This document is best viewed with this option in the menubar: *View → Text Width → Wide*

Unity Entities 101

Entities and components

An entity is a lightweight, unmanaged alternative to a GameObject. Entities resemble GameObjects in many ways and can serve a similar role, but they have key differences:

- Unlike a GameObject, an entity is not a managed object but simply a unique identifier number.
- The components of an entity are usually struct values.
- The components of an entity has no equivalents of the MonoBehaviour "event functions" (such as OnUpdate or OnStart).
- Although entity component types can be given methods, doing so is generally discouraged.
- A single entity can only have one component of any given type, *e.g.* a single entity cannot have two components both of type *Foo*.
- An entity has no built-in concept of parenting. Instead, the standard Parent component contains a reference to another entity, allowing formation of entity transform hierarchies.

A basic component type is defined by creating a struct that implements IComponentData.

```java
// an entity component type with two fields
public struct Health : IComponentData
{
	public int HitPoints;
	public float ArmourRating:
}
```

An IComponentData struct is expected to be unmanaged, so it cannot contain any managed field types. Specifically, the allowed field types are:

- Blittable types
- bool
- char
- BlobAssetReference<T>, a reference to a Blob data structure
- Collections.FixedString, a fixed-sized character buffer
- Collections.FixedList

- Fixed array (only allowed in an unsafe context)
- Other struct types that conform to these same restrictions.

Entity Worlds and EntityManagers

A World is a collection of entities. An entity's ID number is only unique within its own world, *i.e.* the entity with a particular ID in one world is entirely unrelated to an entity with the same ID in a different world.

A world also owns a set of systems, which are units of code that run on the main thread, usually once per frame. The entities of a world are normally only accessed by the world's systems and the jobs scheduled by those systems (but this is not an enforced restriction).

The entities in a world are created, destroyed, and modified through the world's EntityManager, whose methods include:

- CreateEntity(): Creates a new entity.
- Instantiate(): Creates a new entity with a copy of all the components of an existing entity.
- DestroyEntity(): Destroys an existing entity.
- AddComponent<T>(): Adds a component of type T to an existing entity.
- RemoveComponent<T>(): Removes a component of type T from an existing entity.
- HasComponent<T>(): Returns true if an entity currently has a component of type T.
- GetComponent<T>(): Retrieves the value of an entity's component of type T.
- SetComponent<T>(): Overwrites the value of an entity's component of type T.

Archetypes

An archetype represents a particular unique combination of component types in a world: all of the entities in a world with a certain set of component types are stored together in the same archetype. For example:

- All of a world's entities with component types A, B, and C, are stored together in one archetype,
- ...the entities with just component types A and B (but not C) are stored together in a second archetype,
- ...and all the entities with component types B and D are stored in a third archetype.

Effectively, adding or removing components of an entity changes which archetype the entity belongs to, which necessitates that the EntityManager actually move the entity to a new archetype.

When you add or remove components from an entity, the EntityManager moves the entity to the appropriate archetype. For example, if an entity has component types X, Y, and Z and you remove its Y component, the EntityManager moves the entity to the archetype that has component types X and Z, copying over the X and Z values. If no such archetype already exists in the world, the EntityManager creates it.

> **Warning**: The costs of frequently moving many entities between archetypes may add up to be significant.

Archetypes are created by the EntityManager as you create and modify entities, so you don't have to worry about creating archetypes explicitly.

Even if all the entities are removed from an archetype, the archetype is not destroyed until its world is destroyed.

Chunks

The entities of an archetype are stored in 16KiB blocks of memory belonging to the archetype called *chunks*. Each chunk stores up to 128 entities. (In an archetype where the space required for each entity exceeds 16KiB / 128, the max number of entities per chunk will be lower).

The entity ID's and components of each type are stored in their own separate arrays within the chunk. For example, in the archetype for entities which have component types A and B, each chunk will store three arrays:

- one array for the entity ID's
- ...a second array for the A components
- ...and a third array for the B components.

The ID and components of the first entity in the chunk are stored at index 0 of these arrays, the second entity at index 1, the third entity at index 2, and so forth.

A chunk's arrays are always kept tightly packed:

- When a new entity is added to the chunk, it is stored in the first free index of the arrays.
- When an entity is removed from the chunk (which happens either because the entity is being destroyed or because it's being moved to another archetype), the last entity in the chunk is moved to fill in the gap.

The creation and destruction of chunks is handled by the EntityManager:

- The EntityManager creates a new chunk only when an entity is added to an archetype whose already existing chunks are all full.
- The EntityManager only destroys a chunk when the chunk's last entity is removed.

Any EntityManager operation that adds, removes, or moves entities within a chunk is called a structural change. Such changes can only be made on the main thread, not in jobs (though as we'll discuss later, an EntityCommandBuffer can be used as a work around).

Queries

An EntityQuery efficiently finds all entities having a specified set of component types. For example, if a query looks for all entities having component types A and B, then the query will gather the chunks of all archetypes which include A and B, regardless of whatever other component types those archetypes might have. Such a query would match the entities with component types A and B, but the query would also match, say, the entities with component types A, B, and C.

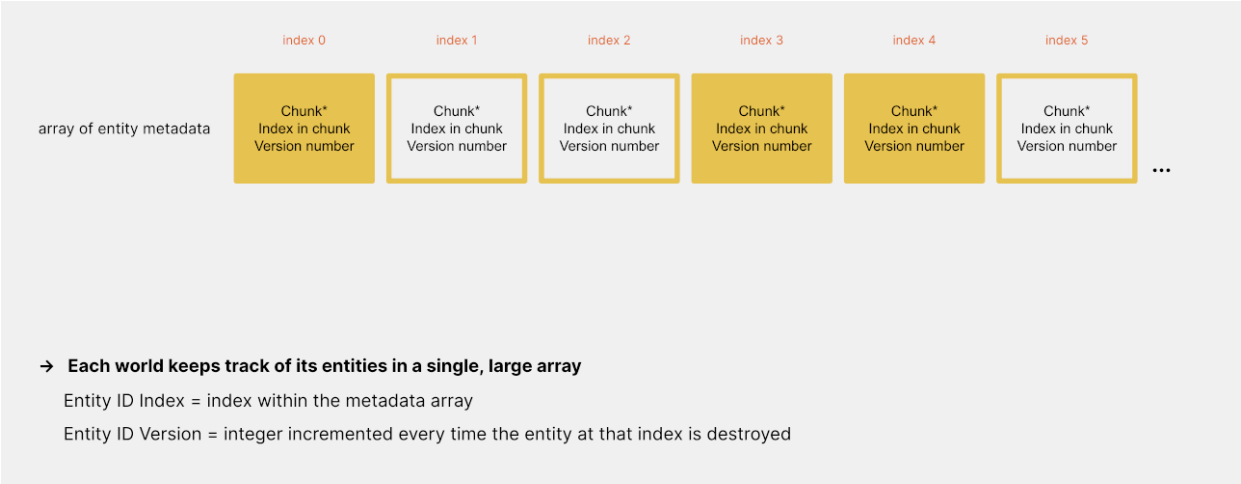
> **Note**: The archetypes matching a query will get cached until the next time a new archetype is added to the world. Because the set of existing archetypes in a world tends to stabilize early in the lifetime of a program, this caching usually helps make the queries much cheaper.

A query can also specify component types to exclude from the matching archetypes. For example, if a query looks for 'all entities having component types A and B but *not* having component type C', the query would match entities with component types A and B but not match entities with component types A, B, and C.

Entity ID's

An entity ID is represented by the struct Entity, which consists of two ints: an *index* and a *version*.

In order to look up entities by ID, the world's EntityManager maintains an array of entity metadata. The index of an entity denotes its slot in this metadata array, and the slot stores a pointer to the chunk where that entity is stored, as well as the index of the entity within the chunk. When no entity exists for a particular index, the chunk pointer at that index is null. Here, for example, no entities with indexes 1, 2, and 5 currently exist, so the chunk pointers in those slots are all null:



The entity version number allows entity indexes to be reused after an entity is destroyed: when an entity is destroyed, the version number stored at its index is incremented, and so if an ID's version number doesn't match the one stored at the index, then the ID must refer to an entity that has already been destroyed or perhaps never existed.

Tag components

An IComponentData struct with no fields is called a tag component. Although tag components store no data, they can still be added and removed from entities like any other component type, which is useful for queries. For example, if all of our entities representing monsters have a Monster tag component, a query for the Monster component type will match all the monster entities.

```java
// a tag component
public struct Monster : IComponentData
{
}
```

DynamicBuffer components

A DynamicBuffer is a component type which is a resizable array. To define a DynamicBuffer component type, create a struct that implements the IBufferElementData interface.

```java
// a dynamic buffer component type
public struct Waypoint : IBufferElementData
{
	public float3 Value:
}
```

The buffer of each entity stores a Length, a Capacity, and a pointer:

- The Length is the number of elements in the buffer. It starts at 0 and increments when you append a value to the buffer.
- The Capacity is the amount of storage in the buffer. It starts out matching the internal buffer capacity (which defaults to 128 / sizeof(T) but can be specified by the InternalBufferCapacity attribute on the IBufferElementData struct). Setting Capacity resizes the buffer.
- The pointer indicates the location of the buffer's contents. Initially it is null, signifying that the contents are stored directly in the chunk. If the capacity is set to exceed the internal buffer capacity, a new larger array is allocated outside of the chunk, the contents are copied to this external array, and the pointer is set to point to this new array. If the length of the buffer ever exceeds the capacity of the external array, then the contents of the buffer are copied to another new, larger array outside the chunk, and the old array is disposed. The buffer can also be shrunk.

The internal buffer capacity and the external capacity (if present) are deallocated when the EntityManager destroys the chunk itself.

> **Note**: When a dynamic buffer is stored outside the chunk, the internal capacity is effectively wasted, and accessing the buffer contents requires following an extra pointer. These costs can be avoided if you ensure the internal capacity is never exceeded. Of course, in many cases, staying within this limit may require an excessively large internal capacity. Another option is to set the internal capacity to 0, which means that any non-empty buffer will always be stored outside the chunk. This incurs the cost of always following a pointer when accessing the buffer, but it avoids wasting unused space in the chunk.

The EntityManager has these key methods for dynamic buffers:

- AddComponent<T>(): Adds a component of type T to an entity, where T can be a dynamic buffer component type.
- AddBuffer<T>(): Adds a dynamic buffer component of type T to an entity; returns the new buffer as a DynamicBuffer<T>.

- RemoveComponent<T>(): Removes the component of type T from an entity, where T can be a dynamic buffer component type.
- HasBuffer<T>(): Returns true if an entity currently has a dynamic buffer component of type T.
- GetBuffer<T>(): Returns an entity's dynamic buffer component of type T as a DynamicBuffer<T>.

DynamicBuffer<T> represents the dynamic buffer component of type T of an individual entity. Its key properties and methods include:

- Length: Gets or sets the length of the buffer.
- Capacity: Gets or sets the capacity of the buffer.
- Item[Int32]: Gets or sets the element at a specified index.
- Add(): Adds an element to the end of the buffer, resizing it if necessary.
- Insert(): Inserts an element at a specified index, resizing if necessary.
- RemoveAt(): Removes the element at a specified index.

> **Note**: Performing any structural change operation will 'invalidate' a DynamicBuffer, meaning an exception will be thrown if the DynamicBuffer is subsequently used. To use a buffer again after a structural change, it must be re-retrieved.

Systems

A system is a unit of code which belongs to an entity world and which runs on the main thread (usually once per frame). Normally, a system will only access entities of its own world, but this is not an enforced restriction.

A system is defined as a struct implementing the ISystem interface, which has three key methods:

- OnUpdate(): Normally called once per frame (though this depends upon the system group to which the system belongs).
- OnCreate(): Called before the first call to OnUpdate and whenever a system resumes running.
- OnDestroy(): Called when a system is destroyed.

> **Note**: These methods have default do-nothing implementations, so they can be omitted in your systems when you don't need them, *e.g.* if you leave the OnCreate body empty, you can just omit the whole method.

If a system's Enabled property is set to false, its updates will be skipped.

A system may additionally implement ISystemStartStop, which has these methods:

- OnStartRunning(): Called before the first call to OnUpdate and after the system has been re-enabled (meaning the system's Enabled property is changed from false to true).
- OnStopRunning(): Called before OnDestroy and after the system has been disabled (meaning the system's Enabled property is changed from true to false).

System groups and system update order

The systems of a world are organized into *system groups*. Each system group has an ordered list of children (systems and other system groups), and by default, each system group calls the OnUpdate of its children in their sorted order. Effectively, the system groups form a hierarchy which determines the system update order.

- A system group is defined as a class inheriting from ComponentSystemGroup.
- When a system group is updated, the group normally updates its children in their sorted order, but this default behavior can be overridden by overriding the group's update method. For example, the FixedStepSimulationGroup has custom update behaviour that will update its children zero or more times per frame to approximate a fixed update interval.
- A group's children are re-sorted every time a child is added or removed from the group.
- Children are added to a system group with the UpdateInGroup attribute. Without this attribute, systems and system groups are added by default to the SimulationSystemGroup.
- The UpdateBefore and UpdateAfter attributes can be used to determine the relative sort order amongst the children in a group. For example, if a *FooSystem* has the attribute [UpdateBefore(typeof(BarSystem))], then *FooSystem* will be put somewhere before *BarSystem* in the sorted order. If, however, *FooSystem* and *BarSystem* don't belong to the same system group, this attribute will have no effect other than trigger a warning.
- If the ordering attributes of a group's children create a contradiction (*e.g.* child A is marked to update before child B but also B is marked to update before A), an exception is thrown when the group's children are sorted.

```Java
// A system group.
// Because this group doesn't override OnUpdate, it follows the default
// behaviour: its children will be updated in their sorted order.
public class MonsterSystemGroup : ComponentSystemGroup
{
}
```

```
// a child system of the MonsterSystemGroup
[UpdateInGroup(typeof(MonsterSystemGroup))]
public struct VampireSystem : ISystem
{
       ...
}
```

Creating worlds and systems

When entering play mode, a default, automatic bootstrapping process creates a default world with three system groups:

- InitializationSystemGroup: Updates at the end of the Initialization phase of the Unity player loop.
- SimulationSystemGroup: Updates at the end of the Update phase of the Unity player loop. Generally the place to put game logic.
- PresentationSystemGroup: Updates at the end of the PreLateUpdate phase of the Unity player loop. Generally the place for rendering code.

Automatic bootstrapping creates an instance of every system and system group (except those with the DisableAutoCreation attribute). These instances are added to the SimulationSystemGroup unless overridden by the UpdateInGroup attribute. For example, if a system has the attribute [UpdateInGroup(typeof(InitializationSystemGroup))], then the system will be added to the InitializationSystemGroup instead of the SimulationSystemGroup.

The automatic bootstrapping process can be disabled with scripting defines:

- #UNITY_DISABLE_AUTOMATIC_SYSTEM_BOOTSTRAP_RUNTIME_WORLD: Disables automatic bootstrapping of the default world.
- #UNITY_DISABLE_AUTOMATIC_SYSTEM_BOOTSTRAP_EDITOR_WORLD: Disables automatic bootstrapping of the Editor world.
- #UNITY_DISABLE_AUTOMATIC_SYSTEM_BOOTSTRAP: Disables automatic bootstrapping of both the default world and the Editor world.

When automatic bootstrapping is disabled, your code is responsible for:

- Creating worlds.
- Calling World.GetOrCreateSystem<T>() to add the system and system group instances to the worlds.

- Registering top-level system groups (like SimulationSystemGroup) to update in the Unity player loop.

Alternatively, you can customize the bootstrapping logic by creating a class that implements ICustomBootstrap.

Time in worlds and systems

A world has a Time property, which returns a TimeData struct, containing the frame delta time and elapsed time. The time value is updated by the world's UpdateWorldTimeSystem. The time value can be manipulated with these World methods:

- SetTime: Set the time value.
- PushTime: Temporarily change the time value.
- PopTime: Restore the time value from before the last push.

Some system groups, such as FixedStepSimulationSystemGroup, push a time value before updating their children and then pop the value once done updating. Effectively, these system groups present “false” time values to their children.

SystemState

A system's OnUpdate(), OnCreate(), and OnDestroy() methods are passed a SystemState parameter. SystemState represents the state of the system instance and has important methods and properties, including:

- World: The system's world.
- EntityManager: The EntityManager of the system's world.
- Dependency: A JobHandle used to pass job dependencies between systems.
- GetEntityQuery(): Returns an EntityQuery.
- GetComponentTypeHandle<T>(): Returns a ComponentTypeHandle<T>.
- GetComponentLookup<T>(): Returns a ComponentLookup<T>.

> **Important**: Although entity queries, component type handles, and component lookups can be acquired directly from the EntityManager, it is generally proper for a system to only acquire these things from the SystemState instead. By going through SystemState, the component types accessed by the system get tracked, which is essential for the system’s Dependency property to correctly pass job dependencies between systems (discussed later).

SystemAPI

The SystemAPI class has many static convenience methods, covering much of the same functionality as World, EntityManager, and SystemState.

The SystemAPI methods rely upon source generators, so they only work in systems and IJobEntity (but not IJobChunk). The advantage of using SystemAPI is that these methods produce the same results in both contexts, so code that uses SystemAPI will generally be easier to copy-paste between these two contexts.

> **Note**: If you get confused about where to look for key Entities functionality, the general rule is to check SystemAPI first. If SystemAPI doesn't have what you're looking for, look in SystemState, and if what you're looking for isn't there, look in the EntityManager and World.

SystemAPI also provides a special convenience Query() method that, through source generation, helps create a foreach loop over the entities and components that match a query (demonstrated in the samples below).

Accessing entities in jobs

You can offload the processing of entity data to worker threads with the C# Job System. The Entities package has two interfaces for defining jobs that access entities:

- IJobChunk, whose Execute() method is called once for each individual chunk matching the query.
- IJobEntity, whose Execute() method is called once for each entity matching the query.

Although IJobEntity is generally more convenient to write and use, IJobChunk provides more precise control. In most cases, their performance is identical for equivalent work.

> **Note**: IJobEntity is not actually a 'real' job type: source generation extends an IJobEntity struct with an implementation of IJobChunk, so actually, an IJobEntity is ultimately scheduled as an IJobChunk.

To split the work of an IJobChunk or IJobEntity across multiple threads, schedule the job by calling ScheduleParallel() instead of Schedule(). When you use ScheduleParallel(), the chunks matching the query will be put into separate batches, and these batches will be farmed out to the worker threads.

Structural changes cannot be made inside a job, so you should only make structural changes on the main thread. However, a job can record structural change commands in an EntityCommandBuffer (discussed later), and then these commands can be played back later on the main thread.

Sync points

Some operations on the main thread trigger 'synchronization points' that complete some or all of the currently scheduled jobs. For example, a call to EntityManager.AddComponent<T>() will first complete all currently scheduled jobs which access any T components. Likewise, the EntityQuery methods ToComponentDataArray<T>(), ToEntityArray(), and ToArchetypeChunkArray() must first complete any currently scheduled jobs which access any of the same components as the query.

In many cases, these synchronization points will also 'invalidate' existing instances of a few types, namely DynamicBuffer and ComponentLookup<T>. When an instance is invalidated, calling its methods will throw safety check exceptions. If an instance you need to still use gets invalidated, you must retrieve a new instance to replace it.

Component safety handles

Just like the native collections, each component type has an associated job safety handle for each world. The implication is that, for any two jobs which access the same component type in a world, the safety checks won't let the jobs be scheduled concurrently. For example, when we try scheduling a job that accesses component type *Foo*, the safety checks will throw an exception if an already scheduled job also accesses component type *Foo*. To avoid this exception:

- the already scheduled job must be completed before scheduling the new job
- …or the new job must depend upon the already scheduled job.

> **Note**: It's safe for two jobs to be scheduled concurrently if they both have read-only access of the same component type. For any component type in your job that is not ever written, be sure to inform the safety checks by marking the component type handle with the ReadOnly attribute.

A DynamicBuffer<T> instance itself holds a safety handle:

- The contents of a DynamicBuffer<T> cannot be accessed while any scheduled jobs that access the same buffer component type remain uncompleted.

- If, however, the uncompleted jobs all have just read-only access to the buffer component type, then the main thread is allowed to read the buffer.

SystemState.Dependency

When we schedule a job in a system, we want it to depend upon any currently scheduled jobs that might conflict with the new job, even if those jobs were scheduled in other systems. To help arrange these dependencies, SystemState has a JobHandle property named Dependency.

Immediately before a system updates:

1. the system's Dependency property is completed
2. ...and then Dependency is assigned a combination of the Dependency job handles of all *other* systems which access any of the same component types as this system. For example, in a system which accesses the *Foo* and *Bar* component types, the Dependency of all other systems in the world which also access either *Foo* or *Bar* will be combined into a job handle that is assigned to this system's Dependency.

You are then expected to follow two rules in every system:

1. All jobs scheduled in a system update should (directly or indirectly) depend upon the job handle that was assigned to Dependency right before the update.
2. Before a system update returns, the Dependency property should be assigned a handle that includes all the jobs scheduled in that update.

As long as you follow these two rules, every job scheduled in a system update will depend upon all jobs scheduled in other systems which might access any of the same component types.

> **Important**: Systems do not track which native collections they use, so the Dependency property only accounts for component types, not native collections. Consequently, if two systems both schedule jobs which use the same native collection, their Dependency job handles will not necessarily be included in the job handle assigned to the Dependency property of the other, and so the jobs of the different systems will not depend upon each other as they should. In these scenarios, you could arrange the dependencies by manually sharing job handles between the systems, but often the better solution is to store the native collection in a component: if you follow the two rules and both systems access the collection through the same component type, the jobs scheduled in both systems should then depend upon each other.

ComponentLookup<T>

The components of individual entities can be randomly accessed through an EntityManager, but we generally can't use an EntityManager inside a job. Instead, we can use a type called ComponentLookup<T>, which gets and sets component values by entity ID. We can also get dynamic buffers by entity ID using BufferLookup<T>.

> **Important**: Keep in mind that looking up an entity by ID tends to incur the performance cost of cache misses, so it's generally a good idea to avoid lookups when you can. There are, though, of course, many problems which require random lookups to solve, so by no means can random lookups be avoided entirely! Just avoid using them carelessly.

The ComponentLookup<T> and BufferLookup<T> method HasComponent() returns true if the specified entity has the component type T. The TryGetComponent<T>() and TryGetBuffer<T>() methods do the same but also output the component value or buffer if it exists.

> **Note**: To test whether an entity simply exists, we can call Exists() of an EntityStorageInfoLookup. Indexing an EntityStorageInfoLookup returns an EntityStorageInfo struct, which includes a reference to the entity's chunk and its index within the chunk.

If a job needs to only read the components accessed through a ComponentLookup<T>, the ComponentLookup<T> field should be marked with the ReadOnly attribute to inform the job safety checks. The same is true for a BufferLookup<T>.

In a parallel-scheduled job, getting component values from a ComponentLookup<T> requires the field to be marked with the ReadOnly attribute. The safety checks do not allow setting component values through a ComponentLookup<T> in a parallel-scheduled job because safety cannot be guaranteed. However, you can fully disable the safety checks on the ComponentLookup<T> by marking it with the NativeDisableParallelForRestriction attribute. The same is true for a BufferLookup<T>. Just make sure that your code sets component values in a thread-safe manner!

Entity command buffers

Changes to entities can be deferred by recording commands into an EntityCommandBuffer. The recorded commands are executed when we call the Playback() method on the main thread.

Deferring changes with an EntityCommandBuffer is particularly useful in jobs because jobs can't directly make structural changes (*i.e.* create entities, destroy entities, add components, or

remove components). Instead, jobs can record commands in an EntityCommandBuffer to be played back on the main thread after the job has been completed. EntityCommandBuffers can also help us avoid unnecessary sync points by deferring structural changes to a few consolidated points of the frame rather than scattered across the frame.

An EntityCommandBuffer has many (but not all) of the same methods as EntityManager. The methods include:

- CreateEntity(): Records a command to create a new entity. Returns a temporary entity ID.
- DestroyEntity(): Records a command to destroy an entity.
- AddComponent<T>(): Records a command to add a component of type T to an entity.
- RemoveComponent<T>(): Records a command to remove a component of type T from an entity.
- SetComponent<T>(): Records a command to set a component value of type T.
- AppendToBuffer(): Records a command that will append an individual value to the end of an existing buffer component.
- AddBuffer(): Returns a DynamicBuffer which is stored in the recorded command, and the contents of this buffer will be copied to the entity's actual buffer when it is created in playback. Effectively, writing to the returned buffer allows you to set the initial contents of the buffer component.
- SetBuffer(): Like AddBuffer(), but it assumes the entity already has a buffer of the component type. In playback, the entity's already existing buffer content is overwritten by the contents of the returned buffer.

  Note: Some EntityManager methods have no EntityCommandBuffer equivalent because an equivalent wouldn't be feasible or make sense. For example, there are no EntityCommandBuffer methods for getting component values because reading data is not something that can be usefully deferred.

  Note: After it has been played back, an EntityCommandBuffer instance cannot be used for additional recording. If you need to record more commands, create a new, separate EntityCommandBuffer instance.

Each EntityCommandBuffer has a job safety handle, so the safety checks will throw an exception if you:

- ...invoke the EntityCommandBuffer's methods on the main thread while the EntityCommandBuffer is still in use by any currently scheduled jobs.
- ... or schedule a job that accesses an EntityCommandBuffer already in use by other currently scheduled jobs (unless the new job depends on those other jobs).

  Important: You might be tempted to share a single EntityCommandBuffer instance across multiple jobs, but this is strongly discouraged. There are cases where it will work

> fine, but in many cases it will not. For example, using the same EntityCommandBuffer.ParallelWriter across multiple parallel jobs might lead to an unexpected playback order of the commands. Instead, it's virtually always best to create and use one EntityCommandBuffer per job. Don't worry about a performance difference: recording and playing back a set of commands split across multiple EntityCommandBuffer's is not really any more expensive than recording the same set of commands all into one EntityCommandBuffer.

Temporary entity IDs

When you call the CreateEntity() or Instantiate() methods of an EntityCommandBuffer, no new entity is created until the command is executed in playback, so the entity ID returned by these methods are *temporary ID's*, which have negative index numbers. Subsequent AddComponent, SetComponent, and SetBuffer commands of the same EntityCommandBuffer may use these temporary ID's. In playback, any temporary ID's in the recorded commands will be remapped to actual, existing entities.

> **Important**: Because a temporary entity ID has no meaning outside of the EntityCommandBuffer instance from which it was created, a temporary entity ID should only be used in subsequent method calls of the same EntityCommandBuffer instance. For example, do not use a temporary ID you get from one EntityCommandBuffer when recording a command to a different EntityCommandBuffer instance.

EntityCommandBuffer.ParallelWriter

To safely record commands from a parallel job, we need an EntityCommandBuffer.ParallelWriter, which is a wrapper around an underlying EntityCommandBuffer. A ParallelWriter has most of the same methods as an EntityCommandBuffer itself, but the ParallelWriter methods all take an additional 'sort key' argument for the sake of determinism.

When an EntityCommandBuffer.ParallelWriter records commands in a parallel job, the order in the buffer of the commands recorded from different threads depends upon thread scheduling, making the order non-deterministic. This isn't ideal because:

- Deterministic code is generally easier to debug.
- Some netcode solutions depend upon determinism to produce consistent results across different machines.

While the recording order of the commands cannot be deterministic, the playback order can be deterministic with a simple trick:

1. Each command records a 'sort key' integer passed as the first argument to each command method.
2. The Playback() method sorts the commands by their sort keys before executing the commands.

As long as the sort key values map deterministically to each recorded command, the sort makes the playback order deterministic.

In an IJobEntity, the sort key we generally want to use is the ChunkIndexInQuery, which is a unique value for every chunk. Because the sort is stable and because all entities of an individual chunk are processed together in a single thread, this index value is suitable as a sort key for the recorded commands. In an IJobChunk, we can use the equivalent unfilteredChunkIndex parameter of the Execute method.

Multi-playback

If an EntityCommandBuffer is created with the PlaybackPolicy.MultiPlayback option, its Playback method can be called more than once. Otherwise, calling Playback more than once will throw an exception. Multi-playback is mainly useful when you want to repeatedly spawn a set of entities.

EntityCommandBufferSystem

An EntityCommandBufferSystem is a system that provides a convenient way to defer EntityCommandBuffer playback. An EntityCommandBuffer instance created from an EntityCommandBufferSystem will be played back and disposed the next time the EntityCommandBufferSystem updates.

> **Important**: Do not manually play back and dispose an EntityCommandBuffer instance created by an EntityCommandBufferSystem: the EntityCommandBufferSystem will both play back and dispose the instance for you.

You rarely need to create any EntityCommandBufferSystems yourself because the automatic bootstrapping process puts these five into the default world:

- BeginInitializationEntityCommandBufferSystem
- EndInitializationEntityCommandBufferSystem

- BeginSimulationEntityCommandBufferSystem
- EndSimulationEntityCommandBufferSystem
- BeginPresentationEntityCommandBufferSystem

The EndSimulationEntityCommandBufferSystem, for example, is updated at the end of the SimulationSystemGroup.

> **Note**: There's no “EndPresentationEntityCommandBufferSystem” at the end of the frame, but you can use BeginInitializationEntityCommandBufferSystem instead: the end of one frame and the beginning of the next are logically the same point in time.

Transform components and systems

The LocalTransform is the main standard component that represents the transform of an entity. Transform hierarchies can be formed with three additional components:

- The Parent component stores the id of the entity's parent.
- The Child dynamic buffer component stores the ids of the entity's children.
- The PreviousParent component stores a copy of the id of the entity's parent.

To modify the transform hierarchy:

- Add the Parent component to parent an entity.
- Remove an entity's Parent component to de-parent it.
- Set an entity's Parent component to change its parent.

The ParentSystem will ensure that:

- Every entity with a parent has a PreviousParent component that references the parent.
- Every entity with one or more children has a Child buffer component that references all of its children.

> **Important**: Although you can safely read an entity's Child buffer and PreviousParent components, you should not modify them directly. You should only modify the transform hierarchy by setting the entities' Parent components.

Every frame, the LocalToWorldSystem computes each entity's world-space transform (from the LocalTransform components of the entity and its ancestors) and assigns it to the entity's LocalToWorld component.

> **Note**: The Entity.Graphics systems read the LocalToWorld component but not any of the other transform components, so LocalToWorld is the only transform component an entity needs to be rendered.

Baking and entity scenes

Baking is a build-time process that creates *entity scenes* from *sub scenes* by executing *bakers* and *baking systems*:

- An **entity scene** is a serialized set of entities and components that can be loaded at runtime.
- A **sub scene** is a Unity scene asset that's embedded in another scene by the SubScene MonoBehaviour.
- A **baker** is a class extending Baker<T>, where T is a MonoBehaviour. A MonoBehaviour with a Baker is called an “authoring component”.
- A **baking system** is a system marked with the [WorldSystemFilter(WorldSystemFilterFlags.BakingSystem)] attribute. (Baking systems are generally only required for advanced use cases.)

A sub scene is baked in a few main steps:

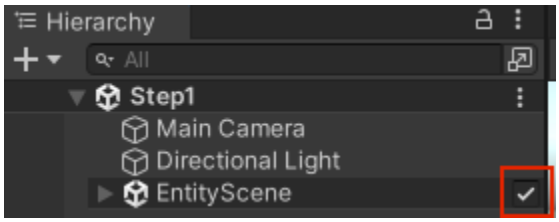
1. For each GameObject of the sub scene, a corresponding entity is created.
2. The baker of each authoring component in the sub scene is executed. Each baker can read the authoring component and add components to the corresponding entity.
3. The baking systems are executed. Each system can read and modify the baked entities. Unlike bakers, baking systems should not access the original GameObjects of the sub scene.

When modified, a sub scene is re-baked:

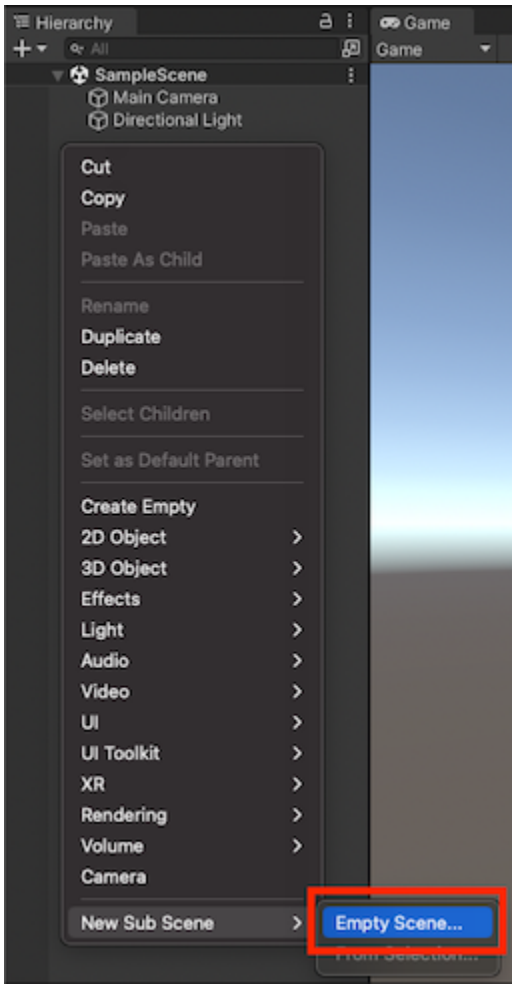
1. Only the bakers of the modified authoring components are re-executed.
2. The baking systems are always re-executed in full.
3. The live entities in Edit mode or Play mode are updated to match the results of baking. (This is possible because baking tracks the correspondence of baked entities to live entities.)

Creating and editing sub scenes

A GameObject with the SubScene MonoBehaviour has a checkbox that opens and closes the sub scene for editing. While a sub scene is open, its GameObjects are loaded and take up resources in the Unity editor, so you may want to close large sub scenes that you're not currently editing.



The convenient way to create a new sub scene is to right click within the Hierarchy window and select *New Subscene > Empty Scene....* This creates both a new scene file and a GameObject with a SubScene component that references the new scene file:



Accessing data in a baker

Incremental baking requires bakers to track all the data they read. The fields of a baker's authoring component are automatically tracked, but other data read by a baker must be added to its list of dependencies by Baker methods:

- GetComponent<T>(): Accesses a component of a GameObject in the sub scene.
- DependsOn(): Declares that an asset should be tracked for this baker.
- GetEntity(): Returns the id of an entity baked in the sub scene or baked from a prefab. (The entity will not yet have been fully baked, so you should not attempt to read or modify the components of the entity through this id.)

Loading and unloading entity scenes

For streaming purposes, the entities of a scene are split into sections identified by index number. Which section an entity belongs to is designated by its SceneSection shared component. By default, an entity belongs to section 0, but this can be changed by setting SceneSection during baking.

> **Important**: During baking, entities in a sub scene can only reference other entities of the same section or section 0 (which is a special case because section 0 is always loaded before the other sections and only unloaded when the scene itself is unloaded).

When a scene is loaded, it is represented by an entity with metadata about the scene, and its sections are also each represented by an entity. An individual section is loaded and unloaded by manipulating its entity's RequestSceneLoaded component: the SceneSectionStreamingSystem in the SceneSystemGroup will respond when this component is changed.

To load and unload entity scenes from your code, use static methods of the SceneSystem:

- LoadSceneAsync(): Initiates loading of a scene. Returns an entity that represents the loaded scene.
- LoadPrefabAsync(): Initiates loading of a prefab. Returns an entity that references the loaded prefab.
- UnloadScene(): Destroys all entities of a loaded scene.
- IsSceneLoaded(): Returns true if a scene is loaded.
- IsSectionLoaded(): Returns true if a section is loaded.
- GetSceneGUID(): Returns the GUID representing a scene asset (specified by its file path).
- GetScenePath(): Returns the path of a scene asset (specified by its GUID).
- GetSceneEntity(): Returns the entity representing a scene (specified by its GUID).

> **Important**: Entity scene and section loading is always asynchronous, and there is no guarantee of how long it will take for the data to be loaded after the request. In most cases, code should check for the presence or absence of specific data loaded from scenes rather than check the loading status of the scenes themselves. This approach avoids tethering code to particular scenes: if the data is moved to a different scene, downloaded from the network, or procedurally generated, the code will still work without modification.

Additional features

Managed IComponentData components

A class implementing IComponentData is a managed component type. Unlike IComponentData structs, which are unmanaged, these managed components can store any managed objects.

In general, managed component types should be used only when really needed because, compared to unmanaged components, they incur some heavy costs:

- Like all managed objects, managed components cannot be used in Burst-compiled code.
- Managed objects cannot normally be used safely in jobs.
- Managed components are not stored directly in the chunks: instead, all managed components of a world are all stored in one big array while the chunks merely store indexes of this array.
- Like all managed objects, creating managed components incurs garbage collection overhead.

To ensure that any resources contained by a managed component get copied when the component itself is copied, the component should implement ICloneable. To ensure that any resources contained by a managed component are properly disposed of when the managed component is destroyed, the component should implement IDisposable.

Enableable components

A struct implementing IComponentData or IBufferElementData can also implement IEnableableComponent. A component type implementing this interface can be enabled and disabled per entity.

When a component of an entity is disabled, queries consider the entity to not have the component type:

- In SystemAPI.Query(), IJobEntity, and other contexts that iterate through the entities of a chunk matched by a query, individual entities that don't match the query because of their components' enabled or disabled state will be skipped.
- If no entities in a chunk match the query because of the components' enabled or disabled state, that chunk will not be included in the array returned by the ToArchetypeChunkArray() method of EntityQuery.

Be clear that disabling a component does *not* remove or modify the component: instead, a bit associated with the specific component of the specific entity is toggled. Also be clear that a disabled component only affects queries: a disabled component can otherwise still be read and modified as normal, such as *via* the GetComponent<T>() method of EntityManager.

All enableable components are enabled by default when added to an entity. When an entity is copied for serialization, copied to another world, or copied by the Instantiate method of EntityManager, the enabled states of the components are copied as well.

The enabled state of an entity's components can be checked and set through:

- EntityManager
- ComponentLookup<T>
- BufferLookup<T>
- EnabledRefRW<T>
- ArchetypeChunk

For instance, the EntityManager includes these methods:

- IsComponentEnabled<T>(): Returns true if an entity has a currently enabled T component.
- SetComponentEnabled<T>(): Enables or disables an entity's enableable T component.

  Note: For the sake of the job safety checks, read or write access of a component's enabled state requires read or write access of the component type itself.

In an IJobChunk, the Execute method parameters signal which entities in the chunk match the query:

- If the useEnableMask parameter is false, all entities in the chunk match the query.
- Otherwise, if the useEnableMask parameter is true, the bits of the chunkEnabledMask parameter signal which entities in the chunk match the query, factoring in all enableable component types of the query. Rather than check these mask bits manually, you can use a ChunkEntityEnumerator to more conveniently iterate through the matching entities.

> **Note**: The chunkEnabledMask is a composite of all the enabled states of the enableable components included in the query of the job. To check enabled states of individual components, use the IsComponentEnabled() and SetComponentEnabled() methods of the ArchetypeChunk.

Aspects

An aspect is an object-like wrapper over a subset of an entity's components. Aspects can be useful for simplifying queries and component-related code. For example, we could define a “MonsterAspect” that groups together the components that comprise a monster entity.

An aspect is defined as a readonly partial struct implementing IAspect. The struct can contain fields of these types:

- Entity: The wrapped entity's entity ID.
- RefRW<T> or RefRO<T>: A reference to the wrapped entity's T component.
- EnabledRefRW<T> and EnabledRefRO<T>: A reference to the enabled state of the wrapped entity's T component.
- DynamicBuffer<T>: The wrapped entity's dynamic buffer T component.
- Another aspect type: The containing aspect will encompass all the fields of the 'embedded' aspect.

Including an aspect in a query is the same as including all the individual components wrapped by the aspect.

These EntityManager methods create instances of an aspect:

- GetAspect<T>: Returns an aspect of type T wrapping an entity.
- GetAspectRO<T>: Returns a readonly aspect of type T wrapping an entity. A read-only aspect throws an exception if you use any method or property that attempts to modify the underlying components.

Aspect instances can also be retrieved by SystemAPI.GetAspectRW<T> or SystemAPI.GetAspectRO<T> and accessed in an IJobEntity or a SystemAPI.Query() loop.

> **Important**: You should generally get aspect instances *via* SystemAPI rather than the EntityManager: unlike the EntityManager methods, the SystemAPI methods register the underlying component types of the aspect with the system, which is necessary for the systems to properly schedule jobs with every dependency they need.

Shared components

For a *shared component* type, all entities in a chunk share the same component value rather than each entity having its own value. Consequently, setting a shared component value of an entity performs a structural change: the entity is moved to a chunk which has the new value. For example, if an entity has a *Foo* shared component value X, then the entity is stored in a chunk that has *Foo* value X; if the entity is then set to have *Foo* value Y, the entity is moved to a chunk that has value Y; if no such chunk already exists, a new chunk is created.

The primary utility of shared components comes from the fact that queries can filter for specific shared component values. For example, a query that includes shared component *Foo* can include a filter that specifies it should match only entities with *Foo* value X.

Instead of storing shared component values directly in chunks, the world stores them in a set of arrays, and the chunks store just indexes into these arrays. This means that each unique shared component value is stored only once within a world.

> **Note**: How a shared component type is compared for equality to determine uniqueness can be customized by implementing IEquatable<T> for the type.

A shared component type is declared as a struct implementing ISharedComponentData. If the struct contains any managed type fields, then the shared component will itself be treated as a managed component type, with the same advantages and restrictions as a managed IComponentData.

The EntityManager has these key methods for shared components:

- AddComponent<T>(): Adds a T component to an entity, where T can be a shared component type.
- AddSharedComponent(): Adds an unmanaged shared component to an entity and sets its initial value.
- AddSharedComponentManaged(): Adds a managed shared component to an entity and sets its initial value.
- RemoveComponent<T>(): Removes a T component from an entity, where T can be a shared component type.
- HasComponent<T>(): Returns true if an entity currently has a T component, where type T can be a shared component type.
- GetSharedComponent<T>(): Retrieves the value of an entity's unmanaged shared T component.
- SetSharedComponent<T>(): Overwrites the value of an entity's unmanaged shared T component.
- GetSharedComponentManaged<T>(): Retrieves the value of an entity's managed shared T component.

- SetSharedComponentManaged<T>(): Overwrites the value of an entity's managed shared T component.

  Important: Because the EntityManager relies upon equality to identify unique and matching shared component values, you should avoid modifying any mutable objects referenced by shared components. For example, if you want to modify an array stored in a shared component of a particular entity, you should not modify the array directly but instead update the component of that entity to have a new, modified copy of the array.

  Important: Having too many unique shared component values may result in chunk fragmentation! Because all entities in a chunk must share the same shared component values, if you give unique shared component values to a high number of entities, the entities will end up fragmented across many chunks. For example, if there are 500 entities of an archetype with a shared component and each entity has a unique shared component value, each entity is stored by itself in a separate chunk. This wastes most of the space in each chunk and also means that looping through all entities of the archetype requires visiting 500 chunks. This fragmentation largely negates the performance benefits of the ECS structure. To avoid this problem, try to use as few unique shared component values as possible. If, say, the 500 entities were to share only ten unique shared component values, they could be stored in as few as ten chunks.

Cleanup components

Cleanup components are special in two ways:

- When an entity with cleanup components is destroyed, the non-cleanup components are removed, but the entity actually continues to exist until you remove all of its cleanup components individually.
- When an entity is copied to another world, copied in serialization, or copied by the Instantiate method of EntityManager, any cleanup components of the original are not added to the new entity.

The primary use case for cleanup components is to help initialize entities after their creation or cleanup entities after their destruction. For example, say we have entities representing monsters which all have a *Monster* tag component:

1. We can find all monster entities needing initialization by querying for all entities which have the *Monster* component but which do not have a *MonsterCleanup* component. For all entities matching this query, we perform any required initialization and add *MonsterCleanup*.
2. We can find all monster entities needing cleanup by querying for all entities which have the *MonsterCleanup* component but not the *Monster* component. For all entities

matching this query, we perform any required cleanup and remove *MonsterCleanup*. Unless the entities have additional remaining cleanup components, this will destroy the entities.

> **Note**: In some cases, you'll want to store information needed for cleanup in your cleanup components, but in many cases, an empty cleanup tag component is sufficient.

Cleanup components come in four varieties:

- A struct implementing ICleanupComponentData: The cleanup variant of an unmanaged IComponentData type.
- A class implementing ICleanupComponentData: The cleanup variant of a managed IComponentData type.
- A struct implementing ICleanupBufferElementData: The cleanup variant of a dynamic buffer type.
- A struct implementing ICleanupSharedComponentData: The cleanup variant of a shared component type.

Chunk components

A *chunk component* is a single value belonging to a chunk.

> **Note**: Shared components also store one value per chunk, but a shared component value logically belongs to the entities, not the chunk (which is why setting an entity's shared component value moves the entity to another chunk rather than modifying the value stored in the chunk). Chunk components truly belong to the chunk itself, and unlike unmanaged shared components, unmanaged chunk components are stored directly in the chunk.

A chunk component is defined as a struct or class implementing IComponentData, but a chunk component is added, removed, get, and set with these EntityManager methods:

- AddChunkComponentData<T>: Adds a chunk component of type T to a chunk, where T is a managed or unmanaged IComponentData.
- RemoveChunkComponentData<T>: Removes a chunk component of type T from a chunk, where T is a managed or unmanaged IComponentData.
- HasChunkComponent<T>: Returns true if a chunk has a chunk component of type T.
- GetChunkComponentData<T>: Retrieves the value of a chunk's chunk component of type T.
- SetChunkComponentData<T>: Sets the value of a chunk's chunk component of type T.

Blob assets

A Blob (Binary Large Object) asset is an immutable (unchanging), unmanaged piece of binary data stored in a contiguous block of bytes:

- Blob assets are efficient to copy and load because they are fully relocatable: all internal pointers are expressed as relative offsets instead of absolute addresses, so copying the whole Blob is as simple as copying every byte.
- Although they are stored independently from entities, Blob assets may be referenced from entity components.
- Because they're immutable, Blob assets are inherently safe to access from multiple threads.

> **Note**: The name Blob "asset" is a bit misleading: a Blob asset is a piece of data in memory, not a project asset file! However, Blob assets are efficiently and easily serializable into files on disk, so in that sense, calling them "assets" is appropriate.

To create a Blob asset:

1. Create a BlobBuilder.
2. Call the builder's ConstructRoot<T> to set the Blob's 'root' (a struct of type T).
3. Call the builder's Allocate<T>, Construct<T> and SetPointer<T> methods to fill in the rest of the Blob data (including BlobArrays, BlobStrings, and BlobPtr).
4. Call the builder's CreateBlobAssetReference, which copies all the data in the builder to create the actual Blob asset and returns a BlobAssetReference.
5. Dispose the BlobBuilder.

When a Blob asset is no longer needed, it should be disposed of by calling Dispose on the BlobAssetReference.

Blob assets referenced in a baked entity scene are serialized and loaded along with the scene. These Blob assets should not be manually disposed: they will be automatically disposed of along with the scene.

> **Important**: All parts of a blob asset that contain internal pointers must always be accessed by reference. For example, the offset values in a BlobString struct are only correct relative to where the BlobString struct is stored inside the Blob; the offsets are not correct relative to copies of the struct.

Version numbers

A world, its systems, and its chunks maintain several 'version numbers' (numbers which are incremented by certain operations). By comparing version numbers, you can determine if certain data might have changed.

All version numbers are 32-bit signed integers, so when incremented, they may eventually wrap around in the lifetime of the program. The proper way to compare version numbers then relies upon subtle quirks of how C# defines signed integer overflow:

```java
// true if VersionB is more recent than VersionA
// false if VersionB is equal or less recent than VersionA
bool changed = (VersionB - VersionA) > 0;
```

The version numbers include:

- World.Version: A version number that's increased every time the world adds or removes a system or system group.
- EntityManager.GlobalSystemVersion: A version number that's increased before every system update in the world.
- SystemState.LastSystemVersion: A version number of the system that's assigned the value of the GlobalSystemVersion immediately after each time the system updates.
- EntityManager.EntityOrderVersion: A version number that's increased every time a structural change is made in the world.
- Each component type has its own version number, which is incremented by any operation that gets write access to the component type. This number can be retrieved by calling the method EntityManager.GetComponentOrderVersion.
- Each shared component value also has a version number that is increased every time a structural change affects a chunk having the value.
- A chunk stores a version number for each component type in the chunk. When a component type in a chunk is accessed for writing, its version number is assigned the value of EntityManager.GlobalSystemVersion, regardless of whether any component values are actually modified. These chunk version numbers can be retrieved by calling the ArchetypeChunk.GetChangeVersion method.
- A chunk also stores a version number which is assigned the value of EntityManager.GlobalSystemVersion every time a structural change affects the chunk. This version number can be retrieved by calling the ArchetypeChunk.GetOrderVersion method.