SharpMedia Services Guidelines

© 2007 Bojan Šernek – SharpMedia project, document version: DRAFT

Table Of Contents

[About 2](#_Toc168322838)

[Goals 2](#_Toc168322839)

[Different Service Architectures 2](#_Toc168322840)

[Single Instance Service 2](#_Toc168322841)

[Multiple instances with Views 3](#_Toc168322842)

[Multiple instances with Views and Proxies 4](#_Toc168322843)

[Services with Wrapper APIs 5](#_Toc168322844)

[Choosing the best Strategy 6](#_Toc168322845)

[Preparing for an Implementation 7](#_Toc168322846)

[Using Services 8](#_Toc168322847)

[Lifetime Services 8](#_Toc168322848)

# About

These guidelines will outline the usage patterns and coding guidelines in creating, using and manipulation SharpMedia kernel-level services. Services are server/daemon like processes that serve specific interface types for consummation by other processes.

# Goals

By the end of this document, you should be able to:

* Understand how service architecture works
* Understand how to ask the OS for a service
* Write code that effectively uses a service
* Write code that serves a simple interface to everyone
* Write code that serves multiple interfaces
* Write code that does access policy checking and process level scoping in a service
* Architecture and design systems that depend on services

# Different Service Architectures

The SharpMedia OS exposes a simple API through which clients may query for and use instances of outside interfaces from entities known as services. To use these features of the OS, the enquiring process (client) should use the ***OS.ServiceLocator*** static property or ***OS.ActivateService<interface>*** static method.

## Single Instance Service

The simplest services will serve a single instance of an implementation to all clients. Such an example would be a string echo service – it does not need to perform any client specific actions, and is therefore oblivious to the identity of the caller (client). See .

Figure - Multiple clients sharing a single service instance

## Multiple instances with Views

More complex services may require caller dependent processing – that is, their implementation will differ based on the identity of the caller. A simple example would be a service that reads data from a storage device – if the caller has the security identity that allows for reading that device, the service will allow its reading and will not if the caller does not have the necessary security capabilities.

A variation of the previous example is when we want to use caller identity not because we want to enforce security measures, but because we want to scope data based on the caller identity. An example is a service that returns a fortune ([Fortune (Wikipedia)](http://en.wikipedia.org/wiki/Fortune_(program))) based on the name of the caller.

Now, Instead of serving a single instance of the service (as was the case in the first example), we need service instances that remember their owner (process). These are called Views, and the reason for that is that they represent views of the service from the perspective of the calling process. Internally, all of the Views may implement their own logic for actually servicing requests, or just forward their requests to a single implementation instance. See .

Usually, the interface between a view and the actual implementation need not be the same as the interface displayed to the user. We can imagine that regarding the previous example, the view would expose a property ***string*** ***Fortune { get; }***. The actual implementation backend would expose a method ***string GetFortune(string processName)***.

Figure - Each process communicates with the root service object through a view object

## Multiple instances with Views and Proxies

The most complex types of services not only provide each caller with its own view, but inject objects between the caller and the view. These can be used to cache intermediate data, pre-process requests and provide a richer, more client friendly API (such as more methods with optional parameters set to defaults). For some services, the only feasible way to use them is with this approach (where they inject code into the calling process).

An example of this would be the configuration service, which in the kernel part only finds the correct node that holds the configuration, but then communicates this node path to the client side object, which actually opens the node and returns a configured object. This also implies that the interface between the proxy and the view does not need to be the same as that exposed by the proxy.

The view may expose a ***string GetConfigurationPath(string key)*** method while the proxy actually exposes an ***object GetConfiguredObject(string key)*** method. See .

## Services with Wrapper APIs

Figure - Applications use proxies who use views who use the main service object(s)

A variation on this design is when a service does not actually inject code, but provides a wrapper API, usually with static methods and properties, and a client side class library that uses the service views, proxies or instances behind the scene (the user does not care, since he uses the wrapper API).

An example of this could be the Graphics library, where we could have code like:

// this actually enumerates services behind the scenes and uses the first

// compatible implementation. Note that we are using a Device, not an

// IDevice, which means that it has a richer API then an IDevice.

using (Device dev = Graphics.CreateFirstMatchingDevice(

Supports.PixelShading,

Supports.Instancing,

Supports.RenderingPixelFormat.R8G8B8A8,

Usage.Shared))

{

// not bound to any device - all processing and memory is local

Texture texture = new Texture(512, 512, PixelFormat.R8G8B8A8);

// bound to the device we are using - uses service calls,

// caches some info locally. Might not actually create on the kernel

// side until needed – this is called lazy initialization

Texture texture2 = new Texture(dev, 512, 512, PixelFormat.R8G8B8A8);

// now bound to the device, will now be doing service calls

texture.BindToDevice(dev);

// not bound to the device anymore - all processing and memory is local

texture2.UnbindFromDevice();

}

This example demonstrates using solid classes (like Texture2D and Device) instead of interface instances. It also demonstrates how using solid classes can make usage of service architecture a whole lot less intrusive.

# Choosing the best Strategy

When architecting a service, the designer must be fully aware of the implications of each of the previously enumerated design ideologies.

Sharing a single instance implies that the service will act only on parameters provided by the API user. If the parameters provided by the user are insufficient, because we need a hard guarantee about the users’ identity (to prevent security risks as well as confusing our end users who have provided us with illegal parameters), we need views.

Views are be objects that cache the callers’ identity and might either actually implement the request handling or defer it to a central service object.

When even more flexibility is desired, or we wish to execute calls that are not preferred to be done in the kernel environment (such as deserializing objects from the database) we will use client side proxies. If these appear to be more trouble than they are worth, we may promote them to a client side API that will use our service views (or instances, or other proxies) behind the scenes.

# Preparing for an Implementation

When designing a service, it is important to note that interfaces that the clients use will probably reside in a different library (assembly) than the actual service code. This is both to enhance modularity and to hide code from the user. It has added benefits of being future proof, as we can add our proxies or client side APIs to this same, client-side assembly as we need them.

The service must implement a dispatcher class, which is an implementation of the ***IServiceBackend*** interface. This object controls the creation of service instances that get used by the client process. It is important to note that you shouldn’t confuse the Dispatcher with the actual service object. Under no circumstances (unless you know exactly what you are doing) should you have the dispatcher return ***this***as a result of a ***Get***.

The dispatcher will get the calling process delivered for inspection, so that it can make possible decisions based on the caller’s identity. This is the only place where you can check the caller when using single instance services.

While implementing single instance services, use the singleton pattern as a global static object that the dispatcher returns to all callers. When implementing views you have two options: either implement all functionality inside the view objects or delegate execution to the main service singleton.

The first option makes the central object obsolete, and it may actually not be needed. If you need service-level synchronization, or keep track of resources shared by multiple views, the second approach is more feasible – the views check incoming parameters for consistency, query about security capabilities and then delegate the work down to the main service.

Sometimes a combination of both practices might be needed, with the view doing some high level work and calling the service for synchronization or lower level details.

When implementing a Dispatcher for a service that injects proxies into the client side, return a ***ServiceInfo*** class instance. When this class is returned by the service, the kernel helpers present in the client process will interpret it as a prescription on how to create proxy instance. You can instruct the kernel helper to load a specified assembly, create an instance of a certain type and provide an argument to its constructor.

Usually, you don’t need to specify an additional assembly, since you will be putting proxy objects into your assembly containing service interfaces. But, in case you have many implementations, or plug-in like proxies, you may have need of this property. You should specify the proxy type – this would be a publically accessible class that takes a single ***object*** parameter for its constructor and implements the interface that the user is requesting. The argument property should be set to an object that the proxy will use to work with, usually a view.

# Using Services

To obtain a service, use the ***OS.ServiceLocator*** static property, where you will find a Find method, which takes a type and returns an ***IService*** array. With this approach, you may select your actual service host by iterating through the array and selecting the best match.

An easier solution with much less control is the ***interface*** ***OS.ActivateService<interface>*** static template method that returns the first creatable object of that interface type. Services are sorted by age, which means that the ***OS.ActivateService*** will yield an implementation by the oldest service that allows creation of your target type.

Once obtained, use the interface instance of the service like you would any object – call methods on it, put it in a list, but keep in mind that the service instance is a sometimes remoted object living in kernel space, and thus calls to it are more expensive than local calls. If possible, reuse values obtained by same parameters.

Do not serialize or share objects received through the services, unless they are marked with support for that feature – serializing or sharing such objects may compromise security or confuse the service, possibly resulting in the shutdown of the service process.

The same goes when writing proxy APIs which communicate with the service – aggregate calls, and use as little as possible of them.

## Lifetime Services

The lifetime of a service object is currently controlled by the amount of time the process that has obtained them will live. That means that objects that are returned by services to be used by a process (when the process wants, i.e. for indefinitely long) must inherit from ***IViewMarshalByRef*** and expect to be garbage collected “when the process exits”.

In the future, this scheme may change to allow for real distributed garbage collection, where service objects will be garbage collected on the server side when their proxies on the client side have been garbage collected. Until this feature is implemented, users and implementers are advised to use the ***IDisposable*** / **using** paradigms.