TilePlus Services

Overview

In the TilePlus system, Runtime-only Scriptable Objects are called Services or SRS, and are controlled by TpServiceManager.

Services are built on a base class called ScriptableRuntimeService, which handles automatically pre-registering the service with TpServiceManager prior to the first scene load.

It's easy to create your own, just inherit from ScriptableRuntimeService and add your own data or code. Be sure to call the base classes in OnEnable and OnDisable if you override them.

Ensure that you have something like this in your derived class:

```
[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
private static void InitOnLoad()
{
    TpServiceManager.PreRegisterSrs(  nameof(YOUR_CLASS), Create);
}
```

The `BeforeSceneLoad` version of the attribute is REQUIRED. If omitted, the service won't be available until this specific service actually registers itself.

TpServiceManager internally keeps Services in two groups:

- Available: Services which have pre-registered themselves using the method shown above.
- Running: Available Services which have been loaded.

TpServiceManager

TpServiceManager has properties and methods to deal with services:

Properties

- GetAllRunningServices: an array of ScriptableObjects that are running services.
  - This is editor-only and used for in-editor diagnostics like the ServicesInspector and System Info windows.
- GetAllRunningServiceTypes: an array of running Service Types (c# Types)
- GetAllAvailableServiceTypes: an array of available service name strings.

Methods

- `HasServiceOfTypeAvailable`: input is a string e.g., nameof(TpSpawner). Returns true if the specified service is AVAILABLE (may not be running).
- `HasAllTheseRunningServices`: Test for a set of active services. Input is an IEnumerable of Types. Returns true if ALL the services are running.
- `GetServiceOfType<T>`: One way of obtaining a service handle.
  - Example: `GetServiceOfType<TpSpawner>();`
  - If the service is NOT running this method will start it.
- `GetRunningService<T>(out T? Service)`: Obtaining a Service handle ONLY if it's already running.
  - Example: `if(GetRunningService<TpSpawner>(out var spawner){...}`
  - Returns true if there's an already-running service of Type T (and the out param has the handle)
  - Returns false if there isn't an already-running service of Type T (and the out param is Null)
  - Unlike GetServiceOfType<T>, the service isn't started if it isn't already running.

GetServiceOfType is quite simple:

- ALWAYS fails unless the Editor is playing or about to change into Play mode.
- Does a fast dictionary lookup to see if the service is already running.
  - If it is, return the service's handle (instance).
  - If it isn't, see if it's available (pre-registered)
    - If it isn't: fail (return NULL).
    - If it is, create the Service instance.
  - If Service Instance was already active or newly created, return that. Otherwise return null.

GetRunningService<T> is also ONLY for use when in Play mode.

Finally, if you want to terminate a service, use:

`TerminateServiceOfType<T>()`

If the service is running this will Destroy the service. Normally, services persist throughout the lifetime of the application once they're loaded. In the Layout demo the Dialog Box seen when you first encounter a treasure chest is actually a Bundle loaded by the `DialogBoxService` and this service is terminated when the Dialog box close button is clicked.

Terminating a running service doesn't mean it's no longer available: you can just use GetServiceOfType to reload it.

Initialization

It's important to avoid race conditions for Services which require initialization. This can occur if more than one source (such as a Component) use the Service.

For example, in the TopDownLayout demo, both the GameController and the TpInputActionToTile components both use the TilePositionDb service. The PositionDb is designed to reject duplicate 'monitored' Tilemaps (i.e., those maps where the PositionDb keeps track of tile positions). Other control parameters should only be modified by one entity. In this demo, that's TpInputActionToTile.

Note that this warning ONLY applies to Services which require initialization. For TilePlus, that's only the TilePositionDb

Convenience Properties

Since the Messaging, Tween, and Spawning services are commonly used, shortcut properties exist that just let you get the handle:

- SpawnerService returns a handle to TpSpawner.
- TweenerService returns a handle to TpTweener.

These use `GetServiceOfType<T>` and will auto-start the service if not already running. These services do not require any initialization.

Note that the TpTweener Service is also available via a protected static (i.e., within the class or derived classes only) property in the TilePlusBase class (which all TilePlus tiles derive from). This property is an alias for TpLib.TweenerService.

- A Monobehaviour or other code can use TweenerService (via TpLib.TweenerService) to create GameObject tweens.
- Creating Tile Tweens requires a TilePlusBase instance as part of the method call.

Please refer to the Tweener documentation for more information.

Services as Singletons

Usually a service is a front-end for something that needs to be defined by an API and/or an Interface, and in that sense, they're by nature singletons at least from an external point of view. Internally a service may handle multiple instances of something else but that's mostly hidden from code that uses the service.

That's the general model for this simple, but efficient, service scheme.

The ScriptableRuntimeService class can actually have multiple instances: there's no static 'Instance' at all.

It's the implementation used in TpServiceManager that enforces only one instance of each service for two main reasons:

- Lookup speed: simpler data structures.
- Multiple instances are harder to differentiate in your code.

Other systems in TilePlus need multiple Scriptable Object instances. Specifically, the Layout System which creates multiple instances of TpZoneManager Scriptable Objects. Here, the different instances are differentiated by their names. It's quite a bit more complex although the details are largely hidden via the use of Monobehaviour Components for setting up parameters.

Hence, Services are forced to be singletons as an implementation detail for this particular type of dynamically loadable service.

The intent is to use Services for, well, Services and not global data storage or state.

The Tweener, Messaging, Spawning, and PositionDb services are autonomous and don't depend on any external state aside from that being provided from internal Unity API interactions, and TpLib logging and 'Tilemap DB' methods. All their internal data are private and only accessible via methods or properties. The only significant dependency occurs when a Service requires an Update method invocation - that's minor and not a Service-to-Service dependency.

Creating services which have cross-dependencies is a bad idea and you will be plagued with bugs unless you are extremely careful.

Back to Singletons...

But there's nothing stopping you from using Services as (sort of) conventional singletons if you want to. See the LayoutSystem demo for an example.

- In that example one finds a 'GameState' service, which holds global state including the data which are saved when code in the example requests a game save.
- This approach was taken to keep the example simple (it's complex enough as it is).
- Depending on your viewpoint about Singletons, you may or may not want to avoid this approach.

Profiling

Services that require updating can set this up via the IScriptableService interface, discussed below. The update invocation for each service is automatically wrapped with Profiler.BeginSample and Profiler.EndSample regions with the Service's name. When using a modified PlayerLoop for updates, look for the TpLib section.

Messaging

Use `ServiceMessage(ObjectPacket packet)` to send messages to running Services. This is implemented with two IScriptableService members (see below).

Use a pooled (wrapped with `using`) or an an unpooled instance of `ObjectPacket` and set the Command property to a particular command string. Other properties of can be used to send whatever information the target understands (this is up to you).

Using the instance's properties, you can add one or more of:

- a UnityEngine.Object
- any instance of any c# class
- a string
- an int
- a boolean value
- all of the above.

Services indicate which command strings are supported.

When ServiceMessage is used, the packet's Command is examined and is used to determine which Services get the message. This is very flexible and open-ended. It's not used by the TilePlus system at all, but is used in the game that this was developed for, which is largely based on services.

More Grisly Details

The IScriptableService interface

IScriptable is an interface that lets a SRS specify how it wants to receive Update events (or not) and if it should persist across Unity scene changes.

If a service DOES NOT implement IScriptableService it inherits the default Interface properties or; no Updates and auto-kill on scene changes.

Not all Services need an Update event; for example, TpMessaging and TpSpawner. Some Services need an Update event; for example, TpTileTweener and TpTilePositionDb.

Services which require Update events must implement the following:

```
bool IScriptableService.WantsUpdate => true;
bool IScriptableService.ReadyForUpdate => true;
void IScriptableService.Update(float deltaTime){}
```

Note that these are EXPLICIT interface implementations and MUST be done this way.

- WantsUpdate is only examined twice: when the Service registers itself when loaded and when it deregisters itself when unloaded (which may not be till program shutdown).
  - WantsUpdate should NEVER be a variable. If this value is different for registration and deregistration then an exception will occur during the deregistration code or soon after, at the next TpLib internal update.
- ReadyForUpdate is examined every Update and doesn't always have to be true. If your code doesn't always need Update or if it wants to delay enabling Update for some reason then the property can return the value of a variable or test some condition. The Tweener uses this feature to only get Update events when there is actually work to do.
  - In words: Update is called every frame unless ReadyForUpdate is false.

Persist Across Scene Changes

If the `PeristThruReload` interface property is EXPLICITLY implemented and returns true then this Service will not be destroyed when a Unity scene changes.

IScriptable messaging

As mentioned above, messages can be sent to Services via ObjectPackets. The packet's Command property controls what Services are sent a particular message.

To do this, implement `AcceptableMessages` and `MessageTarget.` AcceptableMessages is a property that returns an array of strings representing the Commands that the MessageTarget can accept.

AcceptableMessages is examined only when the Service starts.

- Returning new string[1]{"XYZZY"} would mean messages with XYZZY as the command are accepted.
- Returning new string[2]{"XYZZY", "ABCDEF"} would mean messages with XYZZY or ABCDEF as commands are accepted.
- Returning Empty array (the default value of the property) == NO messages accepted.
- Returning single-item array with first item == `"__ALL__"` => ALL messages.
- AVOID: Returning new string[2]{`"__ALL__"`, "ABCDEF"} would mean ALL messages + all ABCDEF messages
  - ABCDEF messages would be sent to the service twice! (so don't do this).

Update Event Timing for Services

Note that Update timing isn't the same as Monobehaviour Update and doesn't originate from a GameObject in any scene. The modified PlayerLoop from TpLibTasks updates services at PostLateUpdate.

Revision #48
Created 2025-06-20 17:09:35 UTC by Vonchor
Updated 2026-07-09 15:06:19 UTC by Vonchor