> We recommend using the online version of the documentation. This offline version may not be updated.

DH Save System Documentation for Unity | OFFLINE

Introduction

Welcome to the DH Save System documentation. This guide provides all the information needed to integrate and utilize the DH Save System within your Unity projects. Designed to offer a robust and secure method for saving and loading game data, this system supports a wide variety of data types and ensures the security of your data through encryption.

1. Overview

The DH Save System distinguishes between global data (e.g., game settings) and slot-specific data (e.g., player progress) to provide flexible and secure data management. With built-in encryption and easy-to-use interfaces, it's an essential tool for Unity developers looking to implement reliable save and load functionality.

2. Configuration

SaveSystemSettings: Accessible via **Project Settings**, this component allows for the configuration of encryption keys, auto-saving preferences, and key history management.

Save Data Window: A Unity Editor tool for viewing saved data directly, enhancing the development and debugging process.

3. SaveSystemManager

A runtime component that loads saved data upon game start. It supports custom converters for non-serializable classes, ensuring that all game data can be efficiently saved and loaded.

4. Basic Operations

Saving Data

Illustrates how to save both global and slot-specific data, with examples for various data types, including dictionaries and transforms.

Loading Data

Guides on retrieving saved data, with an emphasis on type safety and handling optional data.

5. Encryption

Details the encryption mechanism used to secure saved data, including setting up encryption keys and managing key history for data recovery.

6. SaveDataSlot Class

The `SaveDataSlot` class manages individual game save slots within the DH Save System, encapsulating slot details like name, save timestamp, progress, and custom data. It simplifies save slot manipulation and querying, offering a robust framework for game save management.

Properties

- **SlotName, LastSave, Progress, IndexSlot, SavePath, OtherData**: Essential slot details including custom data storage.

Methods

- **SetSlotName, SetProgress, SetOtherData<T>, GetOtherData<T>**: Update slot information and manage custom data, supporting flexible data types for comprehensive save management.

7. SaveSystem Class

The `SaveSystem` class is the core of the DH Save System, providing essential functionalities for saving and loading data. It facilitates access to both global data and individual save slots, ensuring secure and efficient data management across your game.

Properties

- **GetCurrentSaveSlot, GetSaveDataSlots, GetGlobalData, GetSlotData:** Accessors for retrieving current slots, all slots, and their respective data.

Methods

- Comprehensive coverage of all methods, from `Save` and `Load` to more specialized functions like `LoadTransform` and `Serialize`.

8. Exceptions

Explains common exceptions (e.g., `KeyAlreadyExistsException`, `SaveSlotNotLoadedException`) and provides strategies for resolution, ensuring robust error handling.

9. Working with Serializable Types

SerializableDictionary and SerializableTransform

Demonstrates converting complex data types into serializable formats for saving, and vice versa for loading, complete with examples.

Contact & Support

For support, bug reports, or contributions, please contact via email: `support@hurtaweb.cz`. We welcome your feedback and contributions to the project.

Overview

The DH Save System allows for the secure saving and loading of game data within Unity, distinguishing between global data and slot-specific data. Global data typically includes game settings like resolution, fullscreen preference, volume, etc., while slot data contains gameplay-related information such as player scores, lives, and positions. Encryption is utilized to secure data, with encryption keys managed in `SaveSystemSettings`.

Licenses

This project includes the following software components, each under its own license:

DH Save System

Newtonsoft.Json

Unity.Mathematics

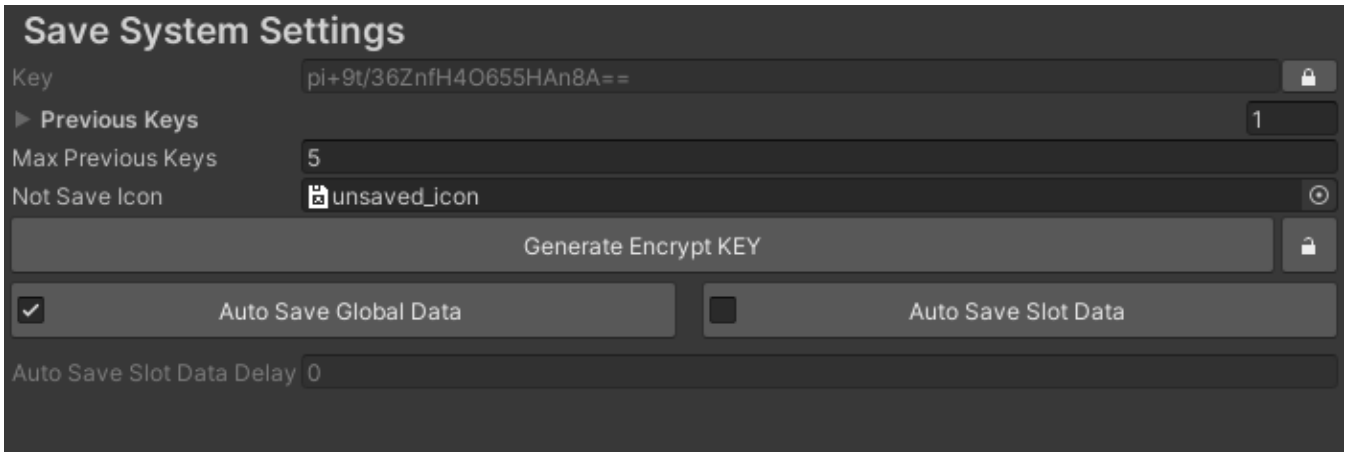
Configuration

SaveSystemSettings

Located under **Project Settings/Save System Settings**, this component is crucial for setting up your encryption key and managing save data preferences.

Key Features:

- **Generate Encrypt Key**: Click to generate a new encryption key, which is immediately set as the current key.
- **Key History**: Maintains a history of used keys (default is 5) to recover from accidental changes.
- **Auto Save Global Data**: Automatically saves global data upon any addition or modification.
- **Auto Save Slot Data**: Enables automatic saving for the currently loaded slot, with a customizable delay (`Auto Save Slot Data Delay Time` in minutes).

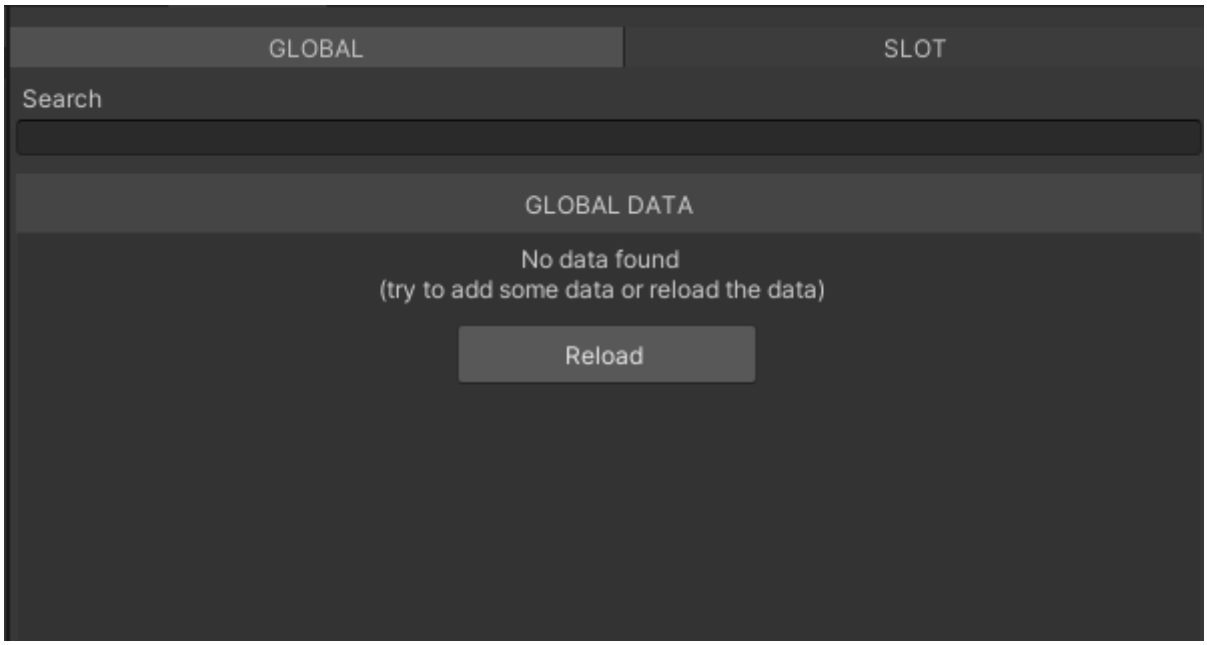


Save Data Window

Accessible via Tools/Save Data, this utility provides a visual interface for managing save data directly in the Unity Editor.

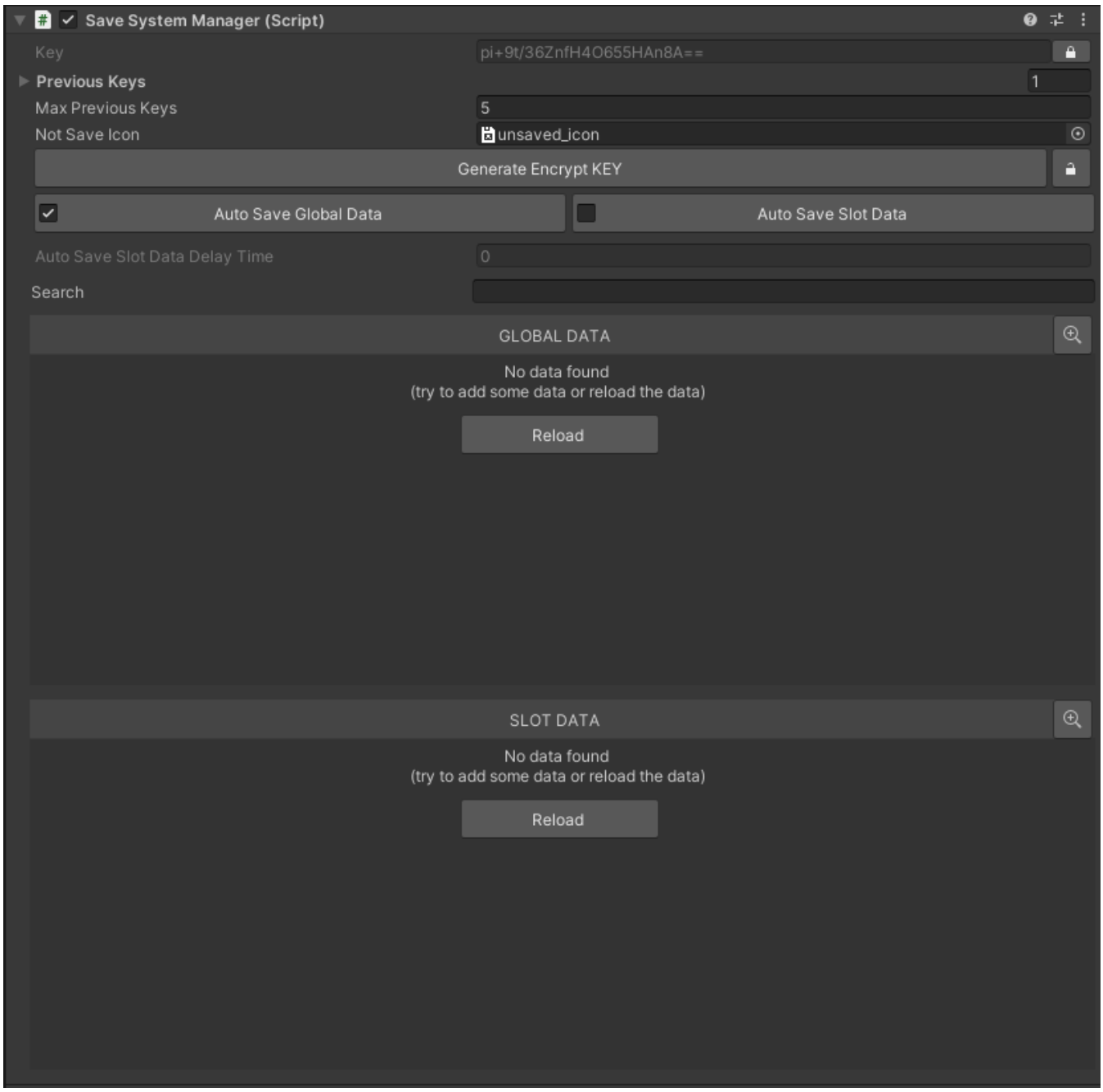
Capabilities:

- **SaveSystemManager Integration**: Automatically creates a `SaveSystemManager` in the current scene if not present, allowing for runtime data management.
- **Data Viewing and Editing**: Separate sections for **GLOBAL** and **SLOT** data, including a search function by ID.
- **Slot Data Management**: Offers the ability to browse through slots or view the current slot's data in the editor.


GLOBAL
SLOT
Search
GLOBAL DATA
No data found
(try to add some data or reload the data)
Reload

SaveSystemManager

A script that facilitates the loading of saved data when the game starts. It also allows for the addition of custom converters for non-serializable classes through the `jsonConverters` list.



Custom Converters

Add your custom converter that inherits from `JsonConverter` to the `jsonConverters` list for handling specific serialization needs.

Basic Operations

Saving Data

Global Data Example:

```
// Save resolution settings as global data
SaveSystem.Save("resolution", new Vector2(1920, 1080), isGlobal: true);
```

Slot Data Example:

```
// Save player score in the current slot
SaveSystem.Save("playerScore", 5000, isGlobal: false);
```

Loading Data

Global Data Example:

```
// Load resolution settings
Vector2 resolution = SaveSystem.Load<Vector2>("resolution", isGlobal: true);
```

Slot Data Example:

```
// Load player score from the current slot
int score = SaveSystem.Load<int>("playerScore", isGlobal: false);
```

Encryption

Ensure the encryption key is correctly set in `SaveSystemSettings` for secure data saving and loading. Use the Generate Encrypt Key button for easy key management and apply auto-saving settings as needed for both global and slot data.

SaveDataSlot Class

The `SaveDataSlot` class represents an individual save slot in your game, encapsulating details such as the slot's name, save progress, and additional custom data. It is designed to work seamlessly with the DH Save System, allowing for the creation, modification, and management of save slots without directly manipulating save files.

Properties

- **SlotName**: The name of the save slot.
- **LastSave**: Timestamp of the last save action.
- **Progress**: A float value indicating the player's progress.
- **IndexSlot**: The index of the slot.
- **SavePath**: The file path where the slot data is saved.
- **OtherData**: Custom data that developers can store related to the save slot.

Methods

`SetSlotName`

Updates the name of the save slot.

Example Usage:

```
saveDataSlot.SetSlotName("NewSlotName");
```

`SetProgress`

Updates the progress value of the save slot.

Example Usage:

```
saveDataSlot.SetProgress(0.5f); // 50% progress
```

SetOtherData<T>

Sets custom data of any type to the save slot and optionally saves this data immediately.

Parameters

- **newOtherData**: The custom data to set.
- **saveImmediately**: If true, the data is saved immediately to disk.

Example Usage

```
// Assuming CustomData is a custom class you want to save as other data
CustomData customData = new CustomData { /* initialization */ };
saveDataSlot.SetOtherData(customData, saveImmediately: true);
```

This method serializes the `newOtherData` into JSON, stores it in the save slot, and optionally saves the global data immediately if `saveImmediately` is set to true.

GetOtherData<T>

Retrieves the custom "other data" stored in the save slot, if any, and deserializes it back into the original data type.

Returns

- The retrieved data of type T, or the default value for type T if not found.

Example Usage

```
CustomData customData = saveDataSlot.GetOtherData<CustomData>();
```

This method deserializes the JSON stored in the `otherData` field of the save slot back into the specified type `T`, allowing for easy access to custom data stored in each save slot.

Working with SaveDataSlot

Save slots are managed through the global data structure and are accessible via the Save Data Window under the name "**SLOTS**". It's crucial not to use "**SLOTS**" as a key for your own save data to avoid conflicts.

Creating and Removing Save Slots

Save slots are created and removed through the `SaveSystem` class, using methods such as `CreateNewSaveSlot` and `RemoveSaveSlot`. The `SavePath` is automatically generated, and the `LastSave` timestamp is updated upon calling `SaveSystem.SaveData()`, which saves the slot and its current state.

Example: Creating a New Save Slot

```
SaveSystem.CreateNewSaveSlot("Slot1", indexSlot: 1);
```

Example: Removing a Save Slot

```
SaveSystem.RemoveSaveSlot("Slot1", indexSlot: 1);
```

Note on Data Access

Access to `SaveDataSlot` properties is straightforward, utilizing public properties without the need for serialization/deserialization methods. This makes retrieving data like progress, slot name, and last save time efficient and direct, streamlining the process of working with save slots in your game.

SaveSystem Class

The `SaveSystem` class provides static functions for saving and loading game data, managing save slots, and handling global and slot-specific data. It also includes methods for data encryption, checking for unsaved changes, and clearing data.

Properties

`GetCurrentSaveSlot`

Returns the currently loaded save slot.

Example Usage:

```
var currentSlot = SaveSystem.GetCurrentSaveSlot;
Debug.Log("Current Save Slot: " + currentSlot.SlotName);
```

`GetSaveDataSlots`

Retrieves all created save slots.

Example Usage:

```
var allSlots = SaveSystem.GetSaveDataSlots;
foreach(var slot in allSlots) {
    Debug.Log("Slot Name: " + slot.SlotName);
}
```

`GetGlobalData`

Returns a dictionary containing all global data.

Example Usage:

```
var globalData = SaveSystem.GetGlobalData;
foreach(var entry in globalData) {
    Debug.Log($"Key: {entry.Key}, Value: {entry.Value}");
}
```

GetSlotData

Returns a dictionary with the current slot's data.

Example Usage:

```
var slotData = SaveSystem.GetSlotData;
foreach(var data in slotData) {
    Debug.Log($"Key: {data.Key}, Value: {data.Value}");
}
```

AreThereUnsavedSlotData

Indicates whether there is unsaved data for the current slot.

Example Usage:

```
bool unsavedData = SaveSystem.AreThereUnsavedSlotData;
Debug.Log("Unsaved Slot Data: " + unsavedData);
```

AreThereUnsavedGlobalData

Indicates whether there is unsaved global data.

Example Usage:

```
bool unsavedGlobalData = SaveSystem.AreThereUnsavedGlobalData;
Debug.Log("Unsaved Global Data: " + unsavedGlobalData);
```

IsSlotLoaded

Checks if a save slot is currently loaded.

Example Usage:

```
bool isLoaded = SaveSystem.IsSlotLoaded;
Debug.Log("Is Slot Loaded: " + isLoaded);
```

Methods

Save<T>

Saves a value to global or slot data.

Parameters

- `ID`: Unique identifier for the data entry.
- `value`: Value to save.
- `isGlobal`: True for global data, false for slot-specific data.

Example Usage

```
SaveSystem.Save("playerHealth", 100, isGlobal: false);
```

Load<T>

Loads a value of type T using the specified ID from either global or slot-specific data.

Parameters

- `ID`: The unique identifier for the data to load.
- `isGlobal`: True for global data, false for slot-specific data.

Example Usage

```
int playerHealth = SaveSystem.Load<int>("playerHealth", isGlobal: false);
```

TryLoad<T>

Attempts to load data of a specified type using the given ID from either global or slot-specific data.

Parameters

- `returnData`: Output parameter for the loaded data.
- `ID`: The unique identifier for the data to load.
- `isGlobal`: True for global data, false for slot-specific data.

Example Usage

```
bool success = SaveSystem.TryLoad(out int playerHealth, "playerHealth", isGlobal:
false);
```

`LoadTransform`

Loads and applies transform data to a single target transform.

Parameters

- `targetTransform`: The target transform to apply data to.
- `ID`: The unique identifier for the data.
- `isGlobal`: True for global data, false for slot-specific data.

Example Usage

```
SaveSystem.LoadTransform(player.transform, "playerPosition", isGlobal: false);
```

`Update<T>`

Updates existing data with the specified ID in either global or slot-specific storage.

Parameters

- `ID`: The unique identifier for the data entry.
- `value`: The new value to store.
- `isGlobal`: True to update global data, false for slot-specific data.

Example Usage

```
SaveSystem.Update("playerHealth", 150, isGlobal: false);
```

`UpdateOrSave<T>`

Updates an existing entry with the specified ID or saves a new entry if it does not exist.

Parameters

- `ID`: The unique identifier for the data entry.
- `value`: The value to update or save.
- `isGlobal`: True to operate on global data, false for slot-specific data.

Example Usage

```
SaveSystem.UpdateOrSave("playerScore", 5000, isGlobal: false);
```

`Remove`

Removes a data entry with the specified ID from either global or slot-specific storage.

Parameters

- `ID`: The unique identifier for the data entry to be removed.
- `isGlobal`: True to remove from global data, false for slot-specific data.

Example Usage

```
SaveSystem.Remove("playerHealth", isGlobal: false);
```

`ClearData`

Clears all data within a specified scope (global or slot-specific) without removing the slots themselves. Optionally saves the changes immediately.

Parameters

- `isGlobal`: Determines whether to clear global data or data specific to the current slot.
- `saveImmediately`: If true, the changes are saved immediately after clearing the data.

Example Usage:

```
SaveSystem.ClearData(isGlobal: true, saveImmediately: true);
Debug.Log("Global data cleared and saved.");
```

`CreateNewSaveSlot`

Creates a new save slot with the specified name and index, and saves it.

Parameters

- `slotName`: The name of the new save slot.
- `indexSlot`: The index of the new save slot.

Example Usage:

```
SaveSystem.CreateNewSaveSlot("New Slot", 1);
Debug.Log("New save slot created.");
```

`LoadSaveSlot`

Loads the save slot with the specified name into memory, replacing any currently loaded data.

Parameters

- `slotName`: The name of the save slot to load.

Example Usage:

```
try {
    SaveSystem.LoadSaveSlot("Slot Name");
    Debug.Log("Save slot loaded.");
} catch(SaveSlotNotLoadedException ex) {
```

```
    Debug.LogError("Error loading slot: " + ex.Message);
}
```

`RemoveSaveSlot`

Removes a save slot with the specified name and index, deletes its associated save file, and updates global data accordingly.

Parameters

- `slotName`: The name of the save slot to remove.
- `indexSlot`: The index of the save slot to remove.

Example Usage:

```
bool removed = SaveSystem.RemoveSaveSlot("Slot to Remove", 0);
Debug.Log("Save slot removed: " + removed);
```

`SaveData`

Saves the current slot data to disk and updates the global save data.

Example Usage:

```
try {
    SaveSystem.SaveData();
    Debug.Log("Data saved successfully.");
} catch(SaveSlotNotLoadedException ex) {
    Debug.LogError("Error saving data: " + ex.Message);
}
```

`LoadData`

Loads data from a 'specified path and optionally applies provided JSON converters for custom deserialization.

Parameters

- `jsonConverters`: (Optional) A list of `JsonConverter` instances for custom serialization settings.

Example Usage:

```
List<JsonConverter> converters = new List<JsonConverter> { new
MyCustomConverter() };
SaveSystem.LoadData(converters);
Debug.Log("Data loaded with custom converters.");
```

`SaveGlobalData`

Saves global data, including information about all save slots, to disk.

Example Usage

```
SaveSystem.SaveGlobalData();
```

`Contains`

Checks if the specified ID exists in either global or slot-specific data.

Parameters

- `ID`: The unique identifier to check.
- `isGlobal`: True to check in global data, false for slot-specific data.

Example Usage

```
bool exists = SaveSystem.Contains("playerHealth", isGlobal: false);
```

`Deserialize<T>`

Deserializes JSON data into an object of the specified type.

Parameters

- `json`: The encrypted JSON string to deserialize.

Example Usage

```
string jsonData = "{ \"playerHealth\": 100 }"; // Encrypted JSON string
int playerHealth = SaveSystem.Deserialize<int>(jsonData);
```

`Serialize`

Serializes an object into a JSON string and encrypts it.

Parameters

- `value`: The object to serialize.

Example Usage

```
PlayerData playerData = new PlayerData { Health = 100, Score = 5000 };
string encryptedJsonData = SaveSystem.Serialize(playerData);
```

Exceptions

When working with the **DH Save System**, you may encounter several exceptions. These are typically easy to resolve and often involve either attempting to save data with an existing ID or trying to save to a slot that hasn't been loaded. Here's how you can address the issues related to these exceptions:

`MissingMethodException`

This exception occurs when a class that is being saved does not have a default constructor. The DH Save System requires a parameterless constructor to instantiate objects during deserialization.

Solution

Ensure that any class you intend to serialize with the DH Save System includes a default, parameterless constructor. If your class has constructors that take parameters, you must still provide a default constructor.

Example

```
public class PlayerData
{
    public int score;
    public string name;

    // Default constructor
    public PlayerData()
    {
    }

    // Constructor with parameters
    public PlayerData(string name, int score)
    {
        this.name = name;
        this.score = score;
    }
}

try
{
    PlayerData playerData = new PlayerData("John Doe", 1000);
    SaveSystem.Save("playerData", playerData, isGlobal: false);
}
```

```
catch (MissingMethodException)
{
    // Ensure PlayerData class includes a default constructor
    Console.WriteLine("Ensure the PlayerData class has a default constructor.");
}
```

Adding a default constructor allows the serialization and deserialization processes to function correctly, avoiding the `MissingMethodException` and ensuring smooth operation of the DH Save System.

KeyAlreadyExistsException

Occurs when attempting to save data with an ID that already exists within the specified scope (global or slot-specific).

Solution

Change the ID you're using for saving. Ensure it's unique to avoid conflicts.

Example

```
try
{
    SaveSystem.Save("playerScore", 5000, isGlobal: false);
}
catch (KeyAlreadyExistsException)
{
    // Change the ID and try again
    SaveSystem.Save("playerScore1", 5000, isGlobal: false);
}
```

SaveSlotNotLoadedException

Thrown when attempting to save to a slot that hasn't been loaded. This can occur if you try to save slot-specific data before a slot is active.

Solution

Before saving data to a slot, ensure that a slot is loaded. If you need to save data but no slot is loaded, either save it as global data or wait until a slot has been loaded.

Example

```
try
{
    SaveSystem.Save("playerLocation", new Vector3(0, 0, 0), isGlobal: false);
}
catch (SaveSlotNotLoadedException)
{
    // Load a slot or save as global data instead
    SaveSystem.LoadSaveSlot("SlotName");
    SaveSystem.Save("playerLocation", new Vector3(0, 0, 0), isGlobal: false);
}
```

`InvalidFieldTypeException`, `InvalidPropertyTypeException`

These exceptions occur when saving custom classes that use unsupported types, such as `Dictionary` or `Transform`, without converting them to a serializable form.

Solution

Convert unsupported types to their serializable counterparts provided by the save system, such as changing `Transform` to `SerializableTransform` and `Dictionary<TKey, TValue>` to `SerializableDictionary<TKey, TValue>`.

Example

```
try
{
    var playerData = new CustomPlayerData
    {
        playerTransform = new SerializableTransform(player.transform),
        inventoryItems = new SerializableDictionary<string, int>(inventoryItems)
    };
    SaveSystem.Save("playerData", playerData, isGlobal: false);
}
catch (InvalidFieldTypeException)
{
    // Ensure all fields are using serializable types
}
catch (InvalidPropertyTypeException)
{
```

```
    // Ensure all properties are using serializable types
}
```

Working with Serializable Types

`SerializableDictionary`

To save a `Dictionary` type **that is in custom class**, it must be stored using `SerializableDictionary`. Upon loading a `SerializableDictionary`, you can easily convert it back to a standard `Dictionary` type using the `.ToDictionary()` method.
When you saving just **simple** `Dictionary`, you can use standard `Dictionary` type.

Usage Example

Saving a `Dictionary`

```
// Assuming you have a Dictionary you wish to save
Dictionary<string, int> playerScores = new Dictionary<string, int>
{
    { "player1", 100 },
    { "player2", 200 }
};

// Convert to SerializableDictionary and save
SerializableDictionary<string, int> serializableScores = new
SerializableDictionary<string, int>(playerScores);
SaveSystem.Save("playerScores", serializableScores, isGlobal: true);
```

Loading a `Dictionary`

```
// Load the SerializableDictionary
SerializableDictionary<string, int> loadedScores =
SaveSystem.Load<SerializableDictionary<string, int>>("playerScores", isGlobal:
true);

// Convert back to Dictionary
Dictionary<string, int> playerScores = loadedScores.ToDictionary();
```

SerializableTransform

For saving `Transform` properties, use `SerializableTransform`. Unlike directly setting a transform, `SerializableTransform` requires you to apply the stored values to a target transform using the `.ApplyToTransform(Transform)` method.

Usage Example

Saving a `Transform`

```
// Create a SerializableTransform from a GameObject's transform
SerializableTransform serializableTransform = new
SerializableTransform(playerGameObject.transform);

// Save the SerializableTransform
SaveSystem.Save("playerTransform", serializableTransform, isGlobal: false);
```

Loading and Applying a `Transform`

```
// Load the SerializableTransform
SerializableTransform loadedTransform =
SaveSystem.Load<SerializableTransform>("playerTransform", isGlobal: false);

// Apply the loaded transform values to a GameObject's transform
loadedTransform.ApplyToTransform(playerGameObject.transform);
```

SerializableFloat2

For saving 2D vector types (such as `Float2` from a mathematics library), use `SerializableFloat2`. This type ensures that 2D vector data can be serialized and deserialized properly.

When `SerializableFloat2` **is used within a custom class, it must** be utilized to ensure serialization compatibility. It includes a method, `.ToFloat2()`, to convert back to the original vector type.

Usage Example

Saving a `Float2`

```
// Assuming you have a Float2 representing player position
Float2 playerPosition = new Float2(1.5f, 2.5f);
```

```
// Convert to SerializableFloat2 and save
SerializableFloat2 serializablePosition = new SerializableFloat2(playerPosition);
SaveSystem.Save("playerPosition", serializablePosition, isGlobal: true);
```

Loading a `Float2`

```
// Load the SerializableFloat2
SerializableFloat2 loadedPosition =
SaveSystem.Load<SerializableFloat2>("playerPosition", isGlobal: true);

// Convert back to Float2
Float2 playerPosition = loadedPosition.ToFloat2();
```

`SerializableFloat3`

For saving 3D vector types (such as `Float3` from a mathematics library), use `SerializableFloat3`. This type is necessary for the serialization and deserialization of 3D vector data.

Similar to `SerializableFloat2`, when `SerializableFloat3` is part of a custom class, it must be used to maintain serialization compatibility. It provides a `.ToFloat3()` method for converting the serialized data back into its original vector form.

Usage Example

Saving `Float3`

```
// Assuming you have a Float3 representing player velocity
Float3 playerVelocity = new Float3(0.5f, 0.0f, 1.0f);

// Convert to SerializableFloat3 and save
SerializableFloat3 serializableVelocity = new SerializableFloat3(playerVelocity);
SaveSystem.Save("playerVelocity", serializableVelocity, isGlobal: false);
```

Loading `Float3`

```
// Load the SerializableFloat3
SerializableFloat3 loadedVelocity =
SaveSystem.Load<SerializableFloat3>("playerVelocity", isGlobal: false);
```

```
// Convert back to Float3
Float3 playerVelocity = loadedVelocity.ToFloat3();
```