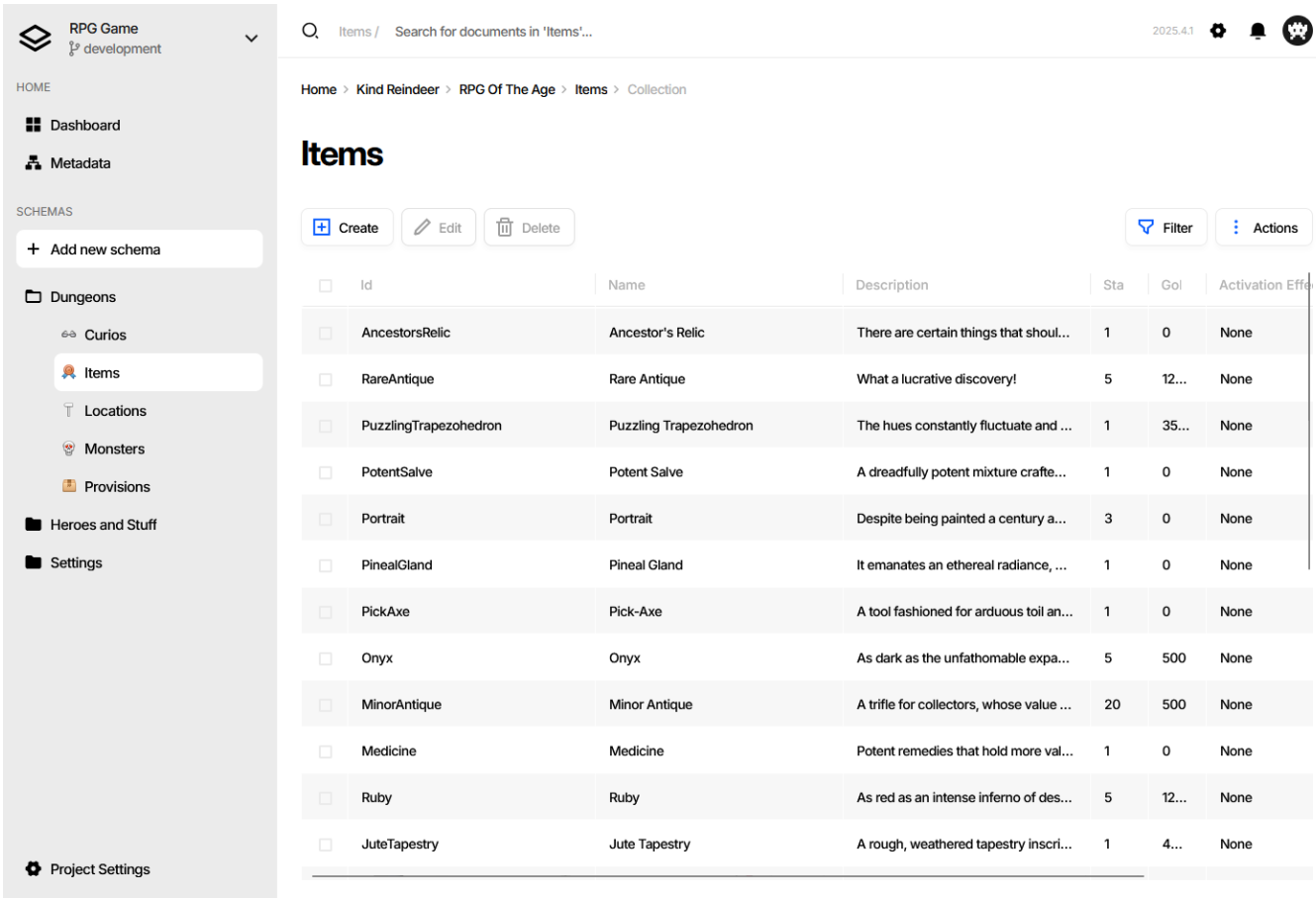
Dashboard

The dashboard is a central hub in the game data's user interface that provides quick access to frequently used features. It includes quick action buttons, such as creating a new schema, export, import, as well as a list of recently visited documents. Additionally, in the case of a web application, the dashboard may display the presence of online members who are currently working in the same project.



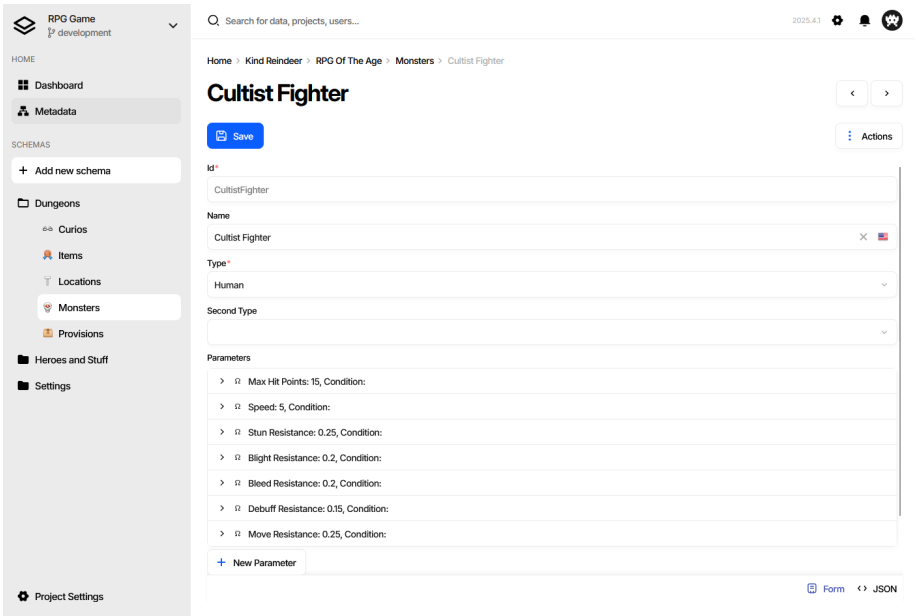
Document Collection

The document collection page is place where user can view a list of all the documents of a specified schema. This page allows users to filter, sort, and customize the list to their liking, making it easier to find the specific document they need.



Document Form

The document form page provides a specific edit form for a selected document. Here, users can view, edit, and save their game data documents in a structured and organized manner. The form allows users to input data into fields that correspond to the schema's properties. The document form page provides a user-friendly interface for updating and modifying game data.



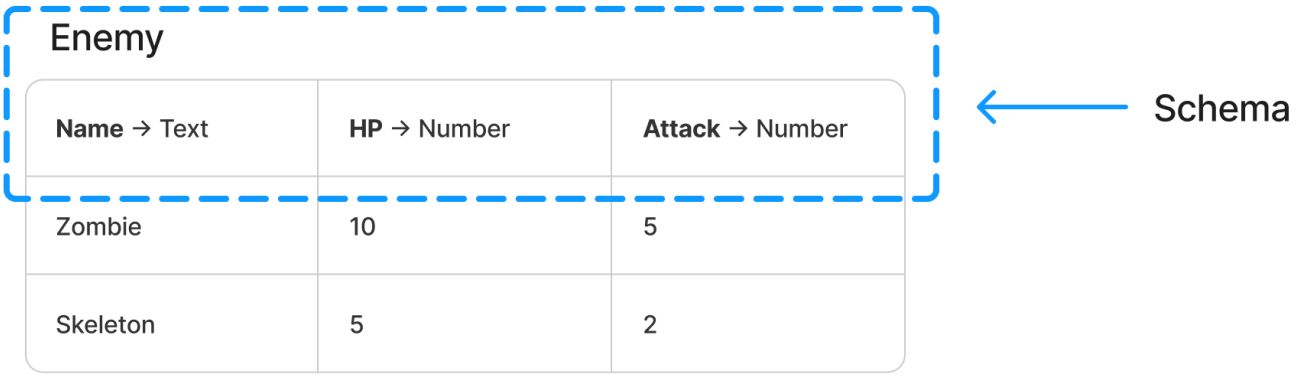
See also

- Creating Document Type (Schema)
- Filling Documents
- Publishing Game Data
- Generating Source Code

Creating Document Type (Schema)

Schema

Schema is essential for organizing and defining game data in a structured framework. In the context of game data modeling, a schema serves as a blueprint or template that establishes the structure and properties of a particular type of data entity in a game. It defines the columns or fields that represent the various attributes of the entity, similar to how a table has columns or a spreadsheet has cells.



Benefits of Structured Data

Data Organization

The organization of game data into logical entities and attributes is facilitated by the schema. This blueprint or template defines the structure, properties, and relationships of different entities and attributes within the game data, ensuring efficient storage, retrieval, and management.

Data Validation

The integrity and adherence to predefined rules of the game data are ensured through the validation capabilities of the schema. Constraints and validations, such as data types, allowed values, and dependencies, can be defined, preventing errors and inconsistencies in the game data.

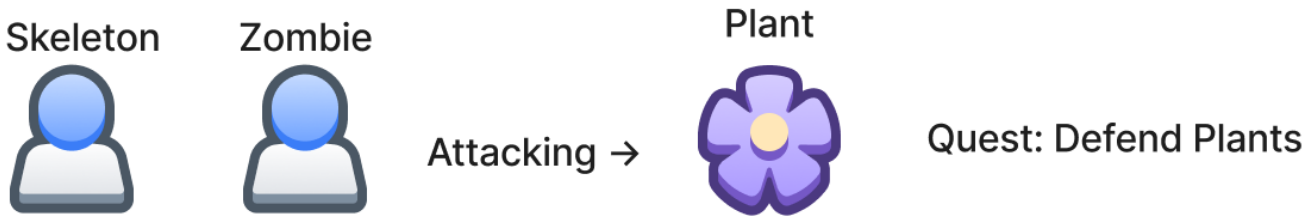
Data Consistency

Consistency across the game data is achieved through the standardized structure and rules provided by the schema. It enforces a consistent naming convention, attribute definitions, and relationships between entities, thereby enhancing coherence and simplifying collaboration.

Data Interoperability

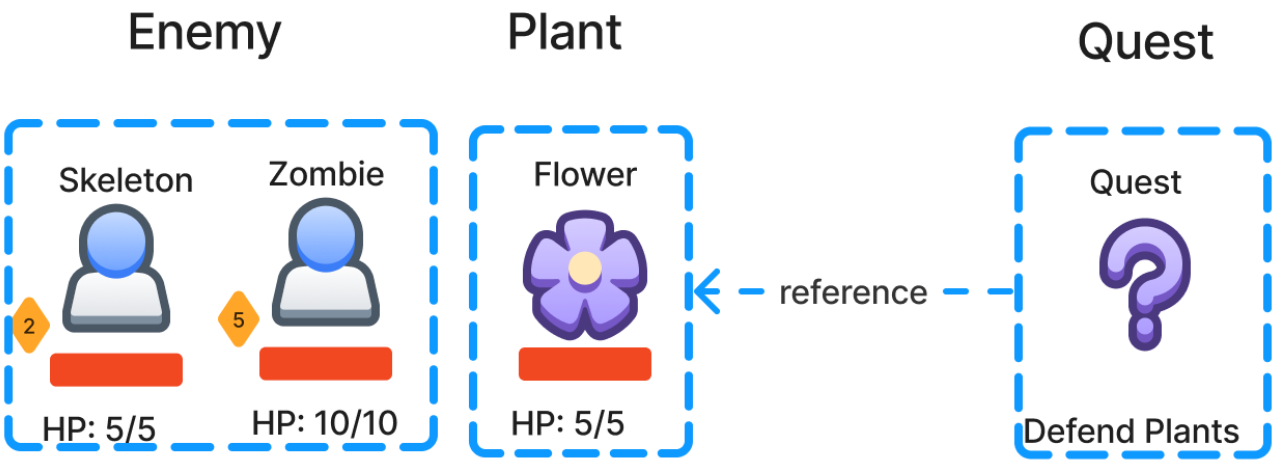
Interoperability and integration with external systems or tools are facilitated by a well-defined schema. By establishing a common language and structure, the schema enables seamless data exchange and collaboration with localization tools, analytics platforms, and asset pipelines.

Analyzing Game Requirements



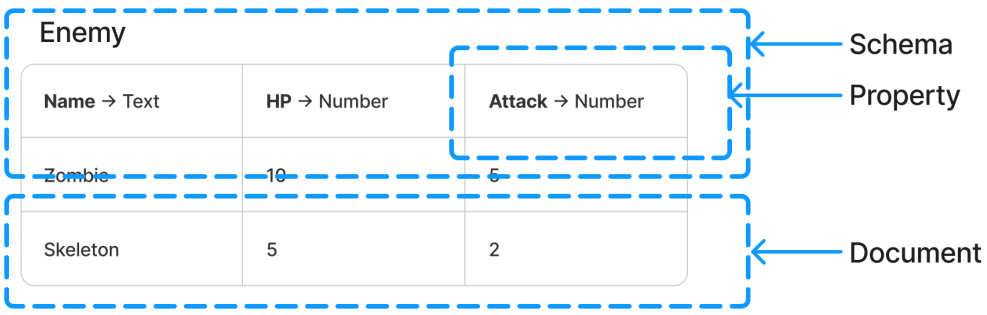
This step involves analyzing the game requirements to understand the design and functionality of the game. It includes studying the game design document and identifying key features, gameplay mechanics, and data elements that need to be captured and represented in the game.

Identifying Schemas and Relationships



In this step, schemas and their relationships within the game are identified. Schemas can be objects, characters, locations, items, quests, or any other significant element in the game. Relationships define how these entities are connected or interact with each other.

Defining Schemas and Properties



This step involves defining schemas to represent the structure and properties of the game data. A schema serves as a blueprint or template for a specific type of data entity, specifying its properties, attributes, and relationships. Properties describe the characteristics and attributes of an entity, such as its name, description, stats, or any other relevant information.

Data Types

Date

The `Date` data type is used to store dates in ISO 8601 format, which includes the year, month, day, and time with UTC time zone. This data type is particularly useful for storing information about events that occur on specific dates or for tracking the age of entities. Since dates are stored with UTC time zone, the data can be consistently interpreted across different time zones.

Source Code Type

Language	Type
C#	System.DateTime
TypeScript	Date
C++ (Unreal Engine)	FDateTime
Haxe	Date

Uniqueness

Date cannot be marked as unique.

Format

`yyyy-MM-ddTHH:mm:ss.fffZ`

Example

```
"2017-12-27T00:00:00.000Z"

 // it is better not to store dates before this mark for compatibility reasons
"1970-01-01T00:00:00.000Z"
```

Document

The `Document` data type in game data schema is used to represent complex structures. A document can contain multiple properties of different data types, including other documents or document collections, allowing for hierarchical data modeling. It is important to note that the lifetime of sub-documents is tied to the lifetime of the parent document, meaning that any changes (e.g. deletion) to the parent document will affect all of its sub-documents.

Source Code Type

Language	Type
C#	`class`
TypeScript	`class`
C++ (Unreal Engine)	`UCLASS`
Haxe	`class`

Uniqueness

Document Text cannot be marked as unique.

Example

For example, in a **Dialog**, each node can be a `Document` with dialog text, response options, and actions that occur after a response is chosen. Each response option can be a sub-document that is another **Dialog** node.

```
{
    "Text": "Welcome to the game. What's your name?",
    "Options": [
        {
            "Text": "My name is John.",
            "Options": [
                {
                    "Text": "Hello John! What brings you here?",
                    "Options": [
                        {
                            "Text": "I'm looking for adventure.",
                            "Action": "dialog.GiveQuestAndEnd()"
                        },
                        {
                            "Text": "I'm on a mission.",
                            "Options": [/* ... */]
                        }
                    ]
                }
            ]
        },
        {
            "Text": "I prefer not to say.",
            "Action": "dialog.End()"
        }
    ]
}
```

Document Collection

The DocumentCollection data type is used to store an array of sub-documents, which are used to represent complex structures. It is important to note that the lifetime of sub-documents is tied to the lifetime of the parent document, meaning that any changes (e.g. deletion) to the parent document will affect all of its sub-documents.

Source Code Type

Language	Type
C#	ReadOnlyCollection{T}
TypeScript	ReadOnlyArray{T}
C++ (Unreal Engine)	TArray{T}
Haxe	ReadOnlyArray{T}

Uniqueness

Document Collection Text cannot be marked as unique.

Size

May be limited in number of items. 0 - no limit.

Example

One example use case for `DocumentCollection` is storing a list of items in a game, such as a chest and its contents. Each item in the chest could be represented by a sub-document containing information such as reference to an item and its quantity.

```
{
    "Name": "Silver Chest",
    "Loot": [
        {
            "Item": { "Id": "Sword" },
            "Quantity": 1
```

```
        },
        {
            "Item": { "Id": "Silver" },
            "Quantity": 100
        }
    ]
}
```

Formula

The **Formula** data type allows storing and evaluating C# expressions within game data. It enables developers to define logic that performs calculations based on dynamic inputs and parameters not known until runtime. A formula can be any valid C# expression that returns a value of a supported data type.

Formulas are especially useful in scenarios involving complex calculations, such as determining weapon damage based on factors like target resistance and attack type. By encapsulating such logic in formulas, developers can fine-tune gameplay mechanics without modifying or recompiling the source code.

At runtime, formulas are evaluated using arguments passed to them. These arguments can reference other data properties or objects, providing a high level of flexibility in defining game rules and balancing.

For example, a damage formula might look like this:

```
(weaponPower * (1.0 - targetResistance)) * attackMultiplier
```

This expression multiplies weapon power by a factor based on the target's resistance and then scales the result by an attack multiplier. The final result is the computed damage.

Source Code Type

Language	Type
C#	`class`
TypeScript	`class`
C++ (Unreal Engine)	`UCLASS`
Haxe	`class`

Uniqueness

Formulas cannot be marked as unique.

Specification

Formulas support the following specification parameters: - `param.<name>` — Defines a formula parameter and its type.

Example: `param.id=System.Int32`

- `resultType` — Specifies the return type of the formula.

  Example: `resultType=System.String`

- `knownType` — Declares types known to the formula. Only static members of known types can be accessed.

  Example: `knownType=System.FMath`

- `typeName` — Custom type name for the generated class. If omitted, a name is derived from the containing schema and property.

  Example: `typeName=MyDamageCalculatingFormula`

Example

```
"target.HP < 100"
"x != 0"
"target.DoDamage(100)"
"(weapon.Damage / target.DamageResistance) / 2"
```

Integer

The `Integer` data type is a whole number data type that is limited to 64 bits. It is used to represent integers without a fractional component. It can be used in cases where you need to store a large range of positive or negative whole numbers, such as in-game currency or player levels.

Unlike the Number data type, integers do not have any precision caveats since they do not store decimal values. Therefore, they are suitable for calculations that require exact values.

Source Code Type

Language	Type
C#	System.Int32 or System.Int64
TypeScript	number or bigint
C++ (Unreal Engine)	int32 or int64
Haxe	Int or Int64

Uniqueness

Integers can be marked as unique.

Size

32 or 64bit

Example

```
0
-1
100
```

Localized Text

The `LocalizedText` data type is used to store text that needs to be displayed in multiple languages. Unlike the Text data type, the LocalizedText data type allows the storage of the same text in multiple languages. It supports the whole range of UTF symbols, just like the Text data type. The `LocalizedText` data type is essential for games that require localization support, and it makes it easy for game developers to manage text that needs to be displayed in multiple languages.

Source Code Type

Language	Type
C#	LocalizedText or System.String
TypeScript	LocalizedText or string
C++ (Unreal Engine)	FLocalizedText or FText
Haxe	LocalizedText or String

Uniqueness

Localized Text cannot be marked as unique.

Size

May be limited in number of characters. 0 - no limit.

Example

```
{"en-US": "Hello", "fr-FR": "Bonjour"}
```

Logical

The `Logical` data type is used to represent boolean values — values that can be either `true` or `false`. It is suitable for modeling simple binary states, conditions, or flags commonly used throughout game data.

Use Cases

The `Logical` type is ideal for:

- Representing **simple state**: Examples include *IsActive*, *IsPublished*, *IsLatest*, *IsReligious*, or *IsOptional*.
- Indicating **ownership or structural conditions**: Examples include *HasPartner*, *HasSecondPhase*.
- Controlling **behavioral expectations**: Examples include *ShouldFinishTutorial*, *ShouldGoFirst*.
- Defining **capabilities or permissions**: Examples include *CanBlock*, *CanRun*, *CanBeSold*.

This makes the `Logical` type a versatile and lightweight choice for many decision-driven properties in schema design.

Comparison with MultiPickList

While the MultiPickList data type can represent combinations of multiple states, the `Logical` type is recommended when the property has exactly two possible values. Using `Logical` ensures simplicity, clarity, and optimal editor behavior for binary flags.

Source Code Type

Language	Type
C#	System.Boolean
TypeScript	boolean
C++ (Unreal Engine)	bool
Haxe	Bool

Uniqueness

Logical can be marked as unique.

Example

```
true
false
True
False
1
0
"Yes"
"No"
```

Multi-Pick List

The `MultiPickList` data type allows a property to represent multiple values selected from a predefined list of options. It is designed for cases where multiple attributes, traits, or capabilities can be applied to a single entity simultaneously. Unlike PickList, which limits the property to a single choice, `MultiPickList` enables the combination of several boolean-like flags into a single compact representation. In the UI, MultiPickList is typically presented as a dropdown menu, allowing users to select or deselect multiple options independently.

Storage on Disk

On disk, the `MultiPickList` is stored as an integer bitmask. Each option is assigned a specific bit position. When multiple options are selected, their bits are combined using bitwise OR operations. The number of options is limited by the bit width of the storage format—commonly 32 or 64 options for a 32-bit or 64-bit integer, respectively.

Example

An item in an RPG might be marked as equippable, sellable, and destructible using a single MultiPickList field instead of three separate boolean flags.

Source Code Type

Language	Type
C#	`[Flags] enum`
TypeScript	`enum`
C++ (Unreal Engine)	`UENUM(meta = (Bitflags, UseEnumValuesAsMaskValuesInEditor = "true"))`
Haxe	`abstract`

Uniqueness

Multi-Pick Lists can be marked as unique.

Size

32 or 64bit

Specification

Multi-Pick Lists support the following specification parameters:

- `typeName` — Custom type name for the generated class. If omitted, a name is derived from the containing schema and property.

  Example: `typeName=MyEnum`

Example

```
1 // internaly stored as integers
"Apple" // string values also valid
```

Number

The `Number` data type is used to represent decimal numbers. It conforms to the IEEE 754 floating-point standard and can represent both positive and negative numbers, as well as zero. However, due to the limitations of the floating-point representation, precision may be lost when performing certain arithmetic operations. Therefore, it is recommended to use the Integer data type for financial calculations and other scenarios that require high precision.

Some use cases for the `Number` data type include representing quantities, such as the count of an items or the amount of gold reward in the chest, or representing percentages, such as the chance of an event occurring. It can also be used to represent measurements, such as the height of a character or the length of a weapon.

When working with `Numbers` in game data, it is important to ensure that the precision is appropriate for the use case. Additionally, it may be necessary to round numbers to a certain number of decimal places to avoid displaying unnecessarily precise values to players.

Source Code Type

Language	Type
C#	System.Single or System.Double
TypeScript	number
C++ (Unreal Engine)	float or double
Haxe	Float

Uniqueness

Numbers can be marked as unique.

Size

32 or 64bit

Specification

Numbers support the following specification parameters:

- `precision` — Specify decimal precision of stored number.

  Example: `precision=2`

- `displayPrecision` — Specify decimal precision of displayed in UI number.

  Example: `displayPrecision=2`

Example

```
3.14
0.21
-3.14
```

Pick List

The `PickList` data type defines a property that can take one value from a predefined set of options. It is best suited for fields where only a single choice is valid at any given time. This ensures consistency and restricts inputs to known, valid values. This type is commonly used for categorical selections such as character class, item rarity, or region. By standardizing the input, PickList helps reduce errors and supports better validation and filtering within game systems. In user interfaces, a `PickList` is typically presented as a dropdown menu, allowing users to choose only one option from the list.

Storage on Disk

On disk, `PickList` values are stored as integer. This value correspond directly to one of the predefined options.

Example

In a character creator, the player might choose “Mage” as their class from a list of available options like Warrior, Mage, or Rogue.

Source Code Type

Language	Type
C#	`enum`
TypeScript	`enum`
C++ (Unreal Engine)	`UENUM`
Haxe	`abstract`

Uniqueness

Pick Lists can be marked as unique.

Size

32 or 64bit

Specification

Pick Lists support the following specification parameters:

- `typeName` — Custom type name for the generated class. If omitted, a name is derived from the containing schema and property.

  Example: `typeName=MyEnum`

Example

```
1 // on-disk stored as integers
"Apple" // string values also valid
```

Reference

The `Reference` data type enables the creation of non-embedded relationships between documents. A reference acts as a pointer to another document by using that document’s `Id` as a key. This approach facilitates linking between related documents without the need to duplicate or embed data.

When using a `Reference`, the target documents are not stored within the parent document but are instead referenced externally. This is particularly beneficial when working with large or complex data sets, where referencing

is more efficient and maintains separation of concerns. It also promotes data integrity by ensuring that only valid document references are maintained.

For example, in a game scenario, a **Chest** with a loot table might hold a reference to an **Item** document, rather than embedding the entire **Item** directly within the **Chest**. This separation allows independent management of the loot logic and item definitions while preserving their relationships.

Source Code Type

Language	Type
C#	DocumentReference{T} or T
TypeScript	DocumentReference{T} or T
C++ (Unreal Engine)	FGameDataDocumentReference
Haxe	DocumentReference{T} or T

Uniqueness

Reference cannot be marked as unique.

Specification

References support the following specification parameter:

- `displayTextTemplate` — Defines a template string for how the referenced value is displayed in the UI.

  Example: `displayTextTemplate=Item%3A+%7BName%7D%2C+Count%3A+%7BCount%7D` (renders as: *Item: {Name}, Count: {Count}*)

- `localOnly` — Limits the selection to documents defined within the current document. References from other documents will be excluded from the drop-down list.

  Example: `localOnly=true`

- `pathFilter` — Filters available documents based on their path (RFC 6901 JSON Pointer). Use `*` to include all paths, or an empty string `""` to include only root-level documents.

  Example: `pathFilter=%2FItem` (documents under */Item*) `pathFilter=*` (all documents) `pathFilter=` (only root documents)

Example

```
{ "Id": "Sword" }
"Sword" // just raw Id is also accepted
```

Reference Collection

The `ReferenceCollection` data type is used to create non-embedded, one-to-many relationships between documents. It allows a document to reference multiple documents of the same type without embedding them directly.

When using a `ReferenceCollection`, the referenced documents are stored as references to their original locations, rather than being embedded in the parent document. This approach is particularly useful when dealing with large or complex datasets where referencing is more efficient and scalable than embedding. It also helps maintain data integrity by ensuring that all references point to valid documents.

For example, in a game, each **Quest** may reference a collection of **Objectives**. These objectives can be defined separately for reuse across multiple quests. The `ReferenceCollection` data type enables the quest document to maintain a list of references to objective documents.

Source Code Type

Language	Type
C#	ReadOnlyCollection{T}
TypeScript	ReadOnlyArray{T}
C++ (Unreal Engine)	TArray{T}

Haxe	ReadOnlyArray{T}

Uniqueness

Reference Collection cannot be marked as unique.

Size

May be limited in number of items. 0 - no limit.

Specification

References support the following specification parameter:

- `displayTextTemplate` — Defines a template string for how the referenced value is displayed in the UI.

  Example: `displayTextTemplate=Item%3A+%7BName%7D%2C+Count%3A+%7BCount%7D` (renders as: *Item: {Name}, Count: {Count}*)

- `localOnly` — Limits the selection to documents defined within the current document. References from other documents will be excluded from the drop-down list.

  Example: `localOnly=true`

- `pathFilter` — Filters available documents based on their path (RFC 6901 JSON Pointer). Use `*` to include all paths, or an empty string `""` to include only root-level documents.

  Example: `pathFilter=%2FItem` (documents under */Item*) `pathFilter=*` (all documents) `pathFilter=` (only root documents)

Example

```
[{ "Id": "Sword" }, { "Id": "Gold" }]
["Sword", "Gold"] // just raw Ids are also accepted
```

Text

The `Text` data type is used to store simple text values in game data. Unlike the LocalizedText data type, `Text` does not have support for multiple translations of the same text. Instead, it allows for the storage of any UTF symbol in a single language. This data type is useful for fields that do not require localization, such as character names, item descriptions, or game lore.

Source Code Type

Language	Type
C#	System.String
TypeScript	string
C++ (Unreal Engine)	FString
Haxe	String

Uniqueness

Text can be marked as unique (case sensitive for uniqueness purposes).

Size

May be limited in number of characters. 0 - no limit.

Example

```
"Hello world!"
```

Sub-Types

- Asset Path
- **`Asset Path Collection`**

- Vector 2/3/4
- Integer Vector 2/3/4
- Rectangle
- Integer Rectangle
- Tags

Time

The `Time` data type in game data is equivalent to the `TimeSpan` data type in C#. It is used to store a duration or a time interval, such as the time it takes to complete a task or the length of a cutscene in a game. The `Time` data type is represented as a string in the format `HH:mm:ss`, where HH is the number of hours, mm is the number of minutes, and ss is the number of seconds.

For example, if a task takes 2 hours and 30 minutes to complete, the `Time` data type value would be `02:30:00`.

Source Code Type

Language	Type
C#	System.TimeSpan
TypeScript	TimeSpan (Bundled)
C++ (Unreal Engine)	FTimespan
Haxe	TimeSpan (Bundled)

Uniqueness

Time cannot be marked as unique.

Format

`[DD.]HH:mm:ss` or `<number-of-seconds>`

Example

```
"02:30:00" // 2 hours and 30 minutes
"1.00:00:00" // 1 day
60 // 60 seconds
120 // two minutes

"-00:30:00" // could be negative
```

Asset Path

The `Asset Path` is a sub-type of the Text data type and is used to store a path to a game asset within game data.

The `Asset Path Collection` is a collection variant of this type and it is used to store multiple paths. Each path is separated by the | (pipe) character.

Paths are typically relative to the project root and can be used to load assets directly at runtime.

Specification

Asset Path Collection supports the following specification parameters:

- `assetType` — Filters the types of assets to select. For Unity and Unreal Engine, this refers to class names. For standalone builds, it refers to file extensions. Example: `assetType=Texture` or `assetType=Texture2D`

UI Behavior

This field is represented in the editor by a multi-select searchable input. Users can type to search for assets and add multiple items to the collection using auto-complete.

The search input supports the `t:<TypeName>` filter pattern to limit search results by asset type.

Examples:

- Unity: `swing t:AudioClip`
- Unreal Engine: `swing t:SoundWave`
- Standalone: `swing t:wav`

Validation Behavior

The Charon editor does not validate asset paths during game data loading (i.e., at runtime). You are responsible for ensuring the existence of assets at the provided paths. To improve reliability, consider writing a custom editor extension that validates all asset references before packaging the game.

Asset Localization

Localizable asset paths are not currently supported. If this is a required feature, you are encouraged to suggest it in Charon's Discord channel.

Path Origin

The meaning of "relative path" for asset references depends on how the game data editor is launched:

- **Unity Editor**: Paths are relative to the Unity project root directory, typically the directory containing the `Assets` folder. Asset paths will usually start with `Assets/`. Example: `Assets/Textures/MyTexture.png`
- **Unreal Engine Editor**: Paths are relative to the Unreal project root directory, usually containing the `Content` folder. Asset paths often start with `/Game/` because it is Unreal Engine package path, not physical path. Example: `/Game/Textures/MyTexture`
- **Standalone Execution** (via CLI): Paths are relative to the game data file path by default. This behavior can be customized using either:
  - The `--gameAssetsPath` CLI argument: `dotnet tool charon ./gamedata.gdjs --gameAssetsPath "C:/Projects/MyGame/"`
  - The `STANDALONE__GAMEASSETSPATH` environment variable.

Example

```
"Assets/Textures/MyTexture.png"     // Unity
"/Game/Textures/MyTexture"          // Unreal Engine
"Textures/MyTexture.png"            // Standalone
```

Rectangle

The `Rectangle` is a sub-type of the Text data type used to store four decimal values representing a rectangle. The components represent `X`, `Y`, `Width`, and `Height`, in that order. Values are separated by spaces and must be parsed from the string before use at runtime.

UI Behavior

This field is represented in the editor as a group of input fields labeled `X`, `Y`, `Width`, and `Height`. Each component accepts decimal numbers.

Validation Behavior

The editor ensures that a valid rectangle value is entered and only stores properly formatted values. However, there is no runtime validation or built-in parsing logic, so values must be parsed manually in game code.

Example

```
"50.0 100.0 200.5 25.0"
```

Parsing

Unity (C#) example:

```
var uiElement = gameData.AllUIElements.Get("HealthBar"); // -> UIElement
var boundsString = uiElement.Bounds; // -> "50.0 100.0 200.5 25.0"
var components = boundsString.Split(' ');
var bounds = new Rect(
    float.Parse(components[0]), // X
    float.Parse(components[1]), // Y
    float.Parse(components[2]), // Width
    float.Parse(components[3])  // Height
);
```

Unreal Engine (C++) example:

```
auto UIElement = GameData->AllUIElements->Find(TEXT("HealthBar")); // -> UUIElement*
FString BoundsString = UIElement->Bounds; // -> "50.0 100.0 200.5 25.0"
TArray<FString> Components;
BoundsString.ParseIntoArray(Components, TEXT(" "));
FSlateRect Bounds(
    FCString::Atof(*Components[0]), // X
    FCString::Atof(*Components[1]), // Y
    FCString::Atof(*Components[2]), // Width
    FCString::Atof(*Components[3])  // Height
);
```

TypeScript example:

```
let uiElement = gameData.uiElementsAll.find("HealthBar"); // -> UIElement
let boundsString = uiElement.Bounds; // -> "50.0 100.0 200.5 25.0"
let parts = boundsString.split(" ").map(x => parseFloat(x));
let bounds = { x: parts[0], y: parts[1], width: parts[2], height: parts[3] };
```

Haxe example:

```
var uiElement = gameData.uiElementsAll.get("HealthBar"); // -> UIElement
var boundsString = uiElement.Bounds; // -> "50.0 100.0 200.5 25.0"
var parts = boundsString.split(" ").map(Std.parseFloat);
var bounds = { x: parts[0], y: parts[1], width: parts[2], height: parts[3] };
```

Integer Rectangle

The `Integer Rectangle` (`RectangleInt`) is a sub-type of the Text data type used to store four integer values representing a rectangle. The components represent `X`, `Y`, `Width`, and `Height`, in that order. Values are separated by spaces and must be parsed from the string before use at runtime.

UI Behavior

This field is represented in the editor as a group of input fields labeled `X`, `Y`, `Width`, and `Height`. Each component accepts only whole numbers (integers).

Validation Behavior

The editor ensures that a valid integer rectangle is entered and only stores properly formatted values. However, there is no runtime validation or built-in parsing logic, so values must be parsed manually in game code.

Example

```
"50 100 200 25"
```

Parsing

Unity (C#) example:

```
var uiElement = gameData.AllUIElements.Get("HealthBar"); // -> UIElement
var boundsString = uiElement.Bounds; // -> "50 100 200 25"
var components = boundsString.Split(' ');
var bounds = new RectInt(
    int.Parse(components[0]), // X
    int.Parse(components[1]), // Y
    int.Parse(components[2]), // Width
    int.Parse(components[3])  // Height
);
```

Unreal Engine (C++) example:

```
auto UIElement = GameData->AllUIElements->Find(TEXT("HealthBar")); // -> UUIElement*
FString BoundsString = UIElement->Bounds; // -> "50 100 200 25"
TArray<FString> Components;
BoundsString.ParseIntoArray(Components, TEXT(" "));
FIntRect Bounds(
    FCString::Atoi(*Components[0]), // X
    FCString::Atoi(*Components[1]), // Y
    FCString::Atoi(*Components[2]), // Width
    FCString::Atoi(*Components[3])  // Height
);
```

TypeScript example:

```
let uiElement = gameData.uiElementsAll.find("HealthBar"); // -> UIElement
let boundsString = uiElement.Bounds; // -> "50 100 200 25"
let parts = boundsString.split(" ").map(x => parseInt(x, 10));
let bounds = { x: parts[0], y: parts[1], width: parts[2], height: parts[3] };
```

Haxe example:

```
var uiElement = gameData.uiElementsAll.get("HealthBar"); // -> UIElement
var boundsString = uiElement.Bounds; // -> "50 100 200 25"
var parts = boundsString.split(" ").map(Std.parseInt);
var bounds = { x: parts[0], y: parts[1], width: parts[2], height: parts[3] };
```

Tags

The `Tag` data type is a sub-type of the Text data type and is used to store a reusable set of tag values (e.g., “fire”, “enemy”, “magic”) that can be shared across multiple documents.

The `Tag Collection` is a collection variant of this type and it is used to store multiple tags. Each tag is separated by the `` ` ` `` (space) character.

UI Behavior

In the editor, this field is represented as a list of chips/pills/tokens with support for free-form input and autocompletion of existing tags. Note that autocompletion relies on scanning all documents of the same type to extract available tags, so performance considerations should be made when using this data type at scale.

Validation Behavior

The Charon editor does not validate tags during game data loading (i.e., at runtime). Ensure that your game logic handles any missing or unexpected tags appropriately.

Relation to Multi-Pick List

Tags are similar to the Multi-Pick List data type but offer more flexibility by allowing free-form input. C-style named tags can be easily converted to a Multi-Pick List by changing the property type. Likewise, Multi-Pick List values can be converted back to Tags if strict value control is no longer needed.

Example

```
"fire"
"poison magical non-resistable"
```

Vector 2/3/4

The `Vector 2/3/4` is a sub-type of the Text data type used to store a group of 2, 3, or 4 decimal values. These values are space-separated and must be parsed from the string before use at runtime.

UI Behavior

This field is represented in the editor as a group of input fields, each allowing the user to enter one component of the vector (e.g., X, Y, Z).

Validation Behavior

The editor ensures that a valid vector is entered and only stores properly formatted values. However, there is no runtime validation or built-in parsing logic, so the value must be manually parsed in game code.

Example

```
"120.1 56.3 30.0"
```

Parsing

Unity (C#) example:

```
var hero = gameData.AllHeroes.Get("Knight"); // -> Hero
var startingLocationString = hero.StartingLocation; // -> "120.1 56.3 30.0"
var components = startingLocationString.Split(' ');
var startingLocation = new Vector3(
    float.Parse(components[0]),
    float.Parse(components[1]),
    float.Parse(components[2])
);
```

Unreal Engine (C++) example:

```
auto Hero = GameData->AllHeroes->Find(TEXT("Knight")); // -> UHero*
FString StartingLocationString = Hero->StartingLocation; // -> "120.1 56.3 30.0"
TArray<FString> Components;
StartingLocationString.ParseIntoArray(Components, TEXT(" "));
FVector StartingLocation = FVector(
    FCString::Atof(*Components[0]),
    FCString::Atof(*Components[1]),
    FCString::Atof(*Components[2])
);
```

TypeScript example:

```
let hero = gameData.heroesAll.find("Knight"); // -> Hero
let startingLocationString = hero.StartingLocation; // -> "120.1 56.3 30.0"
let parts = startingLocationString.split(" ").map(parseFloat);
let startingLocation: [number, number, number] = [parts[0], parts[1], parts[2]];
```

Haxe example:

```
var heroById = gameData.heroesAll.get("Knight"); // -> Hero
var startingLocationString = heroById.StartingLocation; // -> "120.1 56.3 30.0"
var parts = startingLocationString.split(" ").map(Std.parseFloat);
var startingLocation = { x: parts[0], y: parts[1], z: parts[2] };
```

Integer Vector 2/3/4

The `Integer Vector 2/3/4` is a sub-type of the Text data type used to store a group of 2, 3, or 4 integer (whole number) values. Values are separated by spaces and must be parsed from the string before use at runtime.

UI Behavior

This field is represented in the editor as a group of input fields, allowing the user to enter each component of the vector (e.g., X, Y, Z) as whole numbers.

Validation Behavior

The editor ensures that a valid integer vector is entered and only stores properly formatted values. However, there is no runtime validation or built-in parsing logic, so the value must be manually parsed in game code.

Example

`"10 20 5"`

Parsing

Unity (C#) example:

```
var hero = gameData.AllHeroes.Get("Knight"); // -> Hero
var gridPositionString = hero.StartingLocation; // -> "10 20 5"
var components = gridPositionString.Split(' ');
var startingLocation = new Vector3Int(
    int.Parse(components[0]),
    int.Parse(components[1]),
    int.Parse(components[2])
);
```

Unreal Engine (C++) example:

```
auto Hero = GameData->AllHeroes->Find(TEXT("Knight")); // -> UHero*
FString GridPositionString = Hero->StartingLocation; // -> "10 20 5"
TArray<FString> Components;
GridPositionString.ParseIntoArray(Components, TEXT(" "));
FIntVector StartingLocation = FIntVector(
    FCString::Atoi(*Components[0]),
    FCString::Atoi(*Components[1]),
    FCString::Atoi(*Components[2])
);
```

TypeScript example:

```
let hero = gameData.heroesAll.find("Knight"); // -> Hero
let gridPositionString = hero.StartingLocation; // -> "10 20 5"
let parts = gridPositionString.split(" ").map(x => parseInt(x, 10));
let startingLocation: [number, number, number] = [parts[0], parts[1], parts[2]];
```

Haxe example:

```
var heroById = gameData.heroesAll.get("Knight"); // -> Hero
var gridPositionString = heroById.StartingLocation; // -> "10 20 5"
var parts = gridPositionString.split(" ").map(Std.parseInt);
var startingLocation = { x: parts[0], y: parts[1], z: parts[2] };
```

Choosing the appropriate data type is essential for ensuring that game data is accurately structured, efficiently stored, and correctly interpreted by both the editor and runtime systems. Each data type serves a specific purpose and is designed to represent particular kinds of values or relationships.

For instance, use the **`Text`** data type for simple string values, or the **`LocalizedText`** type when the same text must support multiple languages. For numeric data, **`Number`** is suited for decimal values, while **`Integer`** should be used for whole numbers.

When a property should be selected from a defined set of options, **`PickList`** and **`MultiPickList`** are recommended for single or multiple selections, respectively. For more complex structures, use **`Document`** to define an embedded object, or **`DocumentCollection`** to represent a list of such objects.

By selecting data types according to their intended purpose, developers can ensure better validation, clearer data organization, and more predictable behavior across tools and runtime systems.

Data Types

Data Type	Description	Example
Text	A line of text.	“Hello, world!”
LocalizedText	A localized text for multiple languages.	{“en-US”: “Hello”, “fr-FR”: “Bonjour”}
Logical	A true/false value.	true
Time	A time duration (days, hours, etc.).	“1.00:00:00”
Date	A calendar date and time.	“2017-12-27T00:00:00.000Z”
Number	A decimal number.	3.14
Integer	A whole number.	42
PickList	A single value from a predefined list.	“Red”
MultiPickList	Multiple values from a predefined list.	“Apple, Banana, Cherry”
Document	An embedded structured object.	{ “Id”: “Sword”, “Name”: “Rusty Sword” }
DocumentCollection	A list of embedded structured objects.	[{ “Id”: “Sword”, “Name”: “Rusty Sword” }]
Reference	A pointer to another document by ID.	{ “Id”: “Sword” }
ReferenceCollection	A list of pointers to other documents.	[{ “Id”: “Sword” }]
Formula	A C# expression evaluated at runtime.	“target.HP < 100”

Composite Types

Composite Type	Base Type	Description
Asset Path	Text	A relative path to a single game asset.
Vector 2/3/4	Text	A group of 2, 3, or 4 decimal values (e.g., position or scale).
Integer Vector 2/3/4	Text	A group of 2, 3, or 4 whole numbers.
Rectangle	Text	A rectangle with floating-point position and size.
Integer Rectangle	Text	A rectangle with integer position and size.
Tags	Text	A reusable set of tags shared across documents.

Use Cases and Guidance

- Use **Text** and **LocalizedText** for titles, descriptions, and UI content.
- Choose **Integer** or **Number** for quantities, rates, and stats.
- Use **Document** or **Reference** to include structured data or link to another entity.
- Use **PickList** or **MultiPickList** when the value should be selected from a known set.
- Use **Asset Path** or **Asset Path Collection** when referencing assets like textures, sounds, or prefabs by path.

- Use **Formula** to define runtime-calculated values, like damage formulas or dynamic prices.
- Use **Tags** when multiple documents share common descriptors (e.g., “fire”, “enemy”, “magic”).

Display Text Template

When a document needs to be shown or referenced in the UI, a text representation is required. By default, Charon will use one of the following properties, in order of precedence: `DisplayName`, `Name`, `Title`, or `Id`. This fallback behavior can often result in uninformative or unclear document labels.

For example, consider the following document:

```
{
  "Id": 1,
  "Description": "Bring 40 wolf skins",
  "Type": "RepeatableQuest",
  "Tasks": [ /* ... */ ]
}
```

By default, this document would be displayed in UI lists simply as `"1"`. This can be improved by defining a custom **Display Name Template** for the Schema.

For instance, the template:

```
Quest #{Id}: {Type} - {Description}
```

would render the document’s label as:

```
Quest #1: RepeatableQuest - Bring 40 wolf skins
```

Template Syntax

The display name template is a string that may include *replaceable expressions* enclosed in curly braces (`{}`). These expressions support a subset of C-style syntax and can reference any top-level or nested property of the document.

Property Access

- Top-level properties can be accessed directly by name.
- Nested properties in embedded documents can be accessed using dot notation, e.g., `Item.Name`.

Supported Data Types

Type	Example
String	`"a string"`
Number	`1` or `3.14`
Boolean	`true` or `true`
null	`null`

Unary Operators

Operator	Example
Logical NOT	`!CanWalk`
Bitwise Complement	`~ResistanceTypes`
Negation	`-Damage`

Binary Operators

The following binary expressions are supported:

Operator	Example
Subtract	`TotalPrice - BasePrice`
Add	`BaseDamage + BonusDamage`
Concatenate	`"Quest #" + Id`
Division	`CriticalChance / CurseDebuff`
Multiplication	`DropRate * 100.0`
Power	`Damage ** CriticalChance`
Modulo	`Slots % SlotPerPage`
Bitwise AND	`DamageFlags & AdditionalDamageFlags`
Bitwise OR	`DamageTypes | ResistanceTypes`
Bitwise XOR	`ResistanceTypes ^ HealingTypes`
Bitwise Left Shift	`Flags << 1`
Bitwise Right Shift	`Flags >> 1`
Logical AND (short-circuit)	`CanWalk && CanRun`
Logical OR (short-circuit)	`HasHead || CanBeHeadshoted`
Greater Than	`Count > MaxCount`
Greater Than or Equal	`Count >= MaxCount`
Less Than	`HP < MaxHP`
Less Than or Equal	`HP <= MaxHP`
Equal To	`Type == "RepeatableQuest"`
Not Equal To	`Description != null`
Coalesce	`Nickname ?? Name`

Ternary Operator

Ternary expressions are supported using the standard syntax:

```
{Damage >= 0 ? "Total Damage" : "Total Heal"}: {Damage}
```

This will result in either “Total Damage: X” or “Total Heal: X”, depending on the value of `Damage`.

Operator Precedence

You can control expression evaluation order using parentheses. For example:

```
Max Damage: {BaseDamage * (BaseCriticalChance + BonusCriticalChance)}
```

Boolean Coercion

Many types can be implicitly coerced to a Boolean value:

- **Numeric values** are considered `true` if they are not equal to 0.
- **Documents**, **References**, and **Formulas** are considered `true` if they are not null.
- **Collections** (Document or Reference), **Text**, and **Localized Text** are `true` if they are not empty.

Other types must be explicitly compared to a value to be evaluated as Boolean. These coercion rules apply to ternary expressions and unary logical negation (*!*).

```
Has Items: {Items ? "Yes" : "No"}
```

Format Specifiers

Template expressions can include format specifiers, similar to C#'s formatting syntax. Add a colon and format string at the end of the substitution expression:

```
Resist Chance: {ResistChance * 100.0 :F2}
```

This limits the precision to two decimal places.

Supported format specifiers:

- `F`: Fixed-point (decimal precision), e.g., `F2` for two decimal places.
- `X` / `x`: Uppercase / lowercase hexadecimal (integers only), e.g., `X16` for hexadecimal string padded with zeroes to length of 16 characters.
- `B`: Binary representation (integers only), e.g., `B32` for binary string padded with zeroes to length of 32 characters.
- `L`: String length limitation, e.g., `L12` for a string shorter than 12 characters nothing will change, but if it is longer than 12 it will be truncated to 12 and an ellipsis character … will be added.

Collection Accessors

Collections can be accessed by index using square brackets. If the index is out of bounds, the result is `null`.

```
First Item: {Items[0]}
```

The number of items in a collection can be accessed using the `.Count` property:

```
Item Count: {Items.Count}
```

To create a joined string from all items in a collection, use the `Join(separator: string)` method:

```
Items: {Items.Join(" and ")}
```

This would produce output like:

```
Items: Sword and Shield and Armor
```

String Manipulation Methods

String expressions support several methods:

- `Trim()` — Removes leading and trailing whitespace.
- `Substring(start: int, count: int)` — Extracts a substring from a given position.
- `ToString()` — Converts any non-null value to its string representation.

String concatenation is done using the + operator:

```
Damage: {Damage + " [" + DamageType + "]"}
```

Result:

```
Damage: 100 [Fire]
```

See also

- Schema
- Property
- Id Property
- Shared Property
- All Data Types
- Creating Document Type (Schema)

Schema

A **Schema** defines the structure of a specific type of game data entity. It acts as a blueprint that outlines the fields (properties) associated with the entity, much like columns in a database table or cells in a spreadsheet. Schemas are central to organizing and modeling structured game data.

Name

The **Name** of a schema is used as an alternative identifier to the *Id* and is also used in generated source code as the name of the resulting class. Therefore, it must be a valid C-style identifier: it should start with a letter and include only Latin letters, digits, or underscores.

Schema names are also used as the name of the collection when storing or exporting data. It is recommended to use PascalCase and singular form for schema names.

Display Name

The **Display Name** is used in the UI and can contain any characters. There are no restrictions on alphabet or character set.

Type

The **Type** setting defines how instances (documents) of this schema are created and managed:

- **Normal**: Standard behavior. Documents can be created in the root collection, embedded in other documents, or may be absent entirely.
- **Component**: These documents can only be embedded within other documents. They do not appear in the main navigation menu.
- **Settings**: A singleton-style schema. Only one root-level document can exist, and it cannot be embedded in other documents.

Display Text Template

The Display Text Template defines how documents of this schema are presented in the UI when a textual representation is needed—e.g., in dropdowns, reference lists, or relationship views.

Icon

The **Icon** is displayed next to the schema name in the navigation menu or selection dialogs within the UI.

Description

The **Description** helps other users understand the purpose of the schema. It is shown in the UI and is also included as a documentation comment in the generated source code for the corresponding class.

Group

The **Group** defines the hierarchical path of the schema in the left-hand navigation menu. It behaves similarly to a folder path, with sections separated by the / character.

Menu Visibility

This flag determines whether the schema appears in the left-hand navigation menu in the UI.

Id Generator

The **Id Generator** setting defines how the Id Property is managed for this schema. If set to *None*, the Id property must be configured manually.

Properties

A list of **`Properties`** that define the fields associated with this schema.

See also

- Display Text Template
- Property
- Id Property
- Shared Property
- Specification
- All Data Types
- Creating Document Type (Schema)

Specification

The `Specification` field is a flexible extension point available on both Schema and Property documents. It enables developers to attach additional metadata in a standardized format that can be used for custom logic in UI field editors, source code generation, or other tooling integrations.

Overview

The `Specification` field stores a string formatted using the application/x-www-form-urlencoded standard (commonly referred to as URLSearchParams format). It contains key-value pairs separated by ampersands (*&*), where each key and value are URL-encoded.

For example:

```
icon=table&group=Combat
```

Use Cases

The *Specification* field is commonly used for:

- **Schema metadata**: Define an icon, group behavior, or UI preferences:

  ```
  icon=table&category=Gameplay&hideInMenu=true
  ```

- **Property editor extensions**: Pass configuration values to custom editors:

  ```
  colorFormat=RGB&alpha=true
  ```

- **Source code generation**: Customize how types or methods are generated:

  ```
  typeName=MyEnum&csAttribute=Obsolete
  ```

Encoding Guide

All keys and values must be properly URL-encoded to ensure compatibility with parsing libraries and avoid format-breaking characters.

Encoding in JavaScript

In any browser console or Node.js REPL:

```
const params = new URLSearchParams({
  icon: "sword",
  group: "Items and Stuff",
  noThumbnail: true
});
console.log(params.toString());
// Output: icon=sword&group=Items%20and%20Stuff&noThumbnail=true
```

Encoding in Excel

To encode values manually using Excel formulas:

- Build a key-value pair:

  ```
  =ENCODEURL("noThumbnail") & "=" & ENCODEURL("true")
  ```

- Combine multiple:

  ```
  =ENCODEURL("icon") & "=" & ENCODEURL("sword") & "&" & ENCODEURL("group") & "=" & ENCODEU
  ```

Best Practices

- **Use lowercase or camelCase for keys** to ensure consistency.
- **Avoid spaces or special characters** in keys; values should always be encoded.

See also

- Display Text Template
- Property
- Schema
- All Data Types
- Creating Document Type (Schema)

Id Property

Each schema includes an `Id` property that defines the unique identifier for a document. This property is always named `Id` and must satisfy the following constraints:

- Must be unique.
- Required and must be non-empty (for `Text` data type).
- Must use a data type with a stable text representation and comparison support, such as:
  - Time
  - Date
  - Number
  - Integer
  - Text
  - MultiPickList
  - PickList

Generated Id

Schemas can specify that the `Id` property should be automatically generated. Based on the selected `Id Generator` type, an appropriate `Id` property will be added to the schema.

Supported Id Generators:

- ObjectId
- Guid
- Sequence
- GlobalSequence

Generated ObjectId

A BSON ObjectId is a 96-bit unique identifier derived from the machine name, process ID, and timestamp. It is always increasing and well-suited for sorting documents by creation time.

Id property details:

- **Data Type:** Text
- **Size:** 24 characters
- **Requirement:** Not Empty
- **Example:** “049bc0604c363a980b000088”

Generated Guid

A GUID (Globally Unique Identifier) is a 128-bit identifier typically represented as a 32-character hexadecimal string. It ensures global uniqueness across systems.

Id property details:

- **Data Type:** Text
- **Size:** 32 characters
- **Requirement:** Not Empty
- **Example:** “6fe4202b1b9c4565b439980138524112”

Generated Sequence

A sequential numeric identifier unique to each document within a specific schema. Documents from different schemas may share the same numeric `Id`. For globally unique numeric IDs, use `GlobalSequence`.

Id property details:

- **Data Type:** Integer
- **Size:** 32-bit
- **Requirement:** Not Null
- **Example:** 1

Generated Global Sequence

A globally unique numeric identifier across all schemas and documents in the project.

Id property details:

- **Data Type:** Integer
- **Size:** 32-bit
- **Requirement:** Not Null
- **Example:** 69

Non-Generated Id

Schemas may also use a manually defined `Id` property by selecting the `None` option for the `Id Generator`. You can define the `Id` property using any supported data type listed above.

All Id constraints will be automatically validated when saving the schema. You can change a schema’s `Id Generator` to `None` at any time, but you cannot switch back to a generator once this change is made.

Id Placeholder

If you want to assign a temporary value to the ID of a document being created—for example, for internal reference—you can use a placeholder string such as `_ID_XXX_ANY_UNIQUE_TEXT_HERE_XXX`. This placeholder will be automatically replaced with the actual generated ID during saving or importing.

Before save:

```
{
  "Id": "_ID_ITEM_123",
  "Name": "My New Item",
  "ItemReference": {
```

```
    "Id": "_ID_ITEM_123",  // self-reference
    "DisplayName": "My New Item"
  }
}
```

After save:

```
{
  "Id": 42,
  "Name": "My New Item",
  "ItemReference": {
    "Id": 42,
    "DisplayName": "My New Item"
  }
}
```

See also

- Schema
- Property
- Shared Property
- All Data Types
- Creating Document Type (Schema)

Schema Property

Properties define which fields a document will contain.

Name

The **Name** of a property is used as an alternative identifier to the *Id* and is also used in generated source code as the name of the resulting field/property. Therefore, it must be a valid C-style identifier: it should start with a letter and include only Latin letters, digits, or underscores.

Property names are also used as the name of the fields when storing or exporting data. It is recommended to use PascalCase and singular form for property names.

Display Name

The **Display Name** is used in the UI and can contain any characters. There are no restrictions on alphabet or character set.

Description

The **Description** helps other users understand the purpose of the property. It is shown in the UI and is also included as a documentation comment in the generated source code for the corresponding field/property.

Sync Name

Some properties describe data types that are common across multiple schemas. For example, a `PickList` property might benefit from shared pick options used in several schemas. These synced properties are referred to as Shared Properties.

Data Type

Each property has a data type, which determines what kind of data can be entered into the document's field, how it is stored on disk, and how it is represented at runtime. The editor UI will attempt to validate input based on the selected data type but does not strictly enforce it. At runtime, game data loading code will validate the data types and may fail if the document is malformed.

Requirement

Specifies whether a value must be provided for the field. Possible values include:

- **None** – No requirement to fill in the property. It can be absent from the document both on disk and at runtime. Typically represented as a nullable or optional type in supported source code languages.
- **Not Null** – A value must be provided. For the `Text` data type, an empty string is considered a valid value.
- **Not Empty** – The field must contain a non-empty value for `Text` or have at least one item for `Collection` data types.

Uniqueness

Specifies uniqueness constraints for field values across documents. Useful to ensure certain values, such as names, are only used once.

- **Unique** – The value must be unique across all documents.
- **Unique In Collection** – The value must be unique within the scope of the current collection.

Default Value

Defines the default value to use for newly created documents or to populate existing documents when a new required property is added. The default must be a properly formatted value consistent with the specified data type.

Id Property

Each document includes a required Id property, which has specific constraints and must be present and valid.

See also

- Schema
- Id Property
- Shared Property
- All Data Types
- Creating Document Type (Schema)

Shared Property

A **Shared Property** is a Schema property that syncs data type-related configuration with other properties of the same name across different schemas. When a shared property is modified in one schema, all other synced properties with the same shared name are updated accordingly.

Shared properties can be unlinked (or “decoupled”) at any time, converting them into independent, schema-specific properties.

Replicated Parameters

Shared properties replicate a defined subset of configuration parameters between themselves:

- Data Type
- Default Value
- Uniqueness
- Requirement
- Reference Type (for *Document*, *DocumentCollection*, *Reference*, and *ReferenceCollection* types)
- Size
- Specifications for *PickList*, *MultiPickList*, and *Formula* data types

These parameters ensure consistency across schemas that use the same shared property.

Impact on Generated Source Code

In cases where a property results in a distinct type in the generated source code—such as a *PickList* producing an `enum`—the name of the shared property is used to derive the generated type name.

For example:

- A shared *PickList* property named `Damage Type` would generate:
  - *DamageType* enum in C#, Haxe, TypeScript
  - *EDamageType* enum in Unreal Engine C++

To customize this behavior, the *Generated Type Name* field of the property can be set manually to override the default naming convention.

See also

- Schema
- Property
- Id Property
- Shared Property
- All Data Types
- Creating Document Type (Schema)

See also

- Schema
- Property
- Implementing Inheritance
- Filling Documents
- Publishing Game Data
- Generating Source Code

Filling Documents

Once the game data structure has been defined, there are several methods available for creating and populating game entities. One option is to import game data from other sources, such as tables or JSON files. Another option is to generate data using external tools and import it into the editor. Finally, data can be added gradually as development progresses using the game data editor.

Importing JSON files

JSON files can be imported via the user interface by following these steps:

1. Navigate to the document collection page.
2. Click on `Actions` → `Import...`.
3. Select the JSON file and follow the steps in the import wizard.

See structure requirements.

Exporting to Spreadsheet and Importing Back

To export game data to a spreadsheet for editing and then import it back, follow these steps:

1. Navigate to the document collection page.
2. Click on `Actions` → `Export To` → `Spreadsheet (.xlsx)` to export the data to a spreadsheet file.

3. Open the downloaded file and make the necessary edits.
4. **Import the modified data back into the system:**
    a. Drag and drop the edited file onto the document collection page.
    b. Alternatively, click on `Actions` → `Import...` and follow the steps in the import wizard to select and import the modified file.

Adding New Document

To create a new document using the user interface, follow these steps:

1. Navigate to the document collection page.
2. Click on the `Create` button.
3. Fill in the required fields in the form provided.
4. Click `Save` to save the new document.

See also

- Publishing Game Data
- Generating Source Code

Generating Source Code

The process of generating source code allows game data to be used inside a game. This process involves specifying the language (e.g. C#) and various generation parameters/optimizations. It can be done from both the project's dashboard user interface and the command-line interface (CLI).

Features

Feature	C#	TypeScript	C++ (UE)	Haxe
JSON Format	x	x	x	x
MessagePack Format	x	x	x	x
Language Switch	x	x	x	x
Patching	x	x	x	x
Formulas	x	x		
By Unique Value Indexing	x			x

Using Project's Dashboard UI

To generate source code from the dashboard, follow these steps:

1. Go to the dashboard of the project where you want to generate source code.
2. Click on the *Generate Source Code* button.
3. Choose the language you want to generate the source code in.
4. Specify any generation parameters required.
5. Click on the *Generate* button to initiate the process.
6. Download archive file with generated source code.

Using Command-Line Interface (CLI)

To generate source code from the CLI, follow these steps:

1. Open the command-line interface.
2. Navigate to the game data's directory.
3. Use the `GENERATE <SOURCECODE>` command to generate the source code, specifying the target language and any generation parameters require.

Example

```
charon GENERATE CSHARPCODE --dataBase "c:\my app\gamedata.json" --namespace "MyGame.Paramete
```

- Use the `--outputDirectory` parameter to specify the location where generated files will be saved.
- Use the `--namespace` and `--gameDataClassName` parameters to adjust the signature of generated classes.
- Use the `--splitFiles` parameter to generate multiple files instead of one large one.
- Use the `--clearOutputDirectory` parameter to clear the output directory from generated files when re-generating source code.

Once the process is complete, the generated source code will be available at `--outputDirectory`.

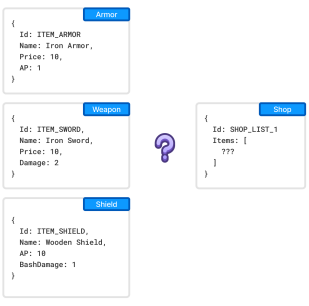
See also

- Publishing Game Data
- Working with Source Code (C# 4.0)
- Working with Source Code (C# 7.3)
- Working with Source Code (TypeScript)
- Working with Source Code (UE C++)
- **Working with Source Code (Haxe)**
- Command Line Interface (CLI)
- GENERATE CSHARPCODE Command
- GENERATE TYPESCRIPTCODE Command
- GENERATE UECPP Command
- GENERATE HAXE Command

Implementing Inheritance

Inheritance is a familiar tool for programmers when working with shared behavior or data. Unfortunately, it is not natively supported in Charon. However, you can achieve similar functionality using alternative approaches.

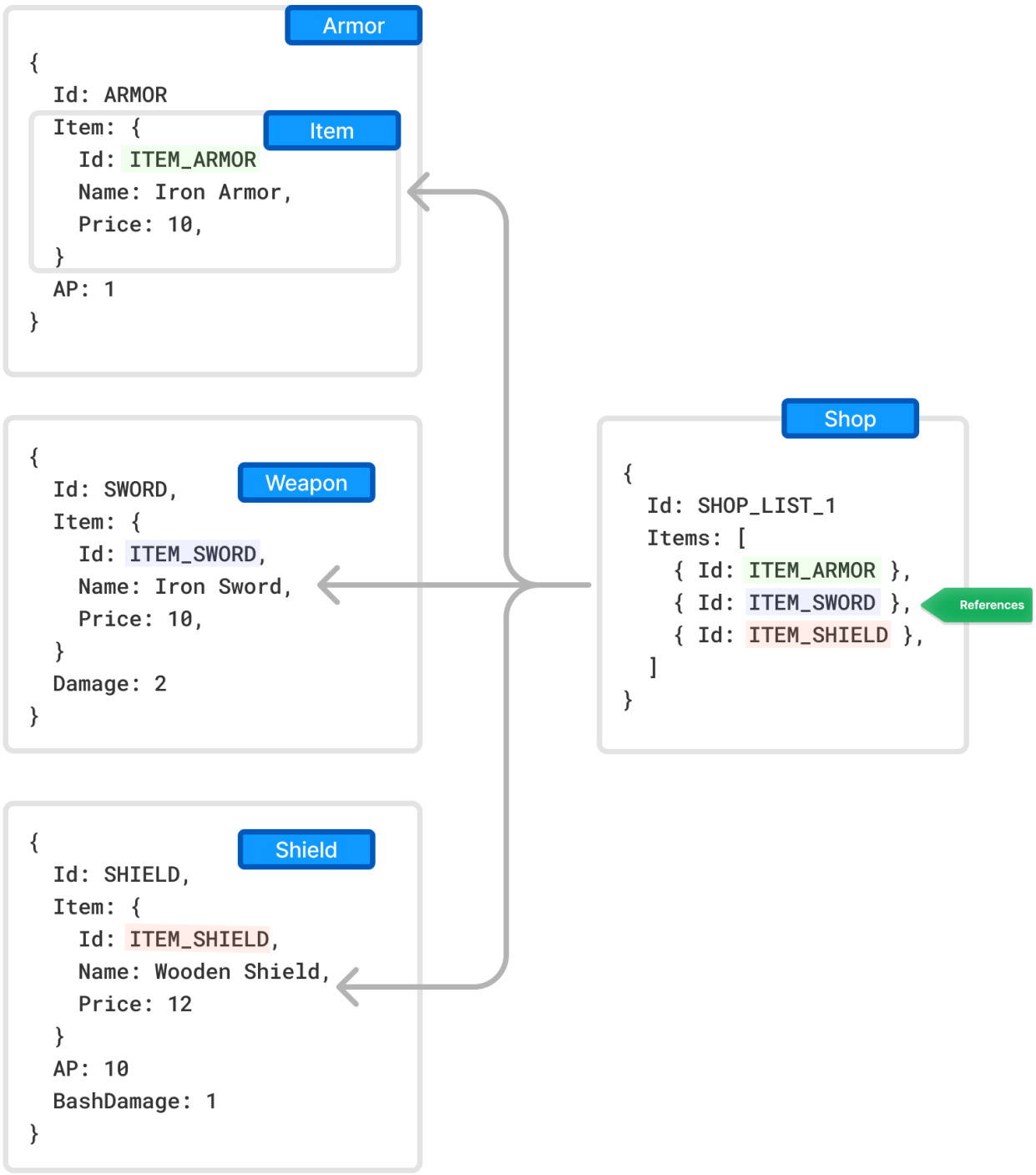
For example, imagine you have three different document types: `Armor`, `Weapon`, and `Shield`. You want to include all these items in a `Shop` list of sellable goods. In traditional inheritance, you could create a base type to unify these documents and refer to it.



Without inheritance, here are three possible approaches:

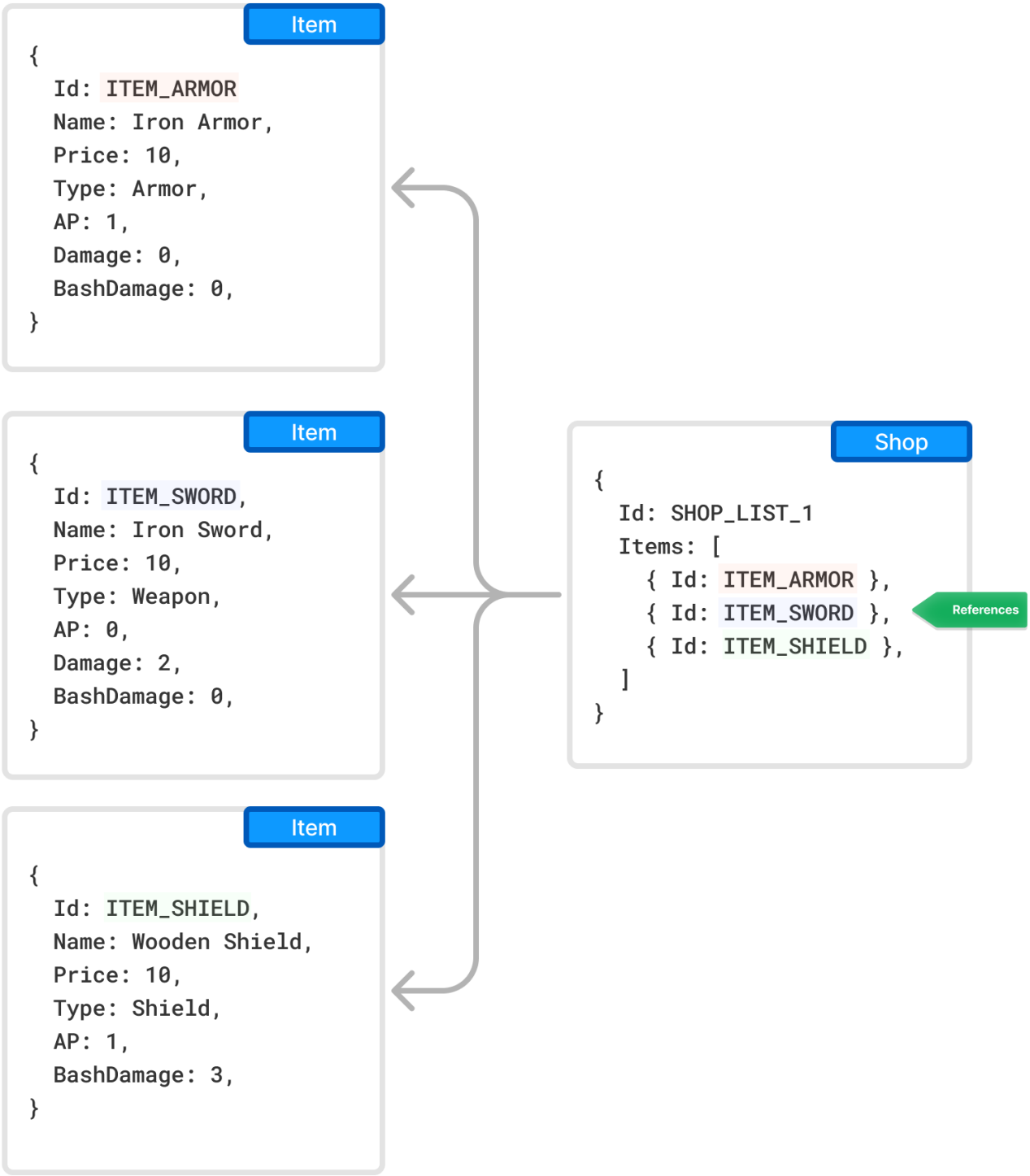
1. Composition

Extract the shared data from all sellable item types into a separate document type called `Item`. Each sellable document (e.g., `Armor`, `Weapon`, `Shield`) should include an embedded `Item` document containing store-relevant information. In the `Shop` list of sellable goods, you can store references to `Item` documents.



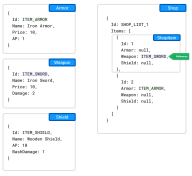
2. Merging

Alternatively, combine the three document types (`Armor`, `Weapon`, and `Shield`) into a single document type called `Item`. This document will contain fields for all three original types, along with an additional `Type` field to specify the item's category. The `Shop` list can then store references to these unified `Item` documents.



3. Aggregation

As a less elegant alternative, introduce a `ShopItem` type with fields referencing all possible document types (`Armor`, `Weapon`, and `Shield`). In each `ShopItem` document, only one of these fields will be filled, depending on the item's type. The `Shop` list can then reference `ShopItem` documents.



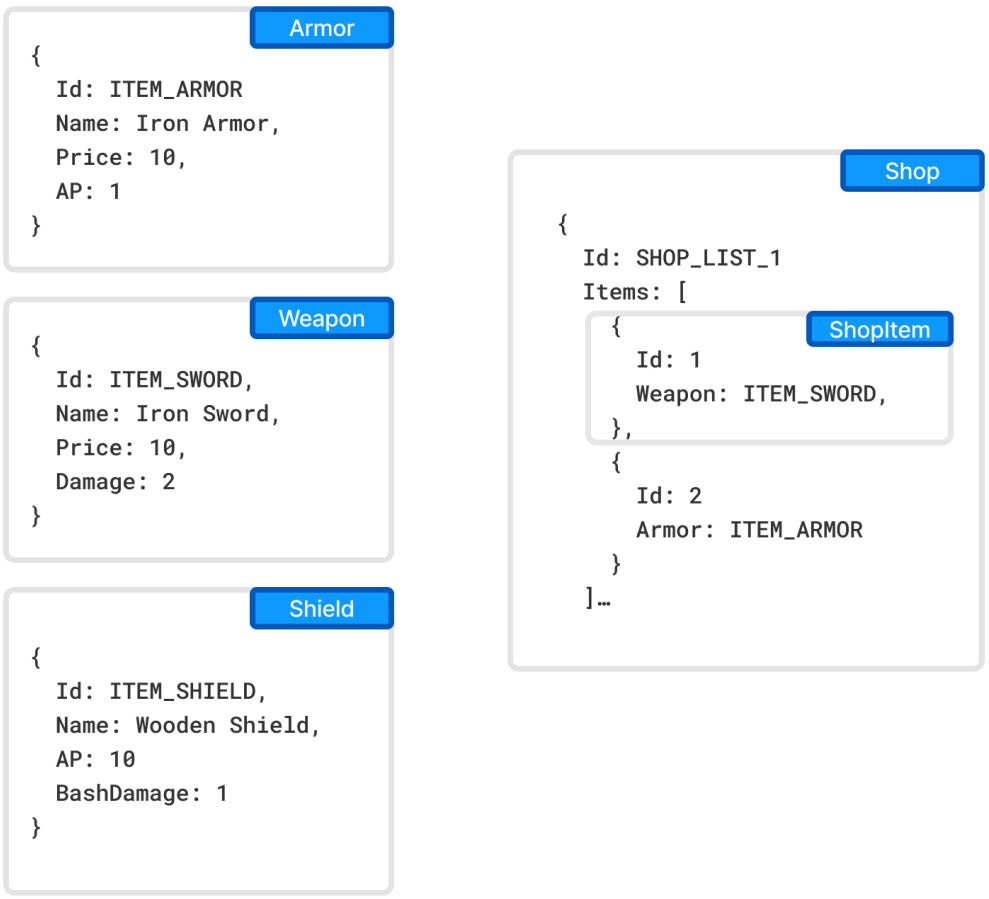
4. Tagged Union (Recommended)

A more powerful and storage-efficient alternative to Aggregation is to use a Tagged Union Schema (also known as a Discriminated Union, Sum Type, or `$oneOf`). Tagged Unions allow documents of different shapes to coexist within the same collection or be embedded inside other documents, making them a natural fit when multiple heterogeneous document types must be handled together.

Union Schemas have several important distinctions from Aggregation:

- They provide first-class support in the editor UI, making them more convenient to create and maintain than other inheritance alternatives.
- Unassigned properties within a Tagged Union document are not stored and therefore take no space.

When used in this scenario, a `ShopItem` can be modeled as a Tagged Union that defines `Armor`, `Weapon`, and `Shield` as its possible variants.



Conclusion

Each of these methods has its trade-offs in terms of simplicity, flexibility, and performance. Choosing the right approach depends on your application's requirements and the expected complexity of your data model.

See also

- Creating Document Type (Schema)
- Filling Documents
- Publishing Game Data
- Generating Source Code

Publishing Game Data

The publication process is a crucial step in preparing game data for usage inside the game. This process involves removing unused data, unused localization, and exporting data in a supported format - `JSON` or `MessagePack`. This documentation will provide an overview of how to perform the publication process from both the project's dashboard user interface and the command-line interface (CLI).

Using Project's Dashboard UI

To perform the publication process from the project's dashboard UI, please follow the steps below:

1. Navigate to the project's dashboard page.
2. Click on the *Publish* link.
3. Choose the format you want to export your data in - JSON or Message Pack.
4. Select the language(s) you want to publish.
5. Click on the *Finish* button to initiate the publication process.
6. Download the file.

Using Command-Line Interface (CLI)

To perform the publication process from the CLI, please follow the steps below:

1. Open the command-line interface.
2. Navigate to the game data's directory.
3. Use the `DATA EXPORT` command to publish the game data.

Example

```
charon DATA EXPORT --dataBase ".\gamedata.json" --mode publication --languages {en-US} --out
```

- Use the `--languages` parameter to specify the language(s) you want to publish. Or omit parameter to publish all languages.
- Use the `--outputFormat` parameter to specify the export format - `json` or `msgpack`.

See also

- Generating Source Code
- Working with Source Code (C# 4.0)
- Working with Source Code (C# 7.3)
- Working with Source Code (TypeScript)
- Command Line Interface (CLI)
- DATA EXPORT Command

Working with Source Code (C# 4.0)

> **Warning**
>
> This is deprecated code generator and shouldn't be used in new projects.

Accessing game data during runtime is possible by utilizing the generated source code.

This section provides examples using default class names, but it is possible to customize class names during the source code generation process. Additionally, this customization allows to avoid naming collisions with existing code.

Loading Game Data

The following C# code creates `GameData` class and loads your game data into memory.

```
using System.IO;

var fileStream = File.OpenRead("gamedata.json");
var gameData = new GameData(fileStream, GameData.Format.Json);
fileStream.Dispose();
```

The file `gamedata.json` could be published game data or original database file (.gdjs or .gdmp).

Accessing Documents

You can access your documents as a list:

```
var characters = gameData.GetCharacters() // -> ReadOnlyList<Character>
var characters = gameData.GetCharacters(onlyRoot: true) // -> ReadOnlyList<Character>
```

Or you can access specific documents by their `Id` or *Unique* properties:

```
var character = gameData.GetCharacter(characterId); // -> Character
var character = gameData.GetCharacterByName(characterName); // -> Character
```

`Settings` schemas are accessed by name:

```
var resetTime = gameData.LootSettings.ResetTime; // -> TimeSpan
```

Formulas

Formulas are executed with Invoke method:

```
var reward = gameData.LootSettings.RewardFormula.Invoke()  // -> int
```

Formula's parameters are passed as arguments of `Invoke` method.

Generated Code Extensions

When generating source code for game data, the resulting C# classes are declared as partial. This means that the classes can be extended by the programmer to add custom functionality.

For example, let's say that you have generated a `GameData` class for your game data. This class contains properties and methods for accessing and manipulating the data. However, you want to add some custom functionality to this class, such as a method for getting specific documents by criteria.

To do this, you can create a new C# file and declare a partial class with the same name as the generated `GameData` class. You can then define your custom method in this class, and it will be merged with the generated class at compile time.

Here is an example of how this could look:

```
// should be in same namespace as generated GameData class
public partial class GameData {
```

```
    public IEnumerable<Character> GetFighterCharacters() {
        return this.GetCharacters().Where(character => character.Role == CharacterRole.Fight
    }

}
```

In this example, the `GameData` class is declared as partial, and two partial classes are defined with the same name: one generated by the source code generation process and one containing custom code added by the programmer.

By using partial classes in this way, you can extend the functionality of the generated classes without modifying the generated code directly. This allows you to keep your custom code separate from the generated code, making it easier to maintain and update your game data classes over time.

There is also two extension points on `GameData` class:

```
partial void OnBeforeInitialize(); // Called after loading the data into lists and dictionar
partial void OnInitialize(); // Called after loading and prepping all data.
```

See also

- Generating Source Code
- GENERATE CSHARPCODE Command

Working with Source Code (C# 7.3)

Accessing game data during runtime is possible by utilizing the generated source code.

This section provides examples using default class names, but it is possible to customize class names during the source code generation process. Additionally, this customization allows to avoid naming collisions with existing code.

Loading Game Data

The following C# code creates `GameData` class and loads your game data into memory.

```
using System.IO;

var fileStream = File.OpenRead("RpgGameData.gdjs"); // or .json
var gameData = new GameData(fileStream, new Formatters.GameDataLoadOptions {
  Format = Formatters.Format.Json,
  // Patches = new [] { patchStream1, patchStream2, ... }
});
fileStream.Dispose();
```

The file `RpgGameData.gdjs` could be published game data or original database file (.gdjs or .gdmp).

Accessing Documents

You can access your documents as a list:

```
var allHeroes = gameData.AllHeroes.AsList // -> IReadOnlyList<Hero>
var heroes = gameData.Heroes.AsList // -> IReadOnlyList<Hero>
```

Or you can access specific documents by their `Id` or *Unique* properties:

```
var heroById = gameData.AllHeroes.Get(heroId); // -> Hero
var heroByName = gameData.AllHeroes.ByName().Get(heroName); // -> Hero
```

`Settings` schemas are accessed by name:

```
var startingHeroes = gameData.StartingSet.Heroes; // -> IReadOnlyList<Hero>
```

Formulas

Formulas are executed with Invoke method:

```
var reward = gameData.LootSettings.RewardFormula.Invoke()  // -> int
```

Formula's parameters are passed as arguments of `Invoke` method.

Extension of Generated Code

Partial Classes and Methods

When generating source code for game data, the resulting C# classes are declared as partial. This means that the classes can be extended by the programmer to add custom functionality.

For example, let's say that you have generated a `GameData` class for your game data. This class contains properties and methods for accessing and manipulating the data. However, you want to add some custom functionality to this class, such as a method for getting specific documents by criteria.

To do this, you can create a new C# file and declare a partial class with the same name as the generated `GameData` class. You can then define your custom method in this class, and it will be merged with the generated class at compile time.

Here is an example of how this could look:

```
// should be in same namespace as generated GameData class
public partial class GameData {

    public IEnumerable<Hero> GetReligiousHeroes() {
        return this.AllHeroes.AsList.Where(hero => hero.Religious);
    }

}
```

In this example, the `GameData` class is declared as partial, and two partial classes are defined with the same name: one generated by the source code generation process and one containing custom code added by the programmer.

By using partial classes in this way, you can extend the functionality of the generated classes without modifying the generated code directly. This allows you to keep your custom code separate from the generated code, making it easier to maintain and update your game data classes over time.

There is also extension point on `GameData` class:

```
partial void OnInitialize(); // Called after loading and prepping all data.
```

Customizing Attributes

You can append additional C# Attributes to the generated classes and their properties by modifying the `Specification` field of the related schema or property.

Attributes are specified using the `csAttribute` key in the `Specification` string, which uses the `application/x-www-form-urlencoded` format.

To help construct the correct value, you can use a spreadsheet formula (e.g., in Excel or Google Sheets):

```
# Place your attribute in cell A1
=TEXTJOIN("&", 1, IF(ISBLANK(A1), "", "&csAttribute=" & ENCODEURL(A1)))
```

Alternatively, use JavaScript to generate the encoded string:

```
const params = new URLSearchParams();
params.append("csAttribute", "Obsolete(\"Value is obsolete, do not use\")");
console.log(params.toString());
// → csAttribute=Obsolete%28%22Value%20is%20obsolete%2C%20do%20not%20use%22%29
```

After obtaining the encoded string, append it to the existing `Specification` value.

Example:

```
# Original Specification value:
icon=material&group=Metadata

# New attribute to add:
csAttribute=Obsolete%28%22Value%20is%20obsolete%2C%20do%20not%20use%22%29

# Final Specification value:
icon=material&group=Metadata&csAttribute=Obsolete%28%22Value%20is%20obsolete%2C%20do%20not%2
```

You can add multiple attributes by including multiple `csAttribute` keys:

```
csAttribute=Serializable&csAttribute=DebuggerDisplay%28%22Id%3D%7BId%7D%22%29
```

These attributes will be emitted directly into the generated C# code, attached to the appropriate class or property.

See also

- Generating Source Code
- GENERATE CSHARPCODE Command

Working with Source Code (Haxe)

Accessing game data during runtime is possible by utilizing the generated source code.

This section provides examples using default class names, but it is possible to customize class names during the source code generation process. Additionally, this customization allows to avoid naming collisions with existing code.

Loading Game Data

The following Haxe code creates `GameData` class and loads your game data into memory.

```
import GameData;
import Formatters;
import haxe.io.Path;
import sys.io.File;

var input = File.read("RpgGameData.gdjs"); // or .json
var options = new GameDataLoadOptions();
options.format = GameDataFormat.Json;
options.leaveInputsOpen = false;
// options.patches <-- put patches here
var gameData = new GameData(input, options);
```

The file `RpgGameData.gdjs` could be published game data or original database file (.gdjs or .gdmp).

Accessing Documents

You can access your documents as a list:

```
var allHeroes = gameData.heroesAll.list // -> ReadOnlyArray<Hero>
var heroes = gameData.heroes.list // -> ReadOnlyArray<Hero>
```

Or you can access specific documents by their `id` or *Unique* properties:

```
using GameData; // load byName() and other extension methods

var heroById = gameData.heroesAll.get(heroId); // -> Hero
var heroByName = gameData.heroesAll.byName().get(heroName); // -> Hero
```

`Settings` schemas are accessed by name:

```
var startingHeroes = gameData.startingSet.heroes; // -> ReadOnlyArray<Hero>
```

Formulas

Formulas are currently not supported.

Extension of Generated Code

Customizing Metadata

You can append additional metadata to the generated classes and their properties by modifying the `Specification` field of the related schema or property.

metadata annotations are specified using the `haxeAttribute` key in the `Specification` string, which uses the `application/x-www-form-urlencoded` format.

To help construct the correct value, you can use a spreadsheet formula (e.g., in Excel or Google Sheets):

```
# Place your attribute in cell A1
=TEXTJOIN("&", 1, IF(ISBLANK(A1), "", "&haxeAttribute=" & ENCODEURL(A1)))
```

Alternatively, use JavaScript to generate the encoded string:

```
const params = new URLSearchParams();
params.append("haxeAttribute", ":deprecated");
console.log(params.toString());
// → haxeAttribute=%3Adeprecated
```

After obtaining the encoded string, append it to the existing `Specification` value.

Example:

```
# Original Specification value:
icon=material&group=Metadata

# New attribute to add:
haxeAttribute=%3Adeprecated

# Final Specification value:
icon=material&group=Metadata&haxeAttribute=%3Adeprecated
```

You can add multiple metadata annotations by including multiple `haxeAttribute` keys:

```
haxeAttribute=broken&haxeAttribute=range%281%2C+2%29
```

These metadata annotations will be emitted directly into the generated Haxe code, attached to the appropriate class or property.

See also

- Generating Source Code
- GENERATE HAXE Command

Working with Source Code (Type Script)

Accessing game data during runtime is possible by utilizing the generated source code.

This section provides examples using default class names, but it is possible to customize class names during the source code generation process. Additionally, this customization allows to avoid naming collisions with existing code.

Loading Game Data

The following Type Script code creates `GameData` class and loads your game data into memory.

```
import { GameData } from './game.data';
import { Formatters } from './formatters';
```

```
// Node.js
import { readFileSync } from 'fs';
const gameDataStream = readFileSync(gameDataFilePath);

// Blob or File
const gameDataStream = gameDataFileBlob.arrayBuffer();

// XMLHttpRequest (XHR)
// gameDataRequest.responseType -> "arraybuffer"
const gameDataStream = gameDataRequest.response;

const gameData = new GameData(gameDataStream, {
  format: Formatters.GameDataFormat.Json,
  // patches: [patchStream1, patchStream2, ...]
});
```

The content of `gameDataStream` could be published game data or original database file (.gdjs or .gdmp).

Accessing Documents

You can access your documents as a list:

```
let heroes = gameData.heroesAll; // all heroes from all documents -> readonly Hero[]
let heroes = gameData.heroes; // heroes only from root collection -> readonly Hero[]
```

Or you can access specific documents by their `Id` or *Unique* properties:

```
let hero = gameData.heroesAll.find(heroId); // -> Hero | undefined
let hero = gameData.heroesAll.withOtherKey('Name').find(heroName); // -> Hero | undefined
```

`Settings` schemas are accessed by name:

```
let resetTime = gameData.lootSettings.resetTime; // -> TimeSpan
```

Formulas

Formulas inherit the `Function` type and can be invoked as-is or with `invoke` method:

```
var reward = gameData.lootSettings.rewardFormula()  // -> number
// or
var reward = gameData.lootSettings.rewardFormula.invoke()  // -> number
```

Formula's parameters are passed as arguments of `invoke` method.

Any non-game data related types are imported from `formula.known.types.ts`, which should be created by the developer and have all required types exported. Here is an example of a `formula.known.types.ts` file:

```
import { MyFormulaContext } from '../my.formula.context';

// example of re-exporting MyFormulaContext type.
export MyFormulaContext;

// example of Assets.Scripts.CheckContext.
export namespace Assets.Scripts {
  export class CheckContext {
    myField: string;
  }
}
```

Extension of Generated Code

Customizing Decorators

You can append additional TypeScript Decorators to the generated classes and their properties by modifying the `Specification` field of the related schema or property.

Decorators are specified using the `tsAttribute` key in the `Specification` string, which uses the `application/x-www-form-urlencoded` format.

To help construct the correct value, you can use a spreadsheet formula (e.g., in Excel or Google Sheets):

```
# Place your attribute in cell A1
=TEXTJOIN("&", 1, IF(ISBLANK(A1), "", "&tsAttribute=" & ENCODEURL(A1)))
```

Alternatively, use JavaScript to generate the encoded string:

```
const params = new URLSearchParams();
params.append("tsAttribute", "deprecated(\"Value is deprecated\")");
console.log(params.toString());
// → tsAttribute=deprecated%28%22Value+is+deprecated%22%29
```

After obtaining the encoded string, append it to the existing `Specification` value.

Example:

```
# Original Specification value:
icon=material&group=Metadata

# New attribute to add:
tsAttribute=deprecated%28%22Value+is+deprecated%22%29

# Final Specification value:
icon=material&group=Metadata&tsAttribute=deprecated%28%22Value+is+deprecated%22%29
```

You can add multiple attributes by including multiple `tsAttribute` keys:

```
tsAttribute=@Input&tsAttribute=range%281%2C+2%29
```

These attributes will be emitted directly into the generated TypeScript code, attached to the appropriate class or property.

See also

- Generating Source Code
- GENERATE TYPESCRIPTCODE Command

Working with Source Code (UE C++)

> **Warning**
>
> The source code for Unreal Engine requires a plugin to be installed to function. If you get compilation errors, make sure the plugin is installed and enabled.

Accessing game data during runtime is possible by utilizing the generated source code.

This section provides examples using default class names, but it is possible to customize class names during the source code generation process. Additionally, this customization allows to avoid naming collisions with existing code.

Loading Game Data

The following C++ code creates `UGameData` class and loads your game data into memory.

```
IFileManager& FileManager = IFileManager::Get();

const FString GameDataFilePath = TEXT("./RpgGameData.gdjs");  // or .json
const TUniquePtr<FArchive> GameDataStream = TUniquePtr<FArchive>(FileManager.CreateFileReade

UGameData* GameData = NewObject<UGameData>();

FGameDataLoadOptions Options;
Options.Format = EGameDataFormat::Json;
// Options.Patches.Add(PatchStream1);
// Options.Patches.Add(PatchStream2);
// ...

if (!GameData->TryLoad(GameDataStream.Get(), Options))
{
  // Handle failure
}
```

The file `RpgGameData.gdjs` could be published game data or original database file (.gdjs or .gdmp).

Accessing Documents

You can access your documents as a list:

```
auto AllHeroes = GameData->AllHeroes // -> TMap<string, UHero>
auto Heroes = GameData->Heroes // -> TMap<string, UHero>
```

`Settings` schemas are accessed by name:

```
auto StartingHeroes = GameData->StartingSet.Heroes; // -> TMap<string, UHero>
```

Formulas

Formulas are currently not supported.

Extension of Generated Code

Customizing Metadata

You can append additional or replace existing macro to the generated classes and their properties by modifying the `Specification` field of the related schema or property.

Metadata annotations are specified using the `uecppAttribute` key in the `Specification` string, which uses the `application/x-www-form-urlencoded` format.

To help construct the correct value, you can use a spreadsheet formula (e.g., in Excel or Google Sheets):

```
# Place your attribute in cell A1
=TEXTJOIN("&", 1, IF(ISBLANK(A1), "", "&uecppAttribute=" & ENCODEURL(A1)))
```

Alternatively, use JavaScript to generate the encoded string:

```
const params = new URLSearchParams();
params.append("uecppAttribute", "UPROPERTY(EditAnywhere, Meta = (Bitmask))");
console.log(params.toString());
// → uecppAttribute=UPROPERTY%28EditAnywhere%2C+Meta+%3D+%28Bitmask%29%29
```

After obtaining the encoded string, append it to the existing `Specification` value.

Example:

```
# Original Specification value:
icon=material&group=Metadata

# New attribute to add:
uecppAttribute=UCLASS%28BlueprintType%29

# Final Specification value:
icon=material&group=Metadata&uecppAttribute=UCLASS%28BlueprintType%29
```

These metadata annotations will be emitted directly into the generated C++ code, attached to the appropriate class or property.

See also

- Generating Source Code
- GENERATE UECPP Command

Command Line Interface (CLI)

Most of Charon functionality could be accessed via CLI commands. The application itself uses the getops syntax. You should be familiar with terminal on your OS to fully tap potential of CLI.

Installation

Download and install NET 8+.

Option 1: dotnet tool (recommended)

The easiest way to install is to use the infrastructure provided by the dotnet tool.

```
# install charon globally
dotnet tool install -g dotnet-charon

# install charon in current working directory
dotnet tool install dotnet-charon --local --create-manifest-if-needed
```

To update current tool use following commands:

```
# update global tool
dotnet tool update --global dotnet-charon

# update local tool
dotnet tool update dotnet-charon --local
```

Option 2: Bootstrap scripts

Alternatively, you can use one of two bootstrap scripts:

- RunCharon.bat (Windows)
- RunCharon.sh (Linux, MacOS)

Both scripts require the dotnet tool to be included in the system PATH. The scripts handle the installation of the Charon tool and ensure it stays up to date.

Windows

```
mkdir Charon
cd Charon
curl -O https://raw.githubusercontent.com/gamedevware/charon/main/scripts/bootstrap/RunCharo

.\RunCharon.bat DATA EXPORT --help
```

```
#               ^
#      your command goes here
```

Linux, MacOS

```
mkdir Charon
cd Charon
curl -O https://raw.githubusercontent.com/gamedevware/charon/main/scripts/bootstrap/RunCharo

chmod +x ./RunCharon.sh

./RunCharon.sh DATA EXPORT --help
#               ^
#      your command goes here
```

Command Syntax

Commands have the following syntax:

```
charon COMMAND --parameterName <parameter-value>

# parameters can have more than one value.
# Use space to separate values
charon EXPORT --schemas Item Armor "Project Settings" Quest

# if your value contains a space, put it inside the quotation marks.
# Escape characters and other rules depend on the OS you are running.
charon "c:\my application\my path.txt"

# some parameters don't require a value (e.g. flag).
charon VERSION --verbose
```

Absolute and relative paths

When running commands, it's crucial to be aware of whether you are using absolute or relative paths to files.

Understanding Paths

1. **Absolute Path**: An absolute path defines a file or directory's location in relation to the root directory. In Linux and macOS, it starts from the root `/`, while in Windows, it begins with a drive letter (like `C:\`).
    - Example for Linux/macOS: `/usr/local/bin`
    - Example for Windows: `C:\Program Files\mono`
2. **Relative Path**: A relative path references a file or directory in relation to the current working directory, without starting with a root slash or drive letter.
    - Example: If currently in `/home/user/Documents`, a file in `/home/user/Documents/Projects` would have the relative path `Projects/FileName`.

Usage in Terminals

- **Windows Command Prompt**: Paths use backslashes (\). Absolute paths start with a drive letter (like `C:\Users\Name`), while relative paths use the file name or paths like `subfolder\file.txt`.
- **macOS/Linux Terminal**: Paths are denoted with forward slashes (/). Absolute paths begin from the root (/), and relative paths use `./` for the current directory or `../` to go up one level.

Getting Help Text

To display list of available commands add *–help* or */?*.

```
charon --help

#> Usage: charon <action> [--<param> || (--<param> <paramValue> ...) ...]
#>
#> Verbs:
#>  DATA        Data manipulation actions.
#>  GENERATE    Code generation actions.
#>  VERSION     Print version.

charon DATA EXPORT --help

#> Usage:
#>   DATA EXPORT --dataBase <URI> [--schemas [<TEXT>]] [--properties [<TEXT>]] [--languages
#>               ] [--outputFormat <TEXT>] [--outputFormattingOptions [<TEXT>]] [--mode <EXP
#>               TEXT>]]
```

Apply Patch (Merge)

Applies patch created with DATA CREATEPATCH command to a game data.

- CLI Installation
- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
charon DATA APPLYPATCH --dataBase "c:\my app\gamedata.json" --input "c:\my app\gamedata_patc

# remote game data
charon DATA APPLYPATCH --dataBase "https://charon.live/view/data/My_Game/develop/" --input "
```

Parameters

`--dataBase`

Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\my app\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--credentials`

The API key used to access remote server in case of *–dataBase* being URL.

`--input` Path to a file with patch to apply. Alternatively, you can use Standart Input or *URL <remote_input_output>*.

```
# standart input (default)
--input in
--input con

# absolute path (windows)
--input "c:\my app\gamedata_patch.json"

# absolute path (unix)
--input "/user/data/gamedata_patch.json"

# relative path (universal)
--input "./gamedata_patch.json"

# remote location (HTTP)
--input "http://example.com/gamedata_patch.json"

# remote location with authentication (FTP)
--input "ftp://user:password@example.com/gamedata_patch.json"
```

`--inputFormat` Format of imported data.

```
# Auto-detect by extension (default)
--inputFormat auto

# JSON
--inputFormat json

# BSON
--inputFormat bson

# Message Pack
--inputFormat msgpack

# XML (removed in 2025.1.1)
--inputFormat xml
```

`--inputFormattingOptions` Additional options for specified format.

This command supports universal parameters.

Create Backup

Backs up game data to a specified file. Saved data could be later used with DATA RESTORE command.
Also this command can be used to convert game data into different format.

- CLI Installation
- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
charon DATA BACKUP --dataBase "c:\my app\gamedata.json" --output "c:\my app\backup.msgpkg" -

# remote game data
charon DATA BACKUP --dataBase "https://charon.live/view/data/My_Game/develop/" --output "./b
```

Parameters

`--dataBase` Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\my app\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--credentials` The API key used to access remote server in case of *–dataBase* being URL.

`--output` Path to a backup file. If the file exists, it will be overwritten. The directory must already exist. Alternatively, you can output to Standard Error, Standard Output, /dev/null, or a URL.

```
# standart output (default)
--output out
--output con

# standart error
--output err

# null device
--output null

# absolute path (windows)
--output "c:\my app\backup.json"

# absolute path (unix)
--output "/user/data/backup.json"

# relative path (universal)
--output "./backup.json"

# remote location (HTTP)
--output "http://example.com/backup.json"

# remote location with authentication (FTP)
--output "ftp://user:password@example.com/backup.json"
```

`--outputFormat` Format of backed up data.

```
# JSON (default)
--outputFormat json

# Message Pack
--outputFormat msgpack
```

`--outputFormattingOptions` Additional options for specified format.

This command supports universal parameters.

Output

The back up data follows the general game data structure.

Create Document

Creates a new document. For a bulk creations use DATA IMPORT command with `--mode create`.
Only the first document from the `--input` will be processed.

- CLI Installation
- Commands Reference

- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
charon DATA CREATE --dataBase "c:\my app\gamedata.json" --schema Item --input "c:\my app\ite

# remote game data
charon DATA CREATE --dataBase "https://charon.live/view/data/My_Game/develop/" --schema Item
```

Parameters

`--dataBase`

Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\my app\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--credentials`

The API key used to access remote server in case of *–dataBase* being URL.

`--schema`

Name or identifier of the type (schema) of the new document.

```
# name
--schema Item

# id
--schema 55a4f32faca22e191098f3d9
```

`--validationOptions`

List of validation checks and repairs to perform during document creation.

```
# no checks
--validationOptions none

# repairs
--validationOptions repair
--validationOptions repair deduplicateIds
--validationOptions repair repairRequiredWithDefaultValue
--validationOptions repair eraseInvalidValue

# checks (default)
--validationOptions checkTranslation
--validationOptions checkRequirements
--validationOptions checkFormat
--validationOptions checkUniqueness
--validationOptions checkReferences
--validationOptions checkSpecification
--validationOptions checkConstraints
```

`--input` Path to a file with document. Alternatively, you can use Standart Input or URL.

```
# standart input (default)
--input in
--input con

# absolute path (windows)
--input "c:\my app\item.json"

# absolute path (unix)
--input "/user/data/item.json"

# relative path (universal)
--input "./item.json"

# remote location (HTTP)
--input "http://example.com/item.json"

# remote location with authentication (FTP)
--input "ftp://user:password@example.com/item.json"
```

`--inputFormat` Format of imported data.

```
# Auto-detect by extension (default)
--inputFormat auto

# JSON
--inputFormat json

# BSON
--inputFormat bson

# Message Pack
--inputFormat msgpack

# XML (removed in 2025.1.1)
--inputFormat xml
```

`--inputFormattingOptions` Additional options for specified format.

`--output` Path to a created document file. If the file exists, it will be overwritten. The directory must already exist. Alternatively, you can output to Standard Error, Standard Output, /dev/null, or a URL.

```
# standart output
--output out
--output con

# standart error
--output err

# null device (default)
--output null

# absolute path (windows)
--output "c:\my app\created_item.json"

# absolute path (unix)
--output /user/data/created_item.json

# relative path (universal)
--output "./created_item.json"

# remote location (HTTP)
--output "http://example.com/created_item.json"

# remote location with authentication (FTP)
--output "ftp://user:password@example.com/created_item.json"
```

`--outputFormat` Format of created data.

```
# JSON (default)
--outputFormat json

# BSON
--outputFormat bson

# Message Pack
--outputFormat msgpack

# XML (removed in 2025.1.1)
--outputFormat xml
```

`--outputFormattingOptions` Additional options for specified format.

This command supports universal parameters.

Input Data Schema

The data you input should follow this schema (recommended):

```
{
  "Collections": {
    "<Schema-Name>": [
      {
        // <Document>
      }
    ]
  }
}
```

This schema is also accepted:

```
{
  "<Schema-Name>": [
    {
      // <Document>
    }
  ]
}
```

A list of documents is accepted:

```
[
  {
    // <Document>
  }
]
```

And single document too:

```
{
  // <Document>
}
```

Output

Outputs the created document with all the edits that were made to make it conform to the schema.

```
{
  "Id": "Sword"

  /* rest of properties of created document */
}
```

Create Patch (Diff)

Outputs the differences between two game datas as a file that can be used later to DATA APPLYPATCH to another game data.

- CLI Installation
- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
charon DATA CREATEPATCH --dataBase "c:\my app\gamedata.json" --input "c:\my app\gamedata_pat

# remote game data
charon DATA CREATEPATCH --dataBase "https://charon.live/view/data/My_Game/develop/" --input
```

Parameters

`--dataBase1`

Absolute or relative path to a first game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase1 "c:\my app\gamedata.json"

# remote server
--dataBase1 "https://charon.live/view/data/My_Game/develop/"
```

`--dataBase2` Absolute or relative path to a second game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase2 "c:\my app\gamedata.json"

# remote server
--dataBase2 "https://charon.live/view/data/My_Game/develop/"
```

`--output` Path to a patch file. If the file exists, it will be overwritten. The directory must already exist. Alternatively, you can output to Standard Error, Standard Output, /dev/null, or a URL.

```
# standart output (default)
--output out
--output con

# standart error
--output err

# null device
--output null

# absolute path (windows)
--output "c:\my app\gamedata_patch.json"

# absolute path (unix)
--output /user/data/gamedata_patch.json

# relative path (universal)
--output "./gamedata_patch.json"

# remote location (HTTP)
--output "http://example.com/gamedata_patch.json"

# remote location with authentication (FTP)
--output "ftp://user:password@example.com/gamedata_patch.json
```

`--outputFormat` Format of exported data.

```
# JSON (default)
--outputFormat json

# BSON
--outputFormat bson

# Message Pack
--outputFormat msgpack

# XML (removed in 2025.1.1)
--outputFormat xml
```

`--outputFormattingOptions` Additional options for specified format.

`--credentials` This parameter sets the API key used to access *BOTH* remote servers. If this is not suitable, consider downloading the data locally and running this command on local files instead.

This command supports universal parameters.

Delete Document

Deletes a document. For a bulk deletion use DATA IMPORT command with `--mode delete`.

- CLI Installation

- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
charon DATA DELETE --dataBase "c:\my app\gamedata.json" --schema Item --id "Sword"

# remote game data
charon DATA DELETE --dataBase "https://charon.live/view/data/My_Game/develop/" --schema Item
```

Parameters

`--dataBase`

Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\my app\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--credentials`

The API key used to access remote server in case of *–dataBase* being URL.

`--schema`

Name or identifier of the type (schema) of deleting document.

```
# name
--schema Item

# id
--schema 55a4f32faca22e191098f3d9
```

`--id`

Identifier of deleting document.

```
# text
--id Sword

# number
--id 101
```

`--output` The path to a file where the *deleted document* should be placed. If the file exists, it will be overwritten. The directory must already exist. Alternatively, you can output to Standard Error, Standard Output, /dev/null, or a URL.

```
# standart output
--output out
--output con

# standart error
--output err

# null device (default)
--output null

# absolute path (windows)
--output "c:\my app\deleted_item.json"

# absolute path (unix)
--output /user/data/deleted_item.json

# relative path (universal)
--output "./deleted_item.json"

# remote location (HTTP)
--output "http://example.com/deleted_item.json"

# remote location with authentication (FTP)
--output "ftp://user:password@example.com/deleted_item.json"
```

`--outputFormat` Format for deleted document.

```
# JSON (default)
--outputFormat json

# BSON
--outputFormat bson

# Message Pack
--outputFormat msgpack

# XML (removed in 2025.1.1)
--outputFormat xml
```

`--outputFormattingOptions` Additional options for specified format.

This command supports universal parameters.

Output

Outputs the deleted document in its state at the time of deletion.

```
{
  "Id": "Sword"

  /* rest of properties of deleted document */
}
```

Export Data

Exports documents into a file.

- CLI Installation
- Commands Reference
- Universal Parameters

- URL-based Input/Output

Command

```
# local game data (windows)
charon DATA EXPORT --dataBase "c:\my app\gamedata.json" --schemas Character --output "c:\my

# remote game data
charon DATA EXPORT --dataBase "https://charon.live/view/data/My_Game/develop/" --schemas Cha
```

Parameters

`--dataBase`

Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\my app\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--credentials`

The API key used to access remote server in case of *–dataBase* being URL.

`--schemas`

A list of types of documents (schemas) to export. By default all schemas *EXCEPT* metadata are exported.

- Use space to separate multiple schemas.
- You can use wildcards (*) at the beginning and end of names.
- You can use identifiers in {} instead of names.
- You can exclude certain names by using an exclamation mark (!) at the beginning of their names.

```
# schema name
--schemas Character
--schemas Character Item

# all (default)
--schemas *

# masks
--schemas Char*
--schemas *Modifier
--schemas *Mod*

# schema id
--schemas {18d4bf318f3c49688087dbed}

# negation
--schemas Char* !Character
--schemas !*Item*

# excluding system schemas (Schema, SchemaProperty, ProjectSetti
--schemas ![system]
```

`--properties`

A list of properties or property types to export. By default all properties are exported.

- *Id* property always included
- Use space to separate multiple properties.
- You can use wildcards (*) at the beginning and end of names.
- You can use identifiers in {} instead of names.
- You can exclude certain names by using an exclamation mark (!) at the beginning of their names.
- You can use data type in [] instead of names.

`--languages`

List of languages to keep in exported data. Language's english name is used or language tag (BCP 47).

Use DATA I18N LANGUAGES to get list of used languages.

- Use space to separate multiple languages
- You can use wildcards (*) at the beginning and end of names.
- You can use LCID or CultureInfo.Name in {} instead of the name.
- You can exclude certain names by using an exclamation mark (!) at the beginning of their names.

```
# language tag (BCP 47)
--languages {en-US}

# language name
--languages "Spanish (Spain)"

# language name mask
--languages Spanish*

# language LCID
--languages {3082}

# negation and masks
--languages !Spanish*
--languages Spanish* !{es-Es}
```

`--mode`

Export mode controls stripping and inclusion rules for exported data.

```
# (default)
--mode normal

--mode publication
--mode extraction
--mode localization
```

normal

Export all specified documents defined in *–schemas*. This mode ensures that the exported graph of documents remains valid by including any necessary additional documents to avoid any broken references.

publication

Same as *–mode normal*, but all non-essential data will be stripped. The result of the export can be safely loaded within the game with the generated code.

extraction

Export only the specified *–schemas* without exporting any referenced documents. In this mode, the exported graph of documents may contain broken references. It is recommended to use the import *–mode safeupdate* when importing this data back.

localization

Same as *–mode extraction* but only `LocalizedText` properties are exported.

`--output`

Path to a exported data file. If the file exists, it will be overwritten. The directory must already exist. Alternatively, you can output to Standard Error, Standard Output, /dev/null, or a URL.

```
# standart output (default)
--output out
--output con

# standart error
--output err

# null device
--output null

# absolute path (windows)
--output "c:\my app\document.json"

# absolute path (unix)
--output /user/data/document.json

# relative path (universal)
--output "./document.json"

# remote location (HTTP)
--output "http://example.com/document.json"

# remote location with authentication (FTP)
--output "ftp://user:password@example.com/document.json"
```

`--outputFormat`	Format of exported data.
	`# JSON (default)`
	`--outputFormat json`
	
	`# BSON`
	`--outputFormat bson`
	
	`# Message Pack`
	`--outputFormat msgpack`
	
	`# XML (removed in 2025.1.1)`
	`--outputFormat xml`
	
	`# XLSX Spreadsheet`
	`--outputFormat xlsx`

`--outputFormattingOptions`	Additional options for specified format.

This command supports universal parameters.

Output

The exported data follows the general game data structure, but omits *ToolsVersion*, *RevisionHash*, and *ChangeNumber* when the export mode is **not** set to publication.

```
{
  "Collections":
  {
    "Character":
    [
      {
        "Id": "Knight"

        /* rest of properties of document */
      },
      {
        "Id": "Templar"

        /* rest of properties of document */
      },
      // ...
    ]
  }
}
```

Modifying Exported Data with *yq*

The exported data can be accessed or modified using the *yq* tool, a lightweight and portable command-line YAML, JSON, and XML processor. *yq* uses *jq*-like syntax and supports common operations for manipulating structured data.

To use *yq* with exported JSON data:

1. **Install `yq`**: Follow the installation instructions from the official *yq* documentation: https://mikefarah.gitbook.io/yq/.
2. **Query Data**: Use *yq* to query specific fields or values from the exported JSON file.

   ```
   # Query a specific field
   yq '.Collections.Character[0].name' characters.json
   ```
3. **Modify Data**: Use *yq* to update or add fields in the exported JSON file.

   ```
   # Export data
   charon DATA EXPORT --dataBase gamedata.json --schemas Character --output characters.json
   ```

```
# Update a field
yq -i '.Collections.Character[0].name = "New Name"' characters.json

# Add a new field
yq -i '.Collections.Character[0].level = 10' characters.json

# Import data back
charon DATA IMPORT --dataBase gamedata.json --schemas Character --input characters.json
```

4. **Convert Formats**: *yq* can also convert between JSON, YAML, and other supported formats.

```
# Convert JSON to YAML
yq -o=yaml characters.json > characters.yaml
```

For more advanced usage, refer to the *yq* documentation: https://mikefarah.gitbook.io/yq/.

Find Document

Seaches for a document.

- CLI Installation
- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
charon DATA FIND --dataBase "c:\my app\gamedata.json" --schema Character --id John

# remote game data
charon DATA FIND --dataBase "https://charon.live/view/data/My_Game/develop/" --schema Charac
```

Parameters

`--dataBase`

Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\my app\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--credentials`

The API key used to access remote server in case of *–dataBase* being URL.

`--schema`

Name or identifier of the type (schema) of document.

```
# name
--schema Item

# id
--schema 55a4f32faca22e191098f3d9
```

`--id`

Identifier of document.

```
# text
--id Sword

# number
--id 101
```

`--output` Path to a found document file. If the file exists, it will be overwritten. The directory must already exist. Alternatively, you can output to Standard Error, Standard Output, /dev/null, or a URL.

```
# standart output (default)
--output out
--output con

# standart error
--output err

# null device
--output null

# absolute path (windows)
--output "c:\my app\document.json"

# absolute path (unix)
--output /user/data/document.json

# relative path (universal)
--output "./document.json"

# remote location (HTTP)
--output "http://example.com/document.json"

# remote location with authentication (FTP)
--output "ftp://user:password@example.com/document.json"
```

`--outputFormat` Format of exported data.

```
# JSON (default)
--outputFormat json

# BSON
--outputFormat bson

# Message Pack
--outputFormat msgpack

# XML (removed in 2025.1.1)
--outputFormat xml
```

`--outputFormattingOptions` Additional options for specified format.

This command supports universal parameters.

Output

Outputs the found document.

```
{
  "Id": "John"

  /* rest of properties of found document */
}
```

Add Translation Languages

Add translation languages to specified game data.

- CLI Installation
- Universal Parameters
- Commands Reference

Command

```
# local game data (windows)
charon DATA I18N ADDLANGUAGE --dataBase "c:\my app\gamedata.json" --languages "es-ES" "en-GB

# remote game data
charon DATA I18N ADDLANGUAGE --dataBase "https://charon.live/view/data/My_Game/develop/" --l
```

Parameters

`--dataBase`

Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\my app\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--languages`

The list of languages to add. Values are language tags (BCP 47) separated by space..

This command supports universal parameters.

Export Translated Data

Export text that can be translated into a file.

- CLI Installation
- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
charon DATA I18N EXPORT --dataBase "c:\my app\gamedata.json" --schemas Character --sourceLan

# remote game data
charon DATA I18N EXPORT --dataBase "https://charon.live/view/data/My_Game/develop/" --schema
```

Parameters

`--dataBase`

Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\my app\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--credentials`

The API key used to access remote server in case of *–dataBase* being URL.

`--schemas`

A list of types of documents (schemas) to export. By default all schemas *EXCEPT* metadata are exported.

- Use space to separate multiple schemas.
- You can use wildcards (*) at the beginning and end of names.
- You can use identifiers in {} instead of names.
- You can exclude certain names by using an exclamation mark (!) at the beginning of their names.

```
# schema name
--schemas Character
--schemas Character Item

# all (default)
--schemas *

# masks
--schemas Char*
--schemas *Modifier
--schemas *Mod*

# schema id
--schemas {18d4bf318f3c49688087dbed}

# negation
--schemas Char* !Character
--schemas !*Item*

# excluding system schemas (Schema, SchemaProperty, ProjectSettings)
--schemas ![system]
```

`--sourceLanguage`

Source (original) language for translation. Value is language tag (BCP 47).

Use DATA I18N LANGUAGES to get list of used languages.

```
# it is used as <source> in XLIFF
--sourceLanguage en-US
```

`--targetLanguage`

Target language for translation. Value is language tag (BCP 47).

```
# it is used as <target> in XLIFF
--targetLanguage es-ES
```

`--output` — Path to a file to which data will be exported. If the file exists, it will be overwritten. The directory must already exist. Alternatively, you can output to Standard Error, Standard Output, /dev/null, or a URL.

```
# standart output (default)
--output out
--output con

# standart error
--output err

# null device
--output null

# absolute path (windows)
--output "c:\my app\input.json"

# absolute path (unix)
--output /user/data/input.json

# relative path (universal)
--output "./input.json"

# remote location (HTTP)
--output "http://example.com/input.json"

# remote location with authentication (FTP)
--output "ftp://user:password@example.com/input.json"
```

`--outputFormat` — Format of exported data.

```
# XLIFF v2 (default)
--outputFormat xliff
--outputFormat xliff2

# XLIFF v1
--outputFormat xliff1

# XSLX Spreadsheet
--outputFormat xslx

# JSON
--outputFormat json
```

`--outputFormattingOptions` — Additional options for specified format.

This command supports universal parameters.

Output

The exported data follows the general game data structure, but omits *ToolsVersion*, *RevisionHash*, and *ChangeNumber* fields.

Importing Translated Data

Import translated text from a specified file into game data.

- CLI Installation
- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
charon DATA I18N IMPORT --dataBase "c:\my app\gamedata.json" --input "c:\my app\character_lo

# remote game data
charon DATA I18N IMPORT --dataBase "https://charon.live/view/data/My_Game/develop/" --input
```

Parameters

`--dataBase`

Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\my app\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--credentials`

The API key used to access remote server in case of *–dataBase* being URL.

`--schemas`

A list of types of documents (schemas) to import. By default all schemas *EXCEPT* metadata are imported.

- Use space to separate multiple schemas.
- You can use wildcards (*) at the beginning and end of names.
- You can use identifiers in {} instead of names.
- You can exclude certain names by using an exclamation mark (!) at the beginning of their names.

```
# schema name
--schemas Character
--schemas Character Item

# all (default)
--schemas *

# masks
--schemas Char*
--schemas *Modifier
--schemas *Mod*

# schema id
--schemas {18d4bf318f3c49688087dbed}

# negation
--schemas Char* !Character
--schemas !*Item*

# excluding system schemas (Schema, SchemaProperty, ProjectSetti
--schemas ![system]
```

`--languages`

The list of languages to import. Values are language tags (BCP 47).

```
# Import specific language
--languages en-US es-ES

# Import all languages
--languages *
```

`--validationOptions` List of validation checks and repairs to perform during import.

```
# no checks
--validationOptions none

# repairs
--validationOptions repair
--validationOptions repair deduplicateIds
--validationOptions repair repairRequiredWithDefaultValue
--validationOptions repair eraseInvalidValue

# checks (default)
--validationOptions checkTranslation
--validationOptions checkRequirements
--validationOptions checkFormat
--validationOptions checkUniqueness
--validationOptions checkReferences
--validationOptions checkSpecification
--validationOptions checkConstraints
```

`--input` Path to a file with data to import. Alternatively, you can use Standart Input or URL.

See input data structure requirements.

```
# standart input (default)
--input in
--input con

# absolute path (windows)
--input "c:\my app\input.json"

# absolute path (unix)
--input /user/data/input.json

# relative path (universal)
--input "./input.json"

# remote location (HTTP)
--input "http://example.com/input.json"

# remote location with authentication (FTP)
--input "ftp://user:password@example.com/input.json"
```

`--inputFormat` Format of imported data.

```
# Auto-detect by extension (default)
--inputFormat auto

# XLIFF v2
--inputFormat xliff
--inputFormat xliff2

# XLIFF v1
--inputFormat xliff1

# XSLX Spreadsheet
--inputFormat xslx
```

`--inputFormattingOptions` Additional options for specified format.

`--output` — Optional path to a import report file. If the file exists, it will be overwritten. The directory must already exist. Alternatively, you can output to Standard Error, Standard Output, /dev/null, or a URL.

```
# standart output
--output out
--output con

# standart error
--output err

# null device (default)
--output null

# absolute path (windows)
--output "c:\my app\document.json"

# absolute path (unix)
--output /user/data/document.json

# relative path (universal)
--output "./document.json"

# remote location (HTTP)
--output "http://example.com/document.json"

# remote location with authentication (FTP)
--output "ftp://user:password@example.com/document.json"
```

`--outputFormat` — Format of import report.

```
# JSON (default)
--outputFormat json

# BSON
--outputFormat bson

# Message Pack
--outputFormat msgpack

# XLSX Spreadsheet
--outputFormat xlsx
```

`--outputFormattingOptions` — Additional options for specified format.

`--dryRun` — Allows you to run the command without actually making any changes to the game data, providing a preview of what would happen.

This command supports universal parameters.

List Translation Languages

Get a list of supported translation languages. Primary language always shows up first in the list.

- CLI Installation
- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
charon DATA I18N LANGUAGES --dataBase "c:\my app\gamedata.json" --output out --outputFormat

# remote game data
charon DATA I18N LANGUAGES --dataBase "https://charon.live/view/data/My_Game/develop/" --out
```

Parameters

`--dataBase`

Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\my app\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--credentials`

The API key used to access remote server in case of *–dataBase* being URL.

`--output`

Path to language list file. If the file exists, it will be overwritten. The directory must already exist. Alternatively, you can output to Standard Error, Standard Output, /dev/null, or a URL.

```
# standart output (default)
--output out
--output con

# standart error
--output err

# null device
--output null

# absolute path (windows)
--output "c:\my app\input.json"

# absolute path (unix)
--output /user/data/input.json

# relative path (universal)
--output "./input.json"

# remote location (HTTP)
--output "http://example.com/input.json"

# remote location with authentication (FTP)
--output "ftp://user:password@example.com/input.json"
```

`--outputFormat` — Format of exported data.

```
# JSON (default)
--outputFormat json

#> [
#>   "en-US",
#>   "es-ES",
#> ]


# Space separated list
--outputFormat list

#> en-US es-ES

# New line (OS specific) separated list
--outputFormat table

#> en-US
    #> es-ES
```

`--outputFormattingOptions` — Additional options for specified format.

This command supports universal parameters.

Import Data

Imports documents from file to a game data.

- CLI Installation
- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
charon DATA IMPORT --dataBase "c:\my app\gamedata.json" --schemas Character --input "c:\my a

# remote game data
charon DATA IMPORT --dataBase "https://charon.live/view/data/My_Game/develop/" --schemas Cha
```

Parameters

`--dataBase` — Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\my app\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--credentials` — The API key used to access remote server in case of *–dataBase* being URL.

`--schemas` A list of types of documents (schemas) to import. By default all schemas *EXCEPT* metadata are imported.

- Use space to separate multiple schemas.
- You can use wildcards (*) at the beginning and end of names.
- You can use identifiers in {} instead of names.
- You can exclude certain names by using an exclamation mark (!) at the beginning of their names.

```
# schema name
--schemas Character
--schemas Character Item

# all (default)
--schemas *

# masks
--schemas Char*
--schemas *Modifier
--schemas *Mod*

# schema id
--schemas {18d4bf318f3c49688087dbed}

# negation
--schemas Char* !Character
--schemas !*Item*

# excluding system schemas (Schema, SchemaProperty, ProjectSettings)
--schemas ![system]
```

`--mode` Import mode controls merge behavior during import.

```
# (default)
--mode createAndUpdate

--mode create
--mode update
--mode safeUpdate
--mode replace
--mode delete
```

createAndUpdate
: creates new documents and updates existing ones

create
: only creates new documents, existing documents are kept unchanged

update
: only updates existing documents, no new ones are created

safeUpdate
: same as *update* but without creating, moving and erasing embedded documents

replace
: replaces the entire collection with the imported documents

delete
: deletes documents found in the imported data

`--validationOptions` List of validation checks and repairs to perform during import.

```
# no checks
--validationOptions none

# repairs
--validationOptions repair
--validationOptions repair deduplicateIds
--validationOptions repair repairRequiredWithDefaultValue
--validationOptions repair eraseInvalidValue

# checks (default)
--validationOptions checkTranslation
--validationOptions checkRequirements
--validationOptions checkFormat
--validationOptions checkUniqueness
--validationOptions checkReferences
--validationOptions checkSpecification
--validationOptions checkConstraints
```

`--input` Path to a data file. Alternatively, you can use Standart Input or URL.

```
# standart input (default)
--input in
--input con

# absolute path (windows)
--input "c:\my app\characters.json"

# absolute path (unix)
--input "/user/data/characters.json"

# relative path (universal)
--input "./characters.json"

# remote location (HTTP)
--input "http://example.com/characters.json"

# remote location with authentication (FTP)
--input "ftp://user:password@example.com/characters.json"
```

`--inputFormat` Format of imported data.

```
# Auto-detect by extension (default)
--inputFormat auto

# JSON
--inputFormat json

# BSON
--inputFormat bson

# Message Pack
--inputFormat msgpack

# XML (removed in 2025.1.1)
--inputFormat xml

# XLSX Spreadsheet
--inputFormat xlsx
```

`--inputFormattingOptions` Additional options for specified format.

`--output`

Optional path to a import report file. If the file exists, it will be overwritten. The directory must already exist. Alternatively, you can output to Standard Error, Standard Output, /dev/null, or a URL.

```
# standart output
--output out
--output con

# standart error
--output err

# null device (default)
--output null

# absolute path (windows)
--output "c:\my app\document.json"

# absolute path (unix)
--output /user/data/document.json

# relative path (universal)
--output "./document.json"

# remote location (HTTP)
--output "http://example.com/document.json"

# remote location with authentication (FTP)
--output "ftp://user:password@example.com/document.json"
```

`--outputFormat`

Format of import report.

```
# JSON (default)
--outputFormat json

# BSON
--outputFormat bson

# Message Pack
--outputFormat msgpack

# XLSX Spreadsheet
--outputFormat xlsx
```

`--outputFormattingOptions`

Additional options for specified format.

`--dryRun`

Allows you to run the command without actually making any changes to the game data, providing a preview of what would happen.

This command supports universal parameters.

Input Data Structure

The data you input should follow this structure (recommended):

```
{
  "Collections": {
    "<Schema-Name>": [
      {
        // <Document>
      }
    ]
  }
}
```

This structure is also accepted:

```
{
  "<Schema-Name>": [
    {
      // <Document>
    }
  ]
}
```

A list of documents is accepted if only one name in `--schemas` is specified:

```
[
  {
    // <Document>
  }
]
```

And single document is accepted too if only one name in `--schemas` is specified:

```
{
  // <Document>
}
```

List Documents

Seaches for a documents.

- CLI Installation
- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
charon DATA LIST --dataBase "c:\my app\gamedata.json" --schema Character

# remote game data
charon DATA LIST --dataBase "https://charon.live/view/data/My_Game/develop/" --schema Charac
```

Parameters

`--dataBase` — Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\my app\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--credentials` — The API key used to access remote server in case of *–dataBase* being URL.

`--schema` — Name or identifier of the type (schema) of document.

```
# name
--schema Item

# id
--schema 55a4f32faca22e191098f3d9
```

`--filters` Document filter expressions.

```
# patterns
--filters <Field> <Operator> <Value> [<Field> <Operator> <Value>...]

# single expression
--filters Id > 10
--filters Name like "Zombie"

# multiple expressions
--filters Id > 10 Name like "Zombie"

# greater than
--filters Id > 0
--filters Id greaterThan 0


# greater than or equal
--filters Id >= 0
--filters Id greaterThanOrEqual 0


# less than
--filters Id < 0
--filters Id lessThan 0

# less than or equal
--filters Id <= 0
--filters Id LessThanOrEqual 0

# equal
--filters Id = 0
--filters Id == 0
--filters Id equal 0

# not equal
--filters Id <> 0
--filters Id != 0
--filters Id notEqual 0

# like - is used to search for specific patterns in a field, allowing for parti
--filters Name like "Zombie"
```

`--sorters` Document sort expressions.

```
# patterns
--sorters <Field> ASC|DESC [<Field> ASC|DESC ...]

# ascending
--sorters Name ASC

# descending
--sorters Name DESC
```

`--path` Embeddance path filter. Could be used to get only embedded documents.

```
# any path
--path *

# root documents (default)
--path ""

# in 'Item' property
--path /Item
```

`--skip` Number of found documents to skip.

```
# skip first ten documents after applying --filter and --sor
--skip 10
```

`--take` Max amount to documents return.

```
# limit to first 100 documents after --skip
--take 100
```

`--output` Path to a found document file. If the file exists, it will be overwritten. The directory must already exist. Alternatively, you can output to Standard Error, Standard Output, /dev/null, or a URL.

```
# standart output (default)
--output out
--output con

# standart error
--output err

# null device
--output null

# absolute path (windows)
--output "c:\my app\document.json"

# absolute path (unix)
--output /user/data/document.json

# relative path (universal)
--output "./document.json"

# remote location (HTTP)
--output "http://example.com/document.json"

# remote location with authentication (FTP)
--output "ftp://user:password@example.com/document.json"
```

`--outputFormat` Format of exported data.

```
# JSON (default)
--outputFormat json

# BSON
--outputFormat bson

# Message Pack
--outputFormat msgpack

# XML (removed in 2025.1.1)
--outputFormat xml
```

`--outputFormattingOptions` Additional options for specified format.

This command supports universal parameters.

Output

The exported data follows the general game data structure, but omits *ToolsVersion*, *RevisionHash*, and *ChangeNumber*.

```
{
  "Collections":
  {
    "Character":
    [
      {
        "Id": "Knight"

        /* rest of properties of document */
      },
      {
        "Id": "Templar"

        /* rest of properties of document */
      },
      // ...
    ]
  }
}
```

Restore from Backup

Restores game data from a file created by DATA BACKUP command.

- CLI Installation
- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
 charon DATA RESTORE --dataBase "c:\my app\gamedata.json" --input "c:\my app\backup.msgpkg"

# remote game data
 charon DATA RESTORE --dataBase "https://charon.live/view/data/My_Game/develop/" --input "./
```

Parameters

`--dataBase`

Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\my app\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--credentials`

The API key used to access remote server in case of *–dataBase* being URL.

`--input` Path to a backup file. Alternatively, you can use Standart Input or URL.

```
# standart input (default)
--input in
--input con

# absolute path (windows)
--input "c:\my app\backup.json"

# absolute path (unix)
--input "/user/data/backup.json"

# relative path (universal)
--input "./backup.json"

# remote location (HTTP)
--input "http://example.com/backup.json"

# remote location with authentication (FTP)
--input "ftp://user:password@example.com/backup.json"
```

`--inputFormat` Format of imported data.

```
# Auto-detect by extension (default)
--inputFormat auto

# JSON
--inputFormat json

# Message Pack
--inputFormat msgpack
```

`--inputFormattingOptions` Additional options for specified format.

This command supports universal parameters.

Update Document

Updates a document. For a bulk updates use DATA IMPORT command with `--mode update`.
The update document in `--input` may be partial, with non-included fields being omitted.
Only the first document from the `--input` will be processed.

- CLI Installation
- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
charon DATA UPDATE --dataBase "c:\my app\gamedata.json" --schema Item --input "c:\my app\ite

# remote game data
charon DATA UPDATE --dataBase "https://charon.live/view/data/My_Game/develop/" --schema Item
```

Parameters

`--dataBase` Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\my app\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--credentials` The API key used to access remote server in case of *–dataBase* being URL.

`--schema` Name or identifier of the type (schema) of updated document.

```
# name
--schema Item

# id
--schema 55a4f32faca22e191098f3d9
```

`--id` Identifier of updated document. Could be omitted if *Id* is specified in `--input` document.

```
# text
--id Sword

# number
--id 101
```

`--validationOptions` List of validation checks and repairs to perform during document update.

```
# no checks
--validationOptions none

# repairs
--validationOptions repair
--validationOptions repair deduplicateIds
--validationOptions repair repairRequiredWithDefaultValue
--validationOptions repair eraseInvalidValue

# checks (default)
--validationOptions checkTranslation
--validationOptions checkRequirements
--validationOptions checkFormat
--validationOptions checkUniqueness
--validationOptions checkReferences
--validationOptions checkSpecification
--validationOptions checkConstraints
```

`--input` Path to a file with update data. Alternatively, you can use Standart Input or URL.

```
# standart input (default)
--input in
--input con

# absolute path (windows)
--input "c:\my app\item.json"

# absolute path (unix)
--input "/user/data/item.json"

# relative path (universal)
--input "./item.json"

# remote location (HTTP)
--input "http://example.com/item.json"

# remote location with authentication (FTP)
--input "ftp://user:password@example.com/item.json"
```

`--inputFormat` Format of update data.

```
# Auto-detect by extension (default)
--inputFormat auto

# JSON
--inputFormat json

# BSON
--inputFormat bson

# Message Pack
--inputFormat msgpack

# XML (removed in 2025.1.1)
--inputFormat xml
```

`--inputFormattingOptions` Additional options for specified format.

`--output` Path to a updated document file. If the file exists, it will be overwritten. The directory must already exist. Alternatively, you can output to Standard Error, Standard Output, /dev/null, or a URL.

```
# standart output
--output out
--output con

# standart error
--output err

# null device (default)
--output null

# absolute path (windows)
--output "c:\my app\updated_item.json"

# absolute path (unix)
--output /user/data/updated_item.json

# relative path (universal)
--output "./updated_item.json"

# remote location (HTTP)
--output "http://example.com/updated_item.json"

# remote location with authentication (FTP)
--output "ftp://user:password@example.com/updated_item.json"
```

`--outputFormat` Format of updated data.

```
# JSON (default)
--outputFormat json

# BSON
--outputFormat bson

# Message Pack
--outputFormat msgpack

# XML (removed in 2025.1.1)
--outputFormat xml
```

`--outputFormattingOptions` Additional options for specified format.

This command supports universal parameters.

Input Data Schema

The data you input should follow this schema (recommended):

```
{
  "Collections": {
    "<Schema-Name>": [
      {
        // <Document>
      }
    ]
  }
}
```

This schema is also accepted:

```
{
  "<Schema-Name>": [
    {
      // <Document>
    }
  ]
}
```

A list of documents is accepted:

```
[
  {
    // <Document>
  }
]
```

And single document too:

```
{
  // <Document>
}
```

Output

Outputs the updated document with all the edits that were made to make it conform to the schema.

```
{
  "Id": "Sword"

  /* rest of properties of updated document */
}
```

Validate Game Data

Checks the game data for validity and produces a report.

The exit code will be `1` if the report contains errors and the `--output` is set to `err`. Otherwise, the exit code will be `0`.

- CLI Installation
- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
charon DATA DELETE --dataBase "c:\my app\gamedata.json" --schema Item --id "Sword"

# remote game data
charon DATA DELETE --dataBase "https://charon.live/view/data/My_Game/develop/" --schema Item
```

Parameters

`--dataBase` Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\my app\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--credentials` The API key used to access remote server in case of *–dataBase* being URL.

`--validationOptions` List of validation checks and repairs to perform.

```
# no checks
--validationOptions none

# repairs
--validationOptions repair
--validationOptions repair deduplicateIds
--validationOptions repair repairRequiredWithDefaultValue
--validationOptions repair eraseInvalidValue

# checks (default)
--validationOptions checkTranslation
--validationOptions checkRequirements
--validationOptions checkFormat
--validationOptions checkUniqueness
--validationOptions checkReferences
--validationOptions checkSpecification
--validationOptions checkConstraints
```

`--output` Path to a validation report file. If the file exists, it will be overwritten. The directory must already exist. Alternatively, you can output to Standard Error, Standard Output, /dev/null, or a URL.

```
# standart output
--output out
--output con

# standart error
--output err

# null device (default)
--output null

# absolute path (windows)
--output "c:\my app\document.json"

# absolute path (unix)
--output /user/data/document.json

# relative path (universal)
--output "./document.json"

# remote location (HTTP)
--output "http://example.com/document.json"

# remote location with authentication (FTP)
--output "ftp://user:password@example.com/document.json"
```

`--outputFormat`	Format of exported data.
	`# JSON (default)`
	`--outputFormat json`
	
	`# BSON`
	`--outputFormat bson`
	
	`# Message Pack`
	`--outputFormat msgpack`
	
	`# XML (removed in 2025.1.1)`
	`--outputFormat xml`

`--outputFormattingOptions`	Additional options for specified format.

This command supports universal parameters.

Output Data Schema

The report follow this pattern:

```
{
  records:
  [
    {
      id: "<document-id>",
      schemaId: "<schema-id>",
      schemaName: "<schema-name>",
      errors: // could be null if no errors
      [
        {
          path: "<path-in-document>",
          message: "<error-message>",
          code: "<error-code>"
        },
        // ...
      ]
    },
    // ...
  ]
}
```

or JSON schema:

```
{
  "type": "object",
  "x-name": "ValidationReport",
  "additionalProperties": false,
  "properties": {
    "records": {
      "type": "array",
      "items": {
        "type": "object",
        "x-name": "ValidationRecord",
        "additionalProperties": false,
        "properties": {
          "id": { },
          "schemaName": {
            "type": "string"
          },
          "schemaId": {
```

```
            "type": "string"
          },
          "errors": {
            "type": "array",
            "items": {
              "type": "object",
              "x-name": "ValidationError",
              "additionalProperties": true,
              "readOnly": true,
              "properties": {
                "path": {
                  "type": "string"
                },
                "message": {
                  "type": "string"
                },
                "code": {
                  "type": "string"
                }
              }
            }
          }
        }
      },
      "metadataHashCode": {
        "type": "integer",
        "format": "int32"
      }
    }
  }
```


Generate C# Source Code

Generates C# source code for game data into output directory.

This command does not delete previously generated files, and it is the responsibility of the user to ensure that any previous files are removed before running the command again.

- CLI Installation
- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
charon GENERATE CSHARPCODE --dataBase "c:\my app\gamedata.json" --namespace "MyGame.Paramete

# remote game data
charon GENERATE CSHARPCODE --dataBase "https://charon.live/view/data/My_Game/develop/" --nam
```

Parameters

`--dataBase` Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\my app\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--credentials` The API key used to access remote server in case of *–dataBase* being URL.

`--outputDirectory` Specifies the path where the source code should be written. It can be either an absolute or relative path to a directory. The specified directory must already be present.

```
# Windows
--outputDirectory "c:\my app\scripts"

# Linux or OSX
--outputDirectory "~/my app/scripts"

# Relative path
--outputDirectory "./my app/scripts"
```

`--languageVersion` Target C# version. By default it is 4.0.

```
--languageVersion CSharp40
--languageVersion CSharp73
```

`--documentClassName` Name for base class for all documents.

```
# name (default)
--documentClassName Document

# in case of name collision
--documentClassName GameDataDocument
```

`--gameDataClassName` Name for class containing whole in-memory game data.

```
# name (default)
--gameDataClassName GameData

# in case of name collision
--gameDataClassName MyGameData
```

`--namespace` Namespace for all generated classes.

```
# name (default)
--namespace GameParameters
```

`--defineConstants` Preprocessor constants to define. Use semicolon(;) to separate multiple values.

```
# Use GameDevWare.Dynamic.Expressions.dll for formulas
--defineConstants USE_DYNAMIC_EXPRESSIONS

# Exclude all formula related code from compilation
--defineConstants SUPPRESS_BUILD_IN_FORMULAS

# Enable some JSON formatting optimizations using System.Memory.dll and Sy
--defineConstants BUFFERS_TEXT_DLL
```

`--indentation` Indentation style for generated code.

```
# Tabs (default)
--indentation Tabs

# Two spaces
--indentation TwoSpaces

# Four spaces
--indentation FourSpaces
```

`--lineEndings` Line ending symbols for generated code.

```
# Windows \\r\\n (default)
--lineEndings Windows

# Unix style \\n
--lineEndings Unix
```

`--splitFiles` Set this flag to lay out generated classes into separate files. If not set, then one giant file with the name of `--gameDataClassName`.cs will be generated.

`--optimizations` List of enabled optimization in generated code.

```
# Eagerly resolves and validates all references in loaded documents.
# When enabled, this optimization ensures that all references in documents
# during loading. This comes with a performance cost but guarantees the va
--optimizations eagerReferenceResolution

# Opts for raw references without generating helper methods for referenced
# With this optimization, the generated code will not include helper metho
# referenced documents, keeping only accessors that work with raw referenc
--optimizations rawReferences

# Avoids generating helper methods for localized strings, keeping only raw
# This optimization eliminates helper methods for accessing localized text
# accessors that deal directly with lists of localized texts.
--optimizations rawLocalizedStrings

# Disables string pooling during game data loading.
# Turning off string pooling can yield a minor performance improvement at
# memory usage, as it avoids reusing short strings.
--optimizations disableStringPooling

# Disables generation of code for loading game data from JSON formatted fi
# This optimization omits code related to JSON serialization, useful when
# game data is not used.
--optimizations disableJsonSerialization

# Disables generation of code for loading game data from Message Pack form
# Similar to DisableJsonSerialization, this option removes code related to
# from Message Pack formatted files.
--optimizations disableMessagePackSerialization

# Disables generation of code related to applying patches during game data
# This removes a significant portion of code that is mainly used for moddi
# where patches are applied to game data at runtime.
--optimizations disablePatching

# Disables generation of enums with known document IDs.
# This removes a significant portion of code that contains listings of IDs
# documents known at the moment of code generation, which improves compila
--optimizations disableDocumentIdEnums
```

`--clearOutputDirectory`	Clear the output directory from generated files. Generated files are identified by the presence of the '<auto-generated>' tag inside.

This command supports universal parameters.

Generate Haxe Source Code

Generates Haxe source code for game data into output directory.

This command does not delete previously generated files, and it is the responsibility of the user to ensure that any previous files are removed before running the command again.

- CLI Installation
- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
 charon GENERATE HAXE --dataBase "c:\my app\gamedata.json" --packageName "" --outputDirector

# remote game data
 charon GENERATE HAXE --dataBase "https://charon.live/view/data/My_Game/develop/" --packageN
```

Parameters

`--dataBase`	Absolute or relative path to game data. Use quotation marks if your path contains spaces.
	`# local file`
	`--dataBase "c:\my app\gamedata.json"`
	
	`# remote server`
	`--dataBase "https://charon.live/view/data/My_Game/develop/"`

`--credentials`	The API key used to access remote server in case of *–dataBase* being URL.

`--outputDirectory`	Specifies the path where the source code should be written. It can be either an absolute or relative path to a directory. The specified directory must already be present.
	`# Windows`
	`--outputDirectory "c:\my app\scripts"`
	
	`# Linux or OSX`
	`--outputDirectory "~/my app/scripts"`
	
	`# Relative path`
	`--outputDirectory "./my app/scripts"`

`--documentClassName`	Name for base class for all documents.
	`# name (default)`
	`--documentClassName Document`
	
	`# in case of name collision`
	`--documentClassName GameDataDocument`

`--gameDataClassName`

Name for class containing whole in-memory game data.

```
# name (default)
--gameDataClassName GameData

# in case of name collision
--gameDataClassName MyGameData
```

`--packageName`

Package name for all generated classes.

```
# empty package (default)
--packageName ""

# named
--packageName GameParameters
```

`--indentation`

Indentation style for generated code.

```
# Tabs (default)
--indentation Tabs

# Two spaces
--indentation TwoSpaces

# Four spaces
--indentation FourSpaces
```

`--lineEndings`

Line ending symbols for generated code.

```
# Windows \\r\\n (default)
--lineEndings Windows

# Unix style \\n
--lineEndings Unix
```

`--splitFiles`

Set this flag to lay out generated classes into separate files. If not set, then one giant file with the name of `--gameDataClassName`.hx will be generated.

`--optimizations` List of enabled optimization in generated code.

```
# Eagerly resolves and validates all references in loaded documents.
# When enabled, this optimization ensures that all references in documents are
# during loading. This comes with a performance cost but guarantees the validit
--optimizations eagerReferenceResolution

# Opts for raw references without generating helper methods for referenced docu
# With this optimization, the generated code will not include helper methods fo
# referenced documents, keeping only accessors that work with raw references.
--optimizations rawReferences

# Avoids generating helper methods for localized strings, keeping only raw acce
# This optimization eliminates helper methods for accessing localized text, ins
# accessors that deal directly with lists of localized texts.
--optimizations rawLocalizedStrings

# Disables string pooling during game data loading.
# Turning off string pooling can yield a minor performance improvement at the c
# memory usage, as it avoids reusing short strings.
--optimizations disableStringPooling

# Disables generation of code for loading game data from JSON formatted files.
# This optimization omits code related to JSON serialization, useful when JSON
# game data is not used.
--optimizations disableJsonSerialization

# Disables generation of code for loading game data from Message Pack formatted
# Similar to DisableJsonSerialization, this option removes code related to load
# from Message Pack formatted files.
--optimizations disableMessagePackSerialization

# Disables generation of code related to applying patches during game data load
# This removes a significant portion of code that is mainly used for modding su
# where patches are applied to game data at runtime.
--optimizations disablePatching

# Disables generation of enums with known document IDs.
# This removes a significant portion of code that contains listings of IDs for
# documents known at the moment of code generation, which improves compilation
--optimizations disableDocumentIdEnums
```

`--clearOutputDirectory`	Clear the output directory from generated files. Generated files are identified by the presence of the '<auto-generated>' tag inside.

This command supports universal parameters.

Export Code Generation Templates (Obsolete)

Exports T4 code generation templates to a specified directory. These templates can be used with Visual Studio, Rider, Visual Studio Code with plugin, `dotnet` tool or other tools to generate source code.

Warning

This command will be removed in future versions. Download templates directly from the repository.

- CLI Installation
- Commands Reference

- Universal Parameters
- URL-based Input/Output

Command

```
# Windows
 charon GENERATE TEMPLATES --outputDirectory "c:\templates"
```

Parameters

`--outputDirectory`	Specifies the path where the templates should be written. It can be either an absolute or relative path to a directory. The specified directory must already be present.
	`# Windows`
	`--outputDirectory "c:\templates"`
	
	`# Linux or OSX`
	`--outputDirectory "~/templates"`
	
	`# Relative path`
	`--outputDirectory "./templates"`

Generate Text from Templates (Obsolete)

Generates text from T4 templates into output directory.

> **Warning**
>
> This command was removed since 2025.1.1 version. It is recommended to use an IDE or open-source alternatives for generating text with T4 templates. See: dotnet-t4

See GENERATE TEMPLATES to get actual T4 templates.

- CLI Installation
- Commands Reference

Generate TypeScript Source Code

Generates TypeScript source code for game data into output directory.

This command does not delete previously generated files, and it is the responsibility of the user to ensure that any previous files are removed before running the command again.

- CLI Installation
- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
charon GENERATE TYPESCRIPTCODE --dataBase "c:\my app\gamedata.json" --outputDirectory "c:\my

# remote game data
charon GENERATE TYPESCRIPTCODE --dataBase "https://charon.live/view/data/My_Game/develop/" -
```

Parameters

`--dataBase` Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\my app\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--credentials` The API key used to access remote server in case of *–dataBase* being URL.

`--outputDirectory` Specifies the path where the source code should be written. It can be either an absolute or relative path to a directory. The specified directory must already be present.

```
# Windows
--outputDirectory "c:\my app\scripts"

# Linux or OSX
--outputDirectory "~/my app/scripts"

# Relative path
--outputDirectory "./my app/scripts"
```

`--documentClassName` Name for base class for all documents.

```
# name (default)
--documentClassName Document

# in case of name collision
--documentClassName GameDataDocument
```

`--gameDataClassName` Name for class containing whole in-memory game data.

```
# name (default)
--gameDataClassName GameData

# in case of name collision
--gameDataClassName MyGameData
```

`--indentation` Indentation style for generated code.

```
# Tabs (default)
--indentation Tabs

# Two spaces
--indentation TwoSpaces

# Four spaces
--indentation FourSpaces
```

`--lineEndings` Line ending symbols for generated code.

```
# Windows \\r\\n (default)
--lineEndings Windows

# Unix style \\n
--lineEndings Unix
```

`--splitFiles` Set this flag to lay out generated classes into separate files. If not set, then one giant file with the name of `--gameDataClassName`.ts will be generated.

`--optimizations` List of enabled optimization in generated code.

```
# Eagerly resolves and validates all references in loaded documents.
# When enabled, this optimization ensures that all references in documents are
# during loading. This comes with a performance cost but guarantees the validit
--optimizations eagerReferenceResolution

# Opts for raw references without generating helper methods for referenced docu
# With this optimization, the generated code will not include helper methods fo
# referenced documents, keeping only accessors that work with raw references.
--optimizations rawReferences

# Avoids generating helper methods for localized strings, keeping only raw acce
# This optimization eliminates helper methods for accessing localized text, ins
# accessors that deal directly with lists of localized texts.
--optimizations rawLocalizedStrings

# Disables string pooling during game data loading.
# Turning off string pooling can yield a minor performance improvement at the c
# memory usage, as it avoids reusing short strings.
--optimizations disableStringPooling

# Disables generation of code for loading game data from JSON formatted files.
# This optimization omits code related to JSON serialization, useful when JSON
# game data is not used.
--optimizations disableJsonSerialization

# Disables generation of code for loading game data from Message Pack formatted
# Similar to DisableJsonSerialization, this option removes code related to load
# from Message Pack formatted files.
--optimizations disableMessagePackSerialization

# Disables generation of code related to applying patches during game data load
# This removes a significant portion of code that is mainly used for modding su
# where patches are applied to game data at runtime.
--optimizations disablePatching

# Disables generation of enums with known document IDs.
# This removes a significant portion of code that contains listings of IDs for
# documents known at the moment of code generation, which improves compilation
--optimizations disableDocumentIdEnums
```

`--clearOutputDirectory`	Clear the output directory from generated files. Generated files are identified by the presence of the '<auto-generated>' tag inside.

This command supports universal parameters.

Generate Unreal Engine C++ Source Code

Generates C++ for Unreal Engine source code for game data into output directory.

This command does not delete previously generated files, and it is the responsibility of the user to ensure that any previous files are removed before running the command again.

- CLI Installation
- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
charon GENERATE UECPPCODE --dataBase "c:\My Project\Content\gamedata.json" --outputDirectory

# remote game data
charon GENERATE UECPPCODE --dataBase "https://charon.live/view/data/My_Game/develop/" --outp
```

Parameters

`--dataBase` Absolute or relative path to game data. Use quotation marks if your path contains spaces.

```
# local file
--dataBase "c:\My Project\Content\gamedata.json"

# remote server
--dataBase "https://charon.live/view/data/My_Game/develop/"
```

`--credentials` The API key used to access remote server in case of *–dataBase* being URL.

`--outputDirectory` Specifies the path where the source code should be written. It can be either an absolute or relative path to a directory. The specified directory must already be present.

```
# Windows
--outputDirectory "c:\My Project\Source\Gamedata"

# Linux or OSX
--outputDirectory "~/My Project/Source/Gamedata"

# Relative path
--outputDirectory "./My Project/Source/Gamedata"
```

`--documentClassName` Name for base class for all documents.

```
# name (default)
--documentClassName Document # became UDocument in generated code

# in case of custom inheritance chain
    # class SHOULD publicly inherit UDocument
--documentClassName GameDataDocument # became UGameDataDocument in ge
```

`--gameDataClassName` Name for class containing whole in-memory game data.

```
# name (default)
--gameDataClassName GameData # became UGameData in generated cod

# in case of name collision
--gameDataClassName MyGameData # became UMyGameData in generated
```

`--defineConstants` Preprocessor constants to define. Use semicolon(;) to separate multiple values.

```
--defineConstants NO_OPTIMIZATIONS;USE_FSTRING_ONLY
```

`--indentation` Indentation style for generated code.

```
# Tabs (default)
--indentation Tabs

# Two spaces
--indentation TwoSpaces

# Four spaces
--indentation FourSpaces
```

`--lineEndings` Line ending symbols for generated code.

```
# Windows \\r\\n (default)
--lineEndings Windows

# Unix style \\n
--lineEndings Unix
```

`--optimizations` List of enabled optimization in generated code.

```
# Eagerly resolves and validates all references in loaded documents.
# When enabled, this optimization ensures that all references in documents
# during loading. This comes with a performance cost but guarantees the va
--optimizations eagerReferenceResolution

# Opts for raw references without generating helper methods for referenced
# With this optimization, the generated code will not include helper metho
# referenced documents, keeping only accessors that work with raw referenc
--optimizations rawReferences

# Avoids generating helper methods for localized strings, keeping only raw
# This optimization eliminates helper methods for accessing localized text
# accessors that deal directly with lists of localized texts.
--optimizations rawLocalizedStrings

# Disables string pooling during game data loading.
# Turning off string pooling can yield a minor performance improvement at
# memory usage, as it avoids reusing short strings.
--optimizations disableStringPooling

# Disables generation of code for loading game data from JSON formatted fi
# This optimization omits code related to JSON serialization, useful when
# game data is not used.
--optimizations disableJsonSerialization

# Disables generation of code for loading game data from Message Pack form
# Similar to DisableJsonSerialization, this option removes code related to
# from Message Pack formatted files.
--optimizations disableMessagePackSerialization

# Disables generation of code related to applying patches during game data
# This removes a significant portion of code that is mainly used for moddi
# where patches are applied to game data at runtime.
--optimizations disablePatching

# Disables generation of enums with known document IDs.
# This removes a significant portion of code that contains listings of IDs
# documents known at the moment of code generation, which improves compila
--optimizations disableDocumentIdEnums
```

`--clearOutputDirectory` Clear the output directory from generated files. Generated files are identified by the presence of the '<auto-generated>' tag inside.

This command supports universal parameters.

Initialize Game Data

Initializes an empty or missing file with initial data. Path to game data should be local file system's file.

- CLI Installation
- Commands Reference

Command

```
# full path (windows)
charon INIT "c:\my app\gamedata.gdjs"

# full path (linux)
charon INIT "/var/mygame/gamedata.json"

# relative path
charon INIT mygame/gamedata.json
```

Parameters

`--fileName`

Absolute or relative path to game data file. Use quotation marks if your path contains spaces.

```
# local file
--fileName "c:\my app\gamedata.json"
```

URL input/output parameters

Some command accept URL as input/output parameter.

Supported URL Schemes

Scheme	Input parameter	Output parameter
HTTP[S]	A GET request will be sent	A POST request with body containing output will be sent
FTP(S)	A RETR command will be sent	A STOR command with output content will be sent
FILE	File will be read	File will be written

Authentication

Authentication data could be passed in *user* part of URL. More advanced authentication schemes are not supported.

Examples

```
# publish data to FTP
charon DATA EXPORT
  --dataBase "https://charon.live/view/data/My_Game/develop/dashboard"
  --output "ftp://user:password@example.com/public/gamedata.json"
  --mode publication
  --outputFormat json
  --credentials "<API-Key>"

# import localization from remote HTTP server
charon DATA I18N IMPORT
  --dataBase "file:///c:/my app/gamedata.json"
  --input "https://example.com/translated/gamedata.xliff"
  --inputFormat xliff

# print languages for game data in local file
charon DATA I18N LANGUAGES --dataBase "file:///c:/my app/gamedata.json"

# print languages for game data in local file relative to current working directory
charon DATA I18N LANGUAGES --dataBase "file:///./gamedata.json"

# print languages for game data at remote server using API Key
export CHARON_API_KEY=87758CC0D7C745D0948F2A8AFE61BC81
charon DATA I18N LANGUAGES --dataBase "https://charon.live/view/data/My_Game/develop/dashboa
```

Start in Standalone Mode

Starts Charon in standalone mode for specified game data. Path to game data could be local file system's file or remote server address.

- CLI Installation
- Commands Reference
- Universal Parameters
- URL-based Input/Output

Command

```
# local game data (windows)
charon SERVER START --dataBase "c:\my app\gamedata.json" --port 8080 --launchDefaultBrowser

# shortcut version
charon "c:\my app\gamedata.json"
```

Parameters

`--dataBase`	Absolute or relative path to game data. Use quotation marks if your path contains spaces.
	`# local file`
	`--dataBase "c:\my app\gamedata.json"`

`--port`	Number of an IP port (1-65535) to be used to listen for browser based UI. `--port 8080`
`--launchDefaultBrowser`	Set this flag to open system-default browser on successful start.
`--denySchemaEditing`	Disable the ability to change the data structure (metadata) for editable game data.

`--resetPreferences`	Set this flag to reset UI preferences on successful start.

This command supports universal parameters.

Universal parameters

All commands accept universal parameters and environment variables.

`--verbose`	Set this flag to get additional diagnostic information in logs.
`--log <path>`	Add additional file logging to the existing logging configuration from `appsettings.json`. `--log "./logs/charon.log"`

`--log out`	Add additional terminal (standard output) logging to the existing logging configuration from `appsettings.json`.
	`--log out`
	`# or`
	`--log con`

`--pause`	Wait for user prompt before the application exits.

Environment variables

In addition to the standard configuration redefinition mechanism using environment variables, the following environment variables are also supported.

CHARON_API_KEY

The **`API key`** which is used to access the remote server. This environment variable is usually used in conjunction with `--dataBase`, which points to a remote server.

```
# Windows
set CHARON_API_KEY=87758CC0D7C745D0948F2A8AFE61BC81

# OSX or Linux
export CHARON_API_KEY=87758CC0D7C745D0948F2A8AFE61BC81
```

Get Charon Version

Gets version of `charon` application.

- CLI Installation
- Commands Reference

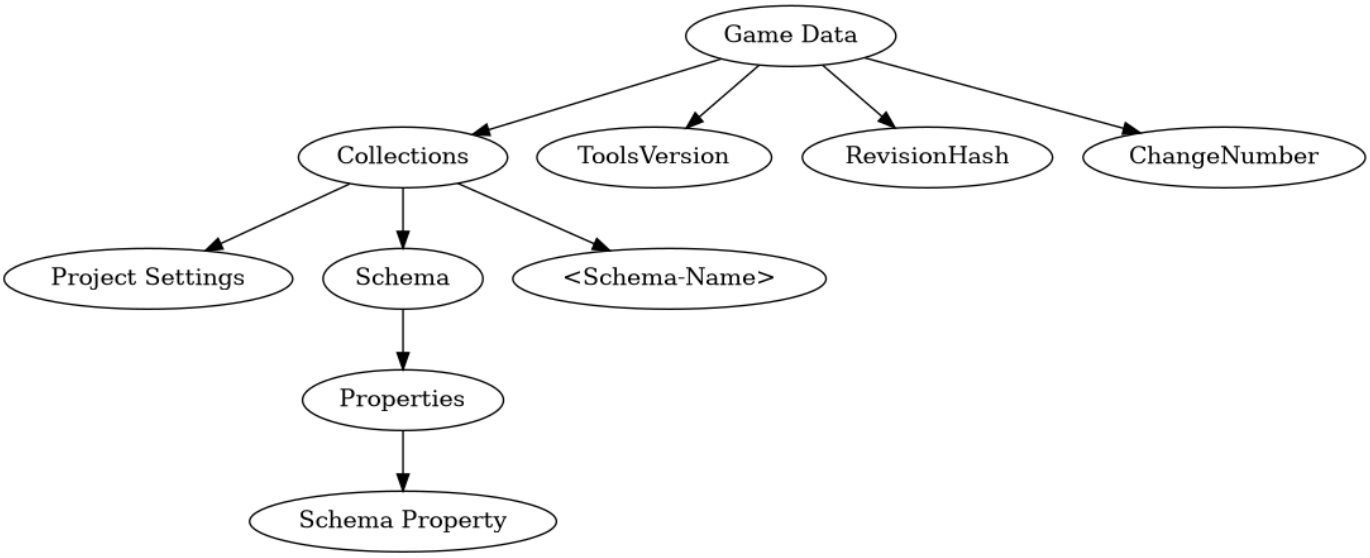
Command

```
# Windows, Linux or OSX
charon VERSION
#> 2023.2.3-alpha
```

Parameters

This command supports universal parameters.

Game Data Structure



Game Data

Fields

- `ToolsVersion` (string): Version of the application used to create this file.
- `RevisionHash` (string): Current changeset hash value.
- `ChangeNumber` (number): Current changeset ordinal number.
- `Collections` (object): List of document collections identified by schema name.
    - `ProjectSettings` (array): Project-related settings for the current file.
        - See Project Settings section below.
    - `Schema` (array): Project-related schemas for the current file.
        - See Schema section below.
    - `<Schema-Name>` (array): Other document collections listed in alphabetical order.

Example

```
{
  "ToolsVersion": "2023.1.2.0",
  "RevisionHash": "678f998993a22d1f54b7fa80",
  "ChangeNumber": 1,
  "Collections":
  {
    "ProjectSettings":
    [
      {
        /* see project settings section below */
      }
    ],
    "Schema":
    [
      {
        /* see schema section below */
      }
    ],
    "SchemaProperty": [ /* always empty */ ],
```

```
    "<Schema-Name>":
    [
      {
        "Id": "<Id>" // All documents have Id

        /* rest of properties of document */
      },
      // ...
    ]
  }
}
```

Project Settings

Fields

- `Id` (string): Unique identifier for the project settings (BSON ObjectId).
- `Name` (string): Name of the project.
- `PrimaryLanguage` (string): Primary language for localizable text in the project (language ID in BCP-47 format).
- `Languages` (string): Alternative languages for localizable text in the project (semicolon-delimited list of language IDs in BCP-47 format).
- `Copyright` (string): Copyright information for the project.
- `Version` (string): Version of the current file, represented as four numbers separated by dots (Major.Minor.Build.Revision).

Example

```
{
  "Id": "049bc0604c363a980b000088",
  "Name": "My Project",
  "PrimaryLanguage": "en-US",
  "Languages": "en-GB;fr-FR",
  "Copyright": "My Company (■) 2025",
  "Version": "1.0.0.0"
}
```

Schema

Fields

- `Id` (string): Unique identifier for the schema (BSON ObjectId).
- `Name` (string): Name of the schema (valid C identifier).
- `DisplayName` (string): Display name of the schema for UI purposes.
- `Description` (string): Schema description used in generated documentation.
- `Specification` (string): Extension data for the schema in *application/x-www-form-urlencoded* format (RFC-1867).
- `IdGenerator` (number): ID generation method for documents created by this schema:
    - *0* : None - ID must be provided manually by the user.
    - *1* : ObjectId - Generates a new BSON ObjectId.
    - *2* : Guid - Generates a new UUID.
    - *3* : Sequence - Uses an incrementing number unique to each schema.
    - *4* : GlobalSequence - Uses an incrementing number shared across all schemas.

- `Type` (number): Schema type:
    - *0* : Normal - Documents can be created in *Collections* or embedded in another document.
    - *1* : Component - Documents are always embedded in another document and never appear in `Collections`.
    - *2* : Settings - Only one document of this schema can exist in `Collections`.
- `Properties` (array): List of schema properties. Always includes the `Id` property.
    - See Schema Property section below.

Example

```
{
  "Id": "592fc86c983a36266c0912a0",
  "Name": "Item",
  "DisplayName": "Items",
  "Type": 0,
  "Description": "An item.",
  "IdGenerator": 1,
  "Specification": "icon=fugue16%2Fabacus&group=Metagame",
  "Properties": [
    // property
  ]
}
```

Schema Property

Structure

- `Id` (string): Unique identifier for the property (BSON ObjectId).
- `Name` (string): Name of the property (valid C identifier).
- `DisplayName` (string): Display name for UI and documentation purposes.
- `Description` (string): Property description used in generated documentation.
- `DataType` (number): Data type of values stored in documents:
    - *0*: Text - Line of text.
    - *1*: LocalizedText - Lines of localized text.
    - *5*: Logical - Boolean value.
    - *8*: Time - Time span.
    - *9*: Date - Specific date.
    - *12* : Number - Decimal number.
    - *13* : Integer - Whole number.
    - *18* : PickList - Predefined value list.
    - *19* : MultiPickList - Multiple selections from predefined values.
    - *22* : Document - Embedded document.
    - *23* : DocumentCollection - Collection of embedded documents.
    - *28* : Reference - Reference to another document.
    - *29* : ReferenceCollection - References to multiple documents.
    - *35* : Formula - C#-like expression for calculations.
- `DefaultValue` (vary|null): Default value for the property used when a new document is created.

- `Uniqueness` (number): Uniqueness requirement for the property:
    - *0* : None - Value does not need to be unique.
    - *1* : Unique - Value must be unique across all documents of this type.
    - *2* : UniqueInCollection - Value must be unique within the containing collection.
- `Requirement` (number): Value requirement for the property:
    - *0* : None - Value is optional and can be null.
    - *1* : <UNUSED> - Remapped to 2 (NotNull) during saving.
    - *2* : NotNull - Value is required but can be an empty string/collection.
    - *3* : NotEmpty - Value is required and cannot be empty.
- `ReferenceType` (object|null): Referenced schema for certain data types (*Document*, *DocumentCollection*, *Reference*, *ReferenceCollection*):
    - `Id` (string): Identifier of the referenced schema.
    - `DisplayName` (string): Optional display name of the referenced schema.
- `Size` (number): Maximum or exact size of the data type. For variable-length types (e.g., text, collections), this defines the size; for others, it is zero.
- `Specification` (string): Extension data for the property in *application/x-www-form-urlencoded* format (RFC-1867).

Example


```
{
  "Id": "592fc9f8983a36266c0912aa",
  "Name": "TextField",
  "DisplayName": "Text Field",
  "Description": "",
  "DataType": 0,
  "DefaultValue": null,
  "Uniqueness": 0,
  "Requirement": 0,
  "ReferenceType": null,
  "Size": 0,
  "Specification": null
}
```


Importing and Exporting Data

Charon supports exporting and importing game data to facilitate workflows such as version control, bulk editing, translation, and integration with external tools or pipelines. This document outlines the general principles of working with exported data, the process of modifying it, and best practices for importing it back into the system.

Overview

Data may be exported from Charon in various formats, edited externally, and re-imported into a project. Data can also be created outside of Charon, provided it conforms to the expected structure. For detailed structural information, refer to Game Data Structure.

Common use cases include:

- Performing batch modifications using tools such as *yq*, spreadsheets, or custom scripts.
- Managing localization and translation workflows.
- Synchronizing game data with external DBs, Tools, Game Engines etc…
- Integrating game data with CI/CD pipelines.

Exporting Data

Data can be exported using either the graphical user interface (GUI) or the command-line interface (CLI). For a full CLI reference, refer to Export via CLI.

Exported data is available in formats such as JSON, BSON, MsgPack, and XLSX. JSON is recommended for most use cases due to its compatibility with standard tooling.

Example (CLI):

```
charon DATA EXPORT --dataBase gamedata.json --schemas Item --output items.json --outputForma
```

Example output (JSON):

```
{
  "Collections": {
    "Item": [
      { "Id": "Sword", "Name": "Iron Sword", "Damage": 3 }
    ]
  }
}
```

Modifying Exported Data

After export, data can be edited manually or via automated tools. Below is an example using *yq* to update a value:

```
yq -i '.Collections.Item[0].Damage = 5' items.json
```

In addition to editing, tools may be used to convert between formats (e.g., JSON to YAML) or to apply structural changes such as renaming fields.

Creating Data from Scratch

Data files can be authored externally and later imported into Charon, provided they conform to the required format. The structure must include a *Collections* object containing arrays of documents for each schema:

```
{
  "Collections": {
    "Item": [
      { "Id": "BronzeSword", "Name": "Bronze Sword", "Damage": 4 }
    ]
  }
}
```

See Game Data Structure for full format specifications.

Importing Data

Data can be imported through both the GUI and CLI. Several import modes are available to support different workflows.

Example (CLI):

```
charon DATA IMPORT --dataBase gamedata.json --schemas Item --input items.json --mode safeUpd
```

For full CLI options, refer to Import via CLI.

Special Considerations

Document Identification (Id Field)

- If an *Id* is not specified, it will be automatically generated.
- If the document is referenced elsewhere in the data, an *Id* must be provided.

- Temporary identifiers of the form *_ID_<UNIQUE>* can be used during import; these will be consistently resolved across all documents.

Partial Updates

When using update modes, documents do not need to include all fields. Partial structures containing only modified or relevant fields are accepted.

```
[
  { "Id": "IronSword", "Damage": 10 }
]
```

Dry Run

It is recommended to use dry run functionality to preview changes before committing them:

- **GUI**: Enable the `Perform a dry run and don't persist changes` option.
- **CLI**: Use the `--dryRun` flag:

```
charon DATA IMPORT --dataBase gamedata.json --schemas Item --input items.json --mode update
```

Validation Options

Validation may be disabled to import incomplete or prototype data:

- **GUI**: Enable the `Ignore consistency errors in imported data` option.
- **CLI**: Use the `--validationOptions None` flag.

```
charon DATA IMPORT --dataBase gamedata.json --schemas Item --input items.json --mode createA
```

> **Note**
>
> When importing schema definitions, validation is always enforced and must pass successfully.

Import Modes

Charon supports multiple import modes depending on the desired outcome:

Mode	Description
createAndUpdate	Default mode. Creates new documents and updates existing.
create	Creates new documents only. Existing ones are untouched.
update	Updates existing documents only. No new documents added.
safeUpdate	Updates only property values. Sub-documents are not added or removed.
replace	Replaces the entire collection with imported documents.
delete	Deletes documents listed in the input file.

All import modes support both full and partial document structures depending on the use case.

See also

- Internationalization (i18n)
- Game Data Structure
- Export via CLI
- Import via CLI

Internationalization (i18n)

Charon supports localization by enabling text data to be stored in multiple languages using the Localized Text data type. This allows game data to be exported for translation, modified externally, and imported back after localization.

Supported export formats include XLSX, XLIFF (XML Localization Interchange File Format) and JSON. These formats are suitable for translation workflows involving external tools or localization teams.

Overview

Localization workflows in Charon rely on identifying fields of type Localized Text and processing them through either spreadsheet or industry-standard interchange formats. Translated data can then be re-integrated using import commands.

Common use cases include:

- Managing multi-language game content.
- Sending text data to third-party localization vendors.
- Automating translation pipelines with CLI tools.

Supported Languages

Translation languages are defined in the **Project Settings**. To view the configured languages via the CLI, run:

```
charon DATA I18N LANGUAGES --dataBase gamedata.json
```

Exporting Translatable Text

Charon provides two primary formats for exporting localizable content: XLSX and XLIFF. Exporting can be done via the graphical interface or command-line interface.

XLSX Export (Spreadsheet)

The following example exports all translatable text to an Excel spreadsheet:

```
charon DATA EXPORT --dataBase "gamedata.json" --properties [LocalizedText] --output "text_al
```

Key parameters:

- `--properties [LocalizedText]`: Filters exported data to include only *LocalizedText* fields.
- `--languages`: (Optional) Specifies which languages to include in the export.

> **Note**
>
> The exported spreadsheet may contain additional metadata columns that are required for correct import.

XLIFF Export (Industry Standard)

To export translation data in XLIFF format:

```
charon DATA I18N EXPORT --dataBase "gamedata.json" --sourceLanguage en --targetLanguage fr -
```

Key parameters:

- `--sourceLanguage`: Language of the original text.
- `--targetLanguage`: Language to which the content will be translated.
- `--outputFormat`: Format of the exported file. Supported values include *xliff*, *xliff1*, and *xliff2*.

Importing Translated Data

After translation, the modified data can be imported back into the project. Both XLSX and XLIFF formats are supported.

XLSX Import

To import translated spreadsheet data:

```
charon DATA IMPORT --dataBase "gamedata.json" --input "text_all_languages.xlsx" --inputForma
```

Key parameters:

- `--inputFormat xlsx`: Indicates the input file format.
- `--mode safeUpdate`: Ensures only existing fields are updated, without creating or deleting data.

XLIFF Import

To import translated XLIFF content:

```
charon DATA I18N IMPORT --dataBase "gamedata.json" --input "en_fr_texts.xliff"
```

Best Practices

- Use **XLSX** when working with translators familiar with spreadsheet tools.
- Use **XLIFF** for integration with professional translation software or localization platforms.
- Validate the translated data with a **dry run** before importing changes into production data.

Unsupported Formats

While Charon may accept other serialization formats (e.g., BSON, MsgPack), compatibility for internationalization workflows is only guaranteed for XLSX, XLIFF and JSON.

Additional Resources

- DATA EXPORT
- DATA IMPORT
- DATA I18N EXPORT
- DATA I18N IMPORT
- DATA I18N LANGUAGES

Working with Logs

Charon creates a log files with various messages that may be useful for troubleshooting and debugging.

Unity Plugin

Log files are saved to `<project-directory>/Library/Charon/logs/`.

Unreal Engine Plugin

Log files are saved to `<project-directory>/Intermediate/Charon/logs/`.

CLI and Standalone

Log files are saved to:

- Windows: `C:/ProgramData/Charon/logs/`.
- MacOS: `/Users/<username>/.config/Charon/logs`
- Linux: `/home/<username>/.config/Charon/logs`

Note: Make sure to replace `<project-directory>` and `<charon-directory>` with the actual directories on your system.

Logging Levels

Normally only the most important events are logged. If you have trouble identifying an issue, you might want to change log to *verbose*. This way more information is included in logs.

Unity Plugin

In menu select `Tools` → `Charon` → `Troubleshooting` → `Verbose Logs`.

CLI and Standalone

Launch with `--verbose` parameter.

Then repeat the action that causes the bug (or the one you want analyzed anyway) and check log file again.

CLI Example:

```
charon SERVER START ./gamedata.json --launchDefaultBrowser --verbose
```

Resetting UI Preferences

If for some reason editor behaves erratically (grids aren't displayed correctly or aren't displayed at all), you can restore default UI settings.

Unity plugin

Select in menu `Tools` → `Charon` → `Troubleshooting` → `Reset Preferences`.

CLI and Standalone

Launch with `--resetPreferences.` parameter.

Web

Use the `Preferences` profile tab `<User Icon>` → `Profile` → `Preferences`.

CLI Example:

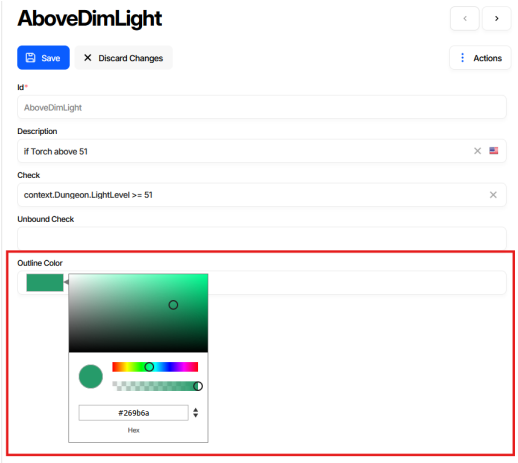
```
charon SERVER START ./gamedata.json --launchDefaultBrowser --resetPreferences
```

UI Extensions

Charon supports a powerful plugin model through **extensions**, which allow developers to create custom field and document editors that integrate with the application. These extensions are built using **Web Components** and packaged as **NPM modules** for easy distribution and reuse.

What Is a Charon Extension?

A **Charon extension** is an external package that provides one or more custom UI components (editors) used inside the document form.



Each extension is:

- A **Web Component** implementing a specific interface.
- Packaged as a standard **NPM module**.
- Discoverable by Charon at runtime (via local `.tgz` or a public NPM registry).
- Described with metadata in `package.json` under the `config.customEditors` section.

What Is a Web Component?

A **Web Component** is a browser-native way to define reusable UI elements that are:

- **Encapsulated**: HTML, CSS, and JS are scoped to the component.
- **Self-contained**: No dependencies on the application's framework.
- **Reusable**: Can be shared across projects regardless of frontend tech (React, Angular, Vue, etc.).

This isolation is essential for Charon, which must load unknown user-defined components without conflict or side effects.

What Is an NPM Package?

An **NPM package** is a collection of files published to the npm registry or used locally. It contains:

- JavaScript bundles
- Stylesheets (optional)
- Metadata (`package.json`)
- Documentation or type definitions

In Charon, an extension is published as an NPM package (or `.tgz` archive) and loaded dynamically at runtime.

Typical NPM Package Structure

A Charon extension NPM package usually contains:

```
dist/
■■■ index.js              # Compiled JS bundle that registers the component
■■■ assets/
■   ■■■ index.css         # Optional styles
■■■ package.json          # Includes Charon metadata and schema reference
■■■ *.tgz                 # (optional) Published archive
```

Example `package.json` metadata:

```
"config": {
  "customEditors": [
    {
      "id": "ext-logical-toggle",
      "selector": "ext-logical-toggle-editor",
      "name": "Logical Toggle",
      "type": ["Property", "Grid"],
      "dataTypes": ["Logical"]
    }
  ]
}
```

Gettings Started

- **Creating a Custom Editor with React**
- **Creating a Custom Editor with Angular**
- **Implementing Custom Property Editor Interface**

Publication and Consumption

After you implement the extension interface and build the UI logic to display and edit the value, you need to create a production build of your package and publish it either locally or to the NPM registry.

To publish locally:

```
npm pack
```

This creates a `.tgz` archive in your `dist` directory that can be manually placed in Charon's extension folder for testing.

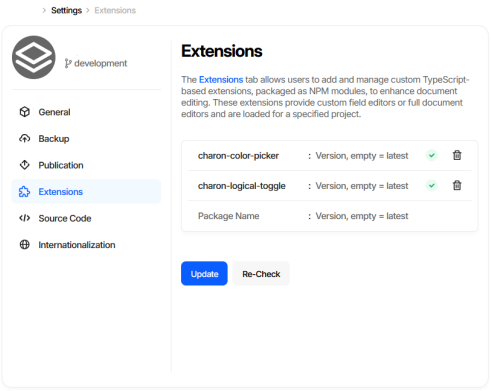
To publish publicly:

```
npm publish
```

Make sure your `package.json` contains all required metadata and the `"private": true` flag is removed.

Once published, the extension package can be added through the `Project Settings` → `Extensions` tab in Charon. Enter the NPM package name and, optionally, specify a version. If the version is omitted, the latest published version will be used.

The list of enabled extensions is saved inside your Charon data file, making it automatically available to all users working on that file.



Note

Locally installed extensions (via `.tgz` files) are intended only for development and debugging. They are available **only on your machine** and will not be shared with other users.

See Also

- **`Creating a Custom Editor with React`**
- **`Creating a Custom Editor with Angular`**
- Example Projects (GitHub)

Frequently Asked Questions (FAQ)

What kind of game data can be modeled in Charon?

Charon is designed for modeling static game data. Typical use cases include Quests, Dialogs, Levels, Loot Tables, Items, Monsters, Abilities, Units, Research Trees, Weapons, Spells, Goods, Buildings, and similar structured data used to drive gameplay logic and content.

What data types are supported in Charon?

Charon provides a broad set of data types to suit a variety of needs. These include primitive types such as text, localized text, booleans, time, dates, numbers, enums (pick lists), and flags. More advanced structures are also supported, including embedded documents, collections, references to other documents, and computed formulas.

Is schema inheritance supported?

Yes, schema inheritance is supported in Charon, although it is implemented using a composition-based approach rather than traditional inheritance. Existing schemas can be extended by including them as `Document` properties within other schemas. Read more about inheritance.

Is Charon free to use?

The offline version, including the command-line interface (CLI) and associated plugins, is free to use and distributed under a permissive license. It imposes no restrictions and may be bundled with games to support modding workflows.

The online version, which enables collaborative editing of shared game data in teams, is available under a subscription model.

Are there limitations in the Unity/Unreal plugins or CLI version?

No, there are no imposed limitations on the CLI version or the Unity and Unreal Engine plugins. This includes file size, number of documents, or schema complexity.

Does Charon support localization?

Yes, Charon supports multi-language content through the localized text type. Localizable text can be exported for translation and imported back once translated. In-game, language switching is supported globally with a single function call, ensuring a consistent localization experience.

How does Charon integrate with Unity or Unreal Engine?

Charon provides dedicated plugins for both Unity and Unreal Engine. These plugins integrate natively into each engine's workflow and expose game data as strongly typed assets. Full source code is included, along with example projects demonstrating how to extend and use the plugins within game code.

Can I define relationships between documents (e.g., parent-child or references)?

Yes. Charon supports embedding documents within each other to model hierarchical structures (e.g., trees). It also allows referencing other documents through reference fields or reference collections. These references can be automatically validated for consistency, avoiding the common pitfalls of spreadsheet-based data modeling.

Can I generate custom editors or tools with Charon?

Yes. Charon is extensible through custom field and document editors. It can also be embedded in other applications using an embedded browser such as CEF. For example, the Unreal Engine plugin demonstrates how to run the Charon UI inside an in-editor browser, allowing you to build a seamless experience around your tools.

Does Charon support formulas or computed fields?

Yes. Charon includes support for C#-like formulas as a first-class data type. These formulas can be evaluated at runtime with user-defined parameters, allowing you to move balance logic and other calculations out of compiled code and into editable data structures.

Can Charon be integrated into CI/CD pipelines or custom toolchains?

Yes. Charon includes a command-line interface (CLI) that supports full CRUD operations and data import/export. The CLI can work with local game data stored in JSON or MessagePack formats, as well as connect to the online service. You can script uploads, downloads, merges, and validation steps as part of your CI/CD workflow.

What export formats does Charon support?

Charon supports multiple export and import formats for game data, including JSON, MessagePack, BSON, and XLSX (spreadsheets). For localization, the XLIFF format is also supported. Runtime and asset storage formats are limited to JSON and MessagePack.

Can I control user access and editing permissions?

Yes. In collaborative environments using the online version of Charon, a role-based access model is available. Roles include Administrator, Schema Designer, Editor, and Viewer, each with a defined set of permissions. For more information, see Roles and Permissions.

Is the editor's source code available?

The source code for the editor itself is not publicly available. However, the plugin source code is available on GitHub: Charon GitHub Repositories

Charon can be extended with custom field editors or even full document editors, such as node-based interfaces.

For teams requiring a fully private, self-contained collaborative workspace, an **on-premises installation** of the online service is available. It provides the same features as the hosted version — real-time collaboration, access control, multiple projects, AI features, Machine Translation, and team workflows — while keeping all data and infrastructure under your control. If you are interested in an on-premises deployment, please contact us support@gamedevware.com for licensing and installation options.

What happens if the project is discontinued or no longer maintained?

In the unlikely event that Charon is no longer maintained or all original contributors become unavailable, a contingency plan is in place. The online service domain would eventually become inaccessible; however, the full source code of the server, as well as private Docker images, will be published under the MIT license by a designated successor. This ensures that the system can be self-hosted or forked by the community.

The CLI and engine plugins rely on published NuGet packages, which are already public and will remain unaffected. Users working in offline mode will experience no disruption. For users of the online version, all project data is periodically cloned and stored locally. These local copies remain fully compatible with the standalone application, allowing teams to continue working without interruption.

Glossary

Game Data

The static information for the game, such as items, quests, dialogues, etc., is stored as game data. Schemas are also included to organize and structure game data.

Schema

A schema is a description of the structure for documents in game data. It defines the properties and structure of each document in the game data.

Schema Property

A part of a schema that defines a specific property or attribute of a document in the game data.

Formula

A property data type in a schema for a C# expression that can be executed at runtime to calculate a value for a field.

Reference

A property data type in a schema for a pointer to another document.

Document

A specific instance of a schema in the game data. It represents a single item or entity in the game, such as an item, quest, or dialogue.

Field

A named part of a document that holds a specific value.

Source Code

The code generated by Charon that represents the game data. This code can be used to load the game data at runtime.

Metadata

All the schemas and relations between them. This data is used by Charon to generate the source code for the game data.

Workspace

In the web application, the workspace is the place where users can manage their projects and subscription.

Project

In the web application, a project is a container for organizing related game data. It can contain multiple branches.

Branch

In the web application, a branch is a specific variant of game data within a project. It can be used to manage different versions or stages of the game data.

API Key

A unique identifier generated for a user in the User's Profile "API Keys" section, which can be used to access the REST API and CLI for various operations. It allows for automation of game build processes, such as pushing game data to local GIT repositories.

Publication

The process of exporting game data in a format that can be loaded into the game.