

Godot project architecture

Application Scopes

In GFF, an application scope is an entry point available during a specific period of time in the application lifetime.

Application is structured within two scopes:

- **App** (`res://sources/App.cs`): The **App** scope, initialized at the start of the application, released before quitting. App act like a singleton.
- **Game** (`res://sources/Game.cs`): The **Game** scope, initialized when starting or loading a game, released when going back to main menu.

Each scope contains `Services` that can be accessed from anywhere within the scope. Services are declared in the `RegisterServices` method of an application scope by filling the dependency builder with instances.

Current rendering driver is accessible in static field `ApplicationScope.RenderingDriver`. It can be checked to enable some compatibility features.

Application Version

Current version of the application can be accessed from the static field `ApplicationScope.Version`. It is modified when using the command `gff release-version` (you can also modify it manually in the `csproj` file).

You may need to execute some code when moving from a version to another (migrate some settings, checking for unlock of newly created achievements, etc ...). You can do this by registering a delegate using

`ApplicationScope.RegisterVersionMigrationCallback(ApplicationVersionChangedEventHandler migrationCallback)`. Registered delegate will be called only the first time after a version change.

App Scope

The App scope is the first entry point of the Godot application. It's constructor parse command line arguments so they are accessible during the whole execution of the application (`App.Args`). They are parsed by the [CommandLine](#) library.

It also have a helper to checks for the availability of a feature (method `HasFeature(string tagName)`). This method replace Godot's helper `OS.HasFeature`. In Debug mode, this method allows developers to force features (e.g., "demo", "steam") to report as available by checking specific arguments, overriding Godot feature status.

Services

The **service locator pattern** is a design pattern used in software development to encapsulate the processes involved in obtaining a service with a strong abstraction layer. This pattern uses a central registry known as the "service locator", which on request returns the information necessary to perform a certain task.

The service locator pattern provides global access to services without the testability issues of singletons. This pattern allows swapping implementations for testing or different platforms while maintaining a simple API for service access.

The API allows to declare services like this:

```
// Register a service by giving its type. A new instance will automatically be created.
dependencyBuilder.AddSingleton<Application.SaveManager>()

// Abstract a service behind an interface. You can get the instance using the interface type.
// This allows to change the instance depending on some conditions without having to change
the code on the user side.
if (App.HasFeature("steam"))
{
    dependencyBuilder.AddSingleton<IDrmService, Application.Store.SteamManager>();
}
else
{
    dependencyBuilder.AddSingleton<IDrmService, Application.Store.NoDrmManager>();
}

// It's also possible to manually instantiate the service (useful when having specific
instanciation rules).
```

```
dependencyBuilder.AddSingleton<Application.AppSettings>(new
Application.AppSettings(this.GetTree().Root))

// You can also provide instances from the scene tree to make them globally accessible.
dependencyBuilder.AddSingleton<Core.UI.LoadingScreen>(this.GetChildren<LoadingScreen>().First(
))
```

Services are then accessible from anywhere within the scope of its service locator (the Application Scope):

```
AppSettings appSettings = App.Services.Get<AppSettings>();
IDrmService drmService = App.Services.Get<IDrmService>();
```

A service must implement the interface `ISingleton<TServiceLocator>` and must declare a method named `Bind` that will receive as parameters other services that it depends on. This method is called by the dependency builder using dependency injection to fill parameters.

```
// Declare IDrmService parameter to create a dependency to this service.
public void Bind(IDrmService drmService)
{
    // Then, use drmService to initialize the service...
}
```

There is 3 methods in `ISingleton` interface, `Initialize` and `Release` are called when the Application Scope start / exit. `Update` is called every frame. To ensure a specific order of call, you can set a dependency between two services.

App State

The execution flow of the application is controlled by a [hierarchical state machine](#) (HSM). It is registered as a service in the App scope (`res://sources/application/AppState.cs`). HSM allow you to easily add new states without having to break your existing state machine. You can control general behavior using parent states, and more specific one with child states. You can add any number of hierarchy level you need.

Another HSM is used to control game execution flow (`res://sources/gameplay/GameState.cs`).

Revision #8

Created 2026-07-13 08:52:07 UTC by Adrien Allard

Updated 2026-07-20 07:14:08 UTC by Adrien Allard