MassAPI Plugin Documentation

Overview

MassAPI is an Unreal Engine 5 plugin designed to significantly simplify interaction with the native **Mass Entity Component System (Mass ECS)**, particularly from Blueprints. While Mass ECS offers high performance for simulations involving large numbers of entities, its native C++ API can be complex, and its Blueprint accessibility is very limited out-of-the-box.

MassAPI bridges this gap by providing:

- **A Comprehensive Blueprint API:** Exposes core Mass functionality (entity creation, modification, querying, template handling) through user-friendly Blueprint nodes.
- **Dynamic Blueprint Nodes:** Utilizes custom K2Nodes for operations like getting/setting fragments and shared fragments. These nodes dynamically adapt their input/output pins to match the selected struct type, offering flexibility not possible with standard Blueprint functions.
- **Blueprint-Safe Data Types:** Wraps native Mass structs (FMassEntityHandle, FMassEntityTemplateData) into Blueprint-accessible USTRUCTs (FEntityHandle, FEntityTemplateData).
- **Editor-Friendly Templates:** Introduces FEntityTemplate for defining entity compositions in the editor (e.g., within Data Assets).
- **Simplified C++ API:** Offers a centralized UMassAPISubsystem for cleaner C++ interactions with the FMassEntityManager.
- **Blueprint Processor Pattern:** Enables creating Blueprint-based logic loops that operate on entities matching specific queries via the Get Matching Entities node.

Goal: To make Mass ECS significantly more accessible and productive for both Blueprint scripters and C++ programmers within Unreal Engine.

Quick Start

1. **Enable Plugin:** Ensure the MassAPI plugin (and its dependencies like MassEntity, MassGameplay, MassSpawner, etc.) are enabled in your project (Edit > Plugins). You might need to restart the editor.
2. **Access Nodes:** The Blueprint nodes provided by MassAPI can be found in the Blueprint editor's context menu, typically under the **Mass|Entity**, **Mass|Template**, and **Mass|Query** categories.
3. **Use Subsystem (C++):** Access the C++ API via the UMassAPISubsystem.

Blueprint API Documentation

The Blueprint API consists of several USTRUCTs for data handling and a

UBlueprintFunctionLibrary containing the nodes for operations.

Blueprint Structs

These structs allow Mass data to be passed around and manipulated in Blueprints.

1. **FEntityHandle** (MassAPIStructs.h)
    - **Purpose:** A Blueprint-safe wrapper for the native FMassEntityHandle. Represents a unique Mass entity.
    - **Fields:**
        - Index (int32): The entity's index.
        - Serial (int32): The entity's serial number.
    - **Usage:** Output by entity creation nodes (Build Entity From Template Data) and used as input for almost all entity operation nodes (Set Mass Fragment, Destroy Entity, etc.). Can be used as keys in Blueprint Maps and Sets.
    - **Note:** Use the IsValid (EntityHandle) node to check if the handle refers to a currently active entity.
2. **FEntityTemplate** (MassAPIStructs.h)
    - **Purpose:** An *editor-friendly* struct for defining the composition and initial data of an entity *before* runtime. Designed to be used in Data Assets or configuration files.
    - **Fields:** (Arrays of FInstancedStruct)
        - Tags: List of FMassTag-derived struct types to add.
        - Fragments: List of FMassFragment-derived structs *with their initial values*.
        - MutableSharedFragments: List of FMassSharedFragment-derived structs *with their initial values*.
        - ConstSharedFragments: List of FMassConstSharedFragment-derived structs *with their initial values*.
    - **Usage:** Create variables of this type in Data Assets or other configuration UObjects. Populate the arrays in the editor with the desired components and their default values. Use the Get Template Data node to convert this definition into a runtime-modifiable FEntityTemplateData handle.
3. **FEntityTemplateData** (MassAPIStructs.h)
    - **Purpose:** A Blueprint-safe *handle* to the *native* FMassEntityTemplateData. This represents a modifiable entity template definition *at runtime*.
    - **Fields:** None directly accessible in BP (it's a handle).
    - **Usage:** Created by the Get Template Data node (from an FEntityTemplate). Can be modified using nodes like Set Fragment in Template, Add Tag (TemplateData), etc. Finally, used as input for the Build Entity From Template Data / Build Entities From Template Data nodes.
    - **Note:** This struct uses a TSharedPtr internally to manage the lifetime of the native data.
4. **FEntityQuery** (MassAPIStructs.h)
    - **Purpose:** Defines criteria for matching Mass entities based on their component composition. Used by Match Entity Query and Get Matching Entities.

    - **Fields:** (Arrays of UScriptStruct*)
        - AllList: Entity must have ALL components listed here (AND logic).
        - AnyList: Entity must have AT LEAST ONE component listed here (OR logic). Supports Fragments and Tags only.
        - NoneList: Entity must have NONE of the components listed here (NOT logic).
    - **Usage:** Create variables of this type. Populate the arrays in the editor or via Blueprint nodes (like Make Array) with the desired Mass component types (FMassFragment, FMassTag, FMassSharedFragment, FMassConstSharedFragment).
    - **Validity:** A query generally needs at least one positive requirement (something in AllList or AnyList) to be considered valid by the Mass framework.

Blueprint Nodes (UMassAPIBPFnLib)

These nodes perform operations on entities and templates. Most require a WorldContextObject.

Entity Operations (Category: Mass|Entity)

- **IsValid (EntityHandle):** Checks if an FEntityHandle refers to a currently active, fully built entity. Returns true or false.
- **Has Mass Fragment:** Checks if an entity possesses a specific FMassFragment-derived struct type. Returns true or false.
- **Has Mass Tag:** Checks if an entity possesses a specific FMassTag-derived struct type. Returns true or false.
- **Get Mass Fragment:** (Custom K2Node) Retrieves a *copy* of a specific fragment's data from an entity.
    - Input: EntityHandle, FragmentType (UScriptStruct* limited to FMassFragment).
    - Output: OutFragment (Wildcard struct pin, dynamically typed), bSuccess (bool).
- **Set Mass Fragment:** (Custom K2Node) Sets the value of an existing fragment on an entity, or adds the fragment if it doesn't exist.
    - Input: EntityHandle, FragmentType (UScriptStruct* limited to FMassFragment), InFragment (Wildcard struct pin, dynamically typed).
    - Output: bSuccess (bool).
- **Add Mass Tag:** Adds a specific tag (FMassTag-derived struct type) to an entity. Does nothing if the tag is already present. This may change the entity's archetype.
    - Input: EntityHandle, TagType (UScriptStruct* limited to FMassTag).
- **Get Shared Mass Fragment:** (Custom K2Node) Retrieves a *copy* of a specific shared fragment's data associated with an entity.
    - Input: EntityHandle, FragmentType (UScriptStruct* limited to FMassSharedFragment).
    - Output: OutFragment (Wildcard struct pin, dynamically typed), bSuccess (bool).
- **Set Shared Mass Fragment:** (Custom K2Node) Associates an entity with a specific shared fragment instance based on the provided data. This may change the entity's archetype. If an instance with the same data already exists, the entity will point to that; otherwise, a new instance might be created.

    - Input: EntityHandle, FragmentType (UScriptStruct* limited to FMassSharedFragment), InFragment (Wildcard struct pin, dynamically typed).
    - Output: bSuccess (bool).
- **Get Const Shared Mass Fragment:** (Custom K2Node) Retrieves a *copy* of a specific const shared fragment's data associated with an entity.
    - Input: EntityHandle, FragmentType (UScriptStruct* limited to FMassConstSharedFragment).
    - Output: OutFragment (Wildcard struct pin, dynamically typed), bSuccess (bool).
- **Set Const Shared Mass Fragment:** (Custom K2Node) Associates an entity with a specific const shared fragment instance based on the provided data. This may change the entity's archetype. Works similarly to Set Shared Mass Fragment.
    - Input: EntityHandle, FragmentType (UScriptStruct* limited to FMassConstSharedFragment), InFragment (Wildcard struct pin, dynamically typed).
    - Output: bSuccess (bool).
- **Destroy Entity:** Immediately destroys a single Mass entity.
    - Input: EntityHandle.
- **Destroy Entities:** Immediately destroys an array of Mass entities (Batch Operation).
    - Input: EntityHandles (TArray<FEntityHandle>).

Entity Building (Category: Mass|Entity)

- **Build Entity From Template Data:** Creates a single Mass entity based on the provided FEntityTemplateData.
    - Input: TemplateData (FEntityTemplateData).
    - Output: Return Value (FEntityHandle).
- **Build Entities From Template Data:** Creates multiple Mass entities based on the provided FEntityTemplateData (Batch Operation).
    - Input: Quantity (int32), TemplateData (FEntityTemplateData).
    - Output: Return Value (TArray<FEntityHandle>).

Template Data Operations (Category: Mass|Template)

- **Get Template Data:** Converts an editor-defined FEntityTemplate struct into a runtime-modifiable FEntityTemplateData handle. This *creates a new instance* of the template data.
    - Input: Template (FEntityTemplate).
    - Output: Return Value (FEntityTemplateData).
- **IsEmpty (TemplateData):** Checks if the FEntityTemplateData handle contains any components (fragments, tags, etc.). Returns true or false.
- **Has Fragment (TemplateData):** Checks if the template data includes a specific FMassFragment type in its composition. Returns true or false.
- **Add Tag (TemplateData):** Adds a specific tag type to the template data composition. Modifies the input TemplateData.
    - Input: TemplateData (ref), TagType (UScriptStruct* limited to FMassTag).
- **Remove Tag (TemplateData):** Removes a specific tag type from the template data

composition. Modifies the input TemplateData.
    - Input: TemplateData (ref), TagType (UScriptStruct* limited to FMassTag).
- **Has Tag (TemplateData):** Checks if the template data includes a specific FMassTag type in its composition. Returns true or false.
- **Get Fragment from Template:** (Custom K2Node) Retrieves a *copy* of the initial value defined for a specific fragment type within the template data.
    - Input: TemplateData, FragmentType (UScriptStruct* limited to FMassFragment).
    - Output: OutFragment (Wildcard struct pin, dynamically typed), bSuccess (bool).
- **Set Fragment in Template:** (Custom K2Node) Sets or adds the initial value for a specific fragment type within the template data. *Creates a new template data instance and returns its handle* (immutable operation).
    - Input: TemplateData (ref), FragmentType (UScriptStruct* limited to FMassFragment), InFragment (Wildcard struct pin, dynamically typed).
- **Set Shared Fragment in Template:** (Custom K2Node) Sets or adds the shared fragment handle for a specific type within the template data based on the provided initial value. *Creates a new template data instance*. Requires WorldContextObject to access the EntityManager for handle creation.
    - Input: WorldContextObject, TemplateData (ref), FragmentType (UScriptStruct* limited to FMassSharedFragment), InFragment (Wildcard struct pin, dynamically typed).
- **Set Const Shared Fragment in Template:** (Custom K2Node) Sets or adds the const shared fragment handle for a specific type within the template data. *Creates a new template data instance*. Requires WorldContextObject.
    - Input: WorldContextObject, TemplateData (ref), FragmentType (UScriptStruct* limited to FMassConstSharedFragment), InFragment (Wildcard struct pin, dynamically typed).

Entity Querying (Category: Mass|Query)

- **Match Entity Query:** Checks if a single entity's current component composition matches the requirements defined in an FEntityQuery struct.
    - Input: EntityHandle, Query (FEntityQuery ref).
    - Output: Return Value (bool).
- **Get Matching Entities:** Executes an FEntityQuery and returns an array of all currently active entities that match its requirements.
    - Input: Query (FEntityQuery ref).
    - Output: Return Value (TArray<FEntityHandle>).
    - **Performance Warning:** Use this node judiciously. Iterating the resulting array in Blueprints to access/modify entity data is significantly slower than native C++ Mass Processors, especially for thousands of entities. It's best suited for smaller entity counts or logic that doesn't run every frame.

Custom K2Nodes

Nodes like Get/Set Mass Fragment, Get/Set Shared Mass Fragment, etc., are custom K2Nodes

(inheriting from UK2Node_FragmentFunction and similar base classes).

- **Why?** Standard UFUNCTIONs cannot have wildcard struct pins (FGenericStruct) that dynamically change their type in the editor based on another pin's selection.
- **How?** These nodes use internal editor logic (OnFragmentTypeChanged) triggered when you select a struct in the FragmentType pin. This logic modifies the InFragment or OutFragment pin to match the selected type, providing a seamless Blueprint experience. They call special CustomThunk C++ functions (Generic_...) that handle the raw memory manipulation behind the scenes.

C++ API Documentation (UMassAPISubsystem)

For C++ developers, the UMassAPISubsystem provides a cleaner, more convenient interface than directly using the FMassEntityManager for many common operations.

Accessing the Subsystem

```
#include "MassAPISubsystem.h" // Include the header

// Get a pointer (check for null)
if (UMassAPISubsystem* MassAPI = UMassAPISubsystem::GetPtr(WorldContextObject))
{
   // Use MassAPI->...
}

// Get a reference (will assert if null)
UMassAPISubsystem& MassAPI = UMassAPISubsystem::GetRef(WorldContextObject);
// Use MassAPI...
```

Key Functions

(See MassAPISubsystem.h for full details and template argument constraints)

- **FMassEntityManager* GetEntityManager() const:** Gets the underlying entity manager.
- **bool IsValid(FMassEntityHandle EntityHandle) const:** Checks if an entity is active.
- **TArray<FMassEntityHandle> BuildEntities(int32 Quantity, FMassEntityTemplateData& TemplateData) const:** Builds entities from template data (synchronous).
- **template<typename... TArgs> TArray<FMassEntityHandle> BuildEntities(int32 Quantity, TArgs&&... Args) const:** Builds entities directly from fragment/tag/shared fragment arguments (synchronous). Convenience wrapper.
- **template<typename... TArgs> FMassEntityHandle BuildEntity(TArgs&&... Args) const:** Builds a single entity directly.

- **void DestroyEntity(FMassEntityHandle EntityHandle) const:** Destroys an entity (synchronous).
- **bool HasFragment(FMassEntityHandle EntityHandle, const UScriptStruct* FragmentType) const:** Checks for a fragment by type.
- **template<typename T> bool HasFragment(FMassEntityHandle EntityHandle) const:** Checks for a fragment by template type.
- **bool HasTag(FMassEntityHandle EntityHandle, const UScriptStruct* TagType) const:** Checks for a tag by type.
- **template<typename T> bool HasTag(FMassEntityHandle EntityHandle) const:** Checks for a tag by template type.
- **template<typename T> T GetFragment(FMassEntityHandle EntityHandle) const:** Gets a copy of fragment data. Asserts if fragment doesn't exist.
- **template<typename T> T* GetFragmentPtr(FMassEntityHandle EntityHandle) const:** Gets a mutable pointer to fragment data (or null).
- **template<typename T> T& GetFragmentRef(FMassEntityHandle EntityHandle) const:** Gets a mutable reference to fragment data. Asserts if fragment doesn't exist.
- **void AddFragment(FMassEntityHandle EntityHandle, const FInstancedStruct& FragmentStruct) const:** Adds a fragment with an initial value (synchronous). Ignores if fragment type already exists.
- **template<typename T> void AddFragment(FMassEntityHandle EntityHandle, const T& FragmentValue) const:** Template version of AddFragment.
- **template<typename T> void RemoveFragment(FMassEntityHandle EntityHandle) const:** Removes a fragment (synchronous).
- **template<typename T> void AddTag(FMassEntityHandle EntityHandle) const:** Adds a tag (synchronous).
- **template<typename T> void RemoveTag(FMassEntityHandle EntityHandle) const:** Removes a tag (synchronous).
- **(Shared/Const Shared Getters/Setters):** Similar Get/Has/Add/Remove functions exist for SharedFragment and ConstSharedFragment (e.g., GetSharedFragmentRef, AddConstSharedFragment). See the header file.
- **Query Matching:**
  - bool MatchQuery(FMassEntityHandle EntityHandle, const FEntityQuery& Query) const: Checks if a single entity matches a Blueprint FEntityQuery.
  - Static helpers HasAll, HasAny, MatchQueryAll, MatchQueryAny, MatchQueryNone, MatchQuery(Composition, Query) for checking FMassArchetypeCompositionDescriptor against queries.

Usage Examples

Example 1: Defining and Spawning Entities (Blueprint)

1. **Create Data Asset:** Create a new Data Asset based on a custom C++ or Blueprint class. Add an FEntityTemplate variable (e.g., MyAgentTemplate).
2. **Configure Template:** Open the Data Asset and configure MyAgentTemplate:

    - Add FMassAgent tag to the Tags array.
    - Add FTransformFragment to the Fragments array and set an initial location.
    - Add FMyHealthFragment to the Fragments array and set initial health.
    - Add FTeamSharedFragment to MutableSharedFragments and set the initial team ID.
3. **Spawn Logic (e.g., in GameMode BeginPlay):**
    - Get a reference to your Data Asset.
    - Use Get Template Data node, passing the MyAgentTemplate from the Data Asset. Store the output FEntityTemplateData handle.
    - (Optional) Modify the runtime template data if needed using Set Fragment in Template, etc. For example, override the initial position based on spawn location.
    - Use Build Entities From Template Data, passing the quantity and the (potentially modified) FEntityTemplateData handle. Store the resulting FEntityHandle array.

Example 2: Accessing and Modifying Fragment Data (Blueprint)

```
// Assuming 'MyEntityHandle' is a valid FEntityHandle variable
// Get Health
Get Mass Fragment (Target: Self, Entity Handle: MyEntityHandle, Fragment Type:
MyHealthFragment) -> OutFragment, bSuccess
IF (bSuccess) THEN
   Break MyHealthFragment (Struct: OutFragment) -> CurrentHealth
   Print String ("Health: " + CurrentHealth)
ENDIF

// Set Health (Example: Take Damage)
Make MyHealthFragment (CurrentHealth: CurrentHealth - 10) -> NewHealthFragment
Set Mass Fragment (Target: Self, Entity Handle: MyEntityHandle, Fragment Type:
MyHealthFragment, InFragment: NewHealthFragment) -> bSuccess
```

Example 3: Adding a Tag (Blueprint)

```
// Add a "Stunned" tag
Add Mass Tag (Target: Self, Entity Handle: MyEntityHandle, Tag Type: StunnedTag)
```

Example 4: Querying Entities and Applying Logic (Blueprint - "BP Processor")

```
// In a Subsystem Tick, Actor Tick, or Timer function:

// Define the Query (can be a variable)
Make Entity Query (
   AllList: [MyAgentTag, TransformFragment],
```

```
    AnyList: [],
    NoneList: [StunnedTag]
) -> MyQuery

// Get Matching Entities
Get Matching Entities (Target: Self, Query: MyQuery) -> MatchingEntities

// Loop through results
ForEachLoop (Array: MatchingEntities)
    Loop Body -> Array Element (FEntityHandle, named CurrentEntity)
        // Get Transform
        Get Mass Fragment (Entity Handle: CurrentEntity, Fragment Type: TransformFragment) ->
TransformData, bGotTransform
        // Get Velocity
        Get Mass Fragment (Entity Handle: CurrentEntity, Fragment Type: VelocityFragment) ->
VelocityData, bGotVelocity

        IF (bGotTransform AND bGotVelocity) THEN
            Break TransformFragment (Struct: TransformData) -> CurrentTransform
            Break VelocityFragment (Struct: VelocityData) -> CurrentVelocity

            // Simple Movement Logic
            NewLocation = CurrentTransform.GetLocation() + CurrentVelocity * DeltaTime
            Make TransformFragment (Transform: CurrentTransform.SetLocation(NewLocation)) ->
NewTransformFragment

            // Set updated Transform
            Set Mass Fragment (Entity Handle: CurrentEntity, Fragment Type: TransformFragment,
InFragment: NewTransformFragment)
        ENDIF
    Completed -> ...
EndForEachLoop
```

Remember the performance implications of this Blueprint loop compared to C++ Processors.

Example 5: Spawning Entities (C++)

```
#include "MassAPISubsystem.h"
#include "MyFragments.h" // Assuming you have defined FLocationFragment,
FVelocityFragment etc.
```

```
void AMySpawner::SpawnAgents(int32 Count)
{
   UMassAPISubsystem& MassAPI = UMassAPISubsystem::GetRef(this);

   FLocationFragment StartLoc;
   StartLoc.Location = FVector(0,0,100);

   FVelocityFragment StartVel;
   StartVel.Velocity = FVector(100, 0, 0);

   // Spawn using variadic arguments
   TArray<FMassEntityHandle> Handles = MassAPI.BuildEntities(Count, FMyAgentTag{},
StartLoc, StartVel);

   UE_LOG(LogTemp, Log, TEXT("Spawned %d agents."), Handles.Num());
}
```

Example 6: Modifying Fragment Data (C++)

```
#include "MassAPISubsystem.h"
#include "MyFragments.h"

void UMyGameplaySystem::ApplyDamage(FMassEntityHandle Entity, float Damage)
{
   UMassAPISubsystem& MassAPI = UMassAPISubsystem::GetRef(this);

   if (MassAPI.IsValid(Entity))
   {
      // Get a reference to modify directly
      if (FMyHealthFragment* Health = MassAPI.GetFragmentPtr<FMyHealthFragment>(Entity))
      {
         Health->CurrentHealth -= Damage;
         if (Health->CurrentHealth <= 0)
         {
            // Add a "Dead" tag
            MassAPI.AddTag<FDeadTag>(Entity);
            // Maybe remove movement fragments
            MassAPI.RemoveFragment<FVelocityFragment>(Entity);
         }
      }
   }
}
```

Entity Query (FEntityQuery) Explained

The FEntityQuery struct (Blueprint) is the primary way to define *which* entities you want to operate on when using Match Entity Query or Get Matching Entities. It mirrors the core concepts of Mass ECS queries used by native processors.

How it Works:

When you use Get Matching Entities:

1. The node takes your Blueprint FEntityQuery.
2. Internally, it creates a native C++ FMassEntityQuery.
3. It iterates through your AllList, AnyList, and NoneList.
4. For each valid Mass component struct (FMassFragment, FMassTag, FMassSharedFragment, etc.) found in your lists, it calls the *appropriate* Add...Requirement function on the native query (e.g., AddTagRequirement for tags, AddSharedRequirement for shared fragments).
5. It tells the native query it only needs ReadOnly access since it's just fetching handles.
6. It calls NativeQuery.GetMatchingEntityHandles(). This function:
    - Calls CacheArchetypes() internally, which finds all archetypes in the EntityManager that satisfy *all* the requirements (All, Any, None combined). It also performs validation checks.
    - Collects the FMassEntityHandles from all matching archetypes.
7. The node converts the native FMassEntityHandle array back to an FEntityHandle array for Blueprint.

Key Concepts:

- **Composition Matching:** Queries work by matching the *composition* of archetypes, not by inspecting individual entities one by one (which is much slower). An archetype matches if its set of components satisfies all the rules defined by the query.
- **AND (AllList):** Every component listed here *must* be present in the entity's archetype.
- **OR (AnyList):** At least one of the components listed here *must* be present. (Note: In MassAPI's BP implementation, this primarily works reliably for Fragments and Tags due to limitations/complexity in the underlying native query structure for shared/chunk components with 'Any').
- **NOT (NoneList):** None of the components listed here can be present in the entity's archetype.
- **Optional:** (Not directly exposed in the BP FEntityQuery struct but part of the native query system). Allows processing entities whether or not they have certain components.
- **Access Mode (ReadOnly / ReadWrite):** Tells the system whether the operation intends to modify the component data. GetMatchingEntities always uses ReadOnly. Native C++ processors specify this accurately for performance and safety.

By combining All, Any, and None, you can create precise filters to target specific types of entities for your Blueprint logic loops.