Gladinet SDK Documentation

Introduction

Gladinet Cloud Desktop is designed with a 2-tiered architecture consisting of the core framework and plugins for each cloud storage service provider. The framework is the heart of the system and provides a common technology platform and a uniform look and feel for all storage services. It implements all the core features like compression and chunking, task, backup and virtual directory management, while managing the interface to the local desktop and operating system. It also presents the UI and is responsible for driving the plugins to talk to their corresponding service provider. Each plugin simply has to implement a common interface to their provider. This interface includes a few basic calls like ListDirectory(), CreateDirectory(), RemoveDirectory(), Read() and Write(). Each of these calls will use the methodology provided by their provider to manipulate data in the cloud. For example, some plugins will communicate with the server using the REST API, while others may require a proprietary interface.

This 2-tiered model encapsulates the idiosyncrasies of communication with each provider within its plugin while allowing the framework to focus on the common core of functionality.

The API

The API is designed to create a consistent interface between the service provider and the Gladinet framework.

Synchronization

It is important to note that the framework does not assume that the work it requested is complete when an api call returns. Instead, it is structured to request data from the plugin and then wait for an asynchronous response. For example, a ListDirectory() operation is not completed when the api returns, it is completed when the plugin calls m_Context.EndDirList(). Similarly, after CreateDirectory() returns success, the framework will not consider the work completed until it is able to do a ListDirectory() and see the created directory in the resulting listing. A plugin developer is therefore allowed the discretion to implement the interface in a synchronous or asynchronous manner. The developer must also be aware that depending on the responsiveness of the service provider, the framework may retry requests before they are completed.

Path Translation

The framework maintains a mapping between the local virtual directories and their paths on the web. For example, mounting an Azure account allows the creation of a virtual directory which might be named Azure. This results in a folder name Azure being created under the users' Gladinet drive. However, the file /Azure/Documents/foo.txt cannot be found unless an endpoint and account are specified. Thus it would need to be translated to something like

http://accountinfo.blob.core.windows.net/Documents/foo.txt. This translation is managed in the plugin factory. To facilitate this, the plugin developer will need to implement a routine like GetCustomerDefinedPlugin() in the factory. This routine will take a Uri and translate it as outlined above. To do so, it will leverage two framework methods, GetVirtualDirectory() and GetSafeCardFieldValue(). For reference, consider the following code snippet:

```
IVirtualDirectory VDir = m_VDirMgr.GetVirtualDirectory(VDirName);
AccountName = VDir.GetSafeCardFieldValue(AccountNameId);
PrimaryKey = VDir.GetSafeCardFieldValue(PrimaryKeyId);
```

In this snippet, AccountNameID and PrimaryKeyId are GUIDs which identify the appropriate entries in customerdefined.gladinetsp where the account name and primary key are stored.

BeginRequest() and EndRequest()

Parameters: None

Returns: GladResult

Description:

Each new request creates a new instance of CustomerDefinedPlugIn. Once the instance has been created, BeginRequest() is called. When the request is completed, EndRequest() will be called.

ListDirectory(string Uri)

Parameters: Uri contains the path for the directory to be listed.

Returns: GladResult

Description:

ListDirectory() translates the Uri from the Gladinet framework into one which is meaningful to the service provider. Once the Uri has been transformed, ListDirectory() will send a request to the service provider to return a directory listing. The response will be parsed and each directory will be added by using m_Context.AddDirectoryItem(). Once all items have been added, m_Context.EndDirList() must be called to let the framework know that the plugin has completed the listing. AddDirectoryItem also allows you to pass any vendor specific data in a byte array.

CreateDirectory(string Uri)

Parameters: Uri contains the path of the directory to be created

Returns: GladResult

Description:

Translates the Uri, constructs and sends a create directory request for the service provider, parses the response and sends the result to the framework. Note that even when a user creates a directory and immediately renames it, Windows Explorer will create a directory with a name derived from "New Folder" and then rename that directory. The framework does not have any inode equivalent so the rename is implemented by calling Move() in the plugin. This routine must be implemented before the create\rename pair will work. Finally, the framework will retry the CreateDirectory() call until ListDirectory() on the parent shows the created directory. So, it makes sense to implement the list, create and move calls together.

RemoveDirectory(string Uri)

Parameters: Uri contains the path of the directory which will be deleted

Returns: GladResult

Description:

Translates the Uri, constructs and sends a remove directory request to the service provider, parses the response and sends the result to the framework.

DeleteFile(string Uri)

Parameters: Uri contains the path of the file which will be deleted

Returns: GladResult

Description:

Translates the Uri, constructs and sends a delete file request to the service provider, parses the response and sends the result to the framework.

Move(string SrcUri, string DstUri, bool Overwrite)

Parameters: SrcUri and DstUri contain the paths of the source and destination objects. Overwrite specified whether the destination object should be overwritten if it exists.

Returns: GladResult

Description:

Translates the source and destination Uris. Constructs and sends requests to create the destination object, copy the source to the destination and delete the source if the copy succeeded. If the destination already existed it will not be overwritten unless the Overwrite flag is set to true.

OpenFileForRead(string Uri, UInt32 Offset, UInt32 Length)

Parameters: Uri contains the path of the file to be read. Offset specifies the location to begin reading from and Length how much data will be read

Returns: GladResult

Description:

Prepares the plugin and framework for a read sequence and requests a read from the provider. The framework supports partial reads and writes, so this routine let's the framework know whether the plugin can handle that. This is done by returning either GladResult.PartialContent, GladResult.Success, GladResult.Fail, or GladResult.NotEnoughData. PartialContent means that the framework's request was smaller than the file size and partial content is supported. Success means that we can read but do not support partial content. NotEnoughData means that the requested data length exceeds the file size and Fail means that we are unable to open the file for reading.

Read()

Parameters: None

Returns: GladResult

Description:

The read routine passes data back to the framework using m_Context.OnReadData(). For example, assuming that the response from the provider has been stored in a stream, the routine might behave as implied by the following snippet:

```
int count = stream.Read(buffer, 0, (int) m_Length);
while (count > 0)
{
    if (!m_Context.OnReadData(buffer, (uint)count).Succeeded)
        return GladResult.Fail;
    count = stream.Read(buffer, 0, 256);
}
```

OpenFileForWrite(string Uri, bool Overwrite, UInt32 Offset, UInt32 Length)

Parameters: Uri contains the path of the file and Overwrite indicates whether the existing file should be changed. Offset indicates where the write will start and Length indicates the size of the write.

Returns: GladResult

Description:

OpenFileForWrite() assures the framework that the plugin is prepared to start receiving write requests. Because the framework will probably complete the write by sending small 8K chunks, it also let's the plugin know how much data to expect and gives it a chance to setup, if needed.

Write(byte[] Data, uint DataSize)

Parameters: Data contains the list of bytes that are to be written and DataSize specifies the size of the current write.

Returns: GladResult

Description:

Optionally buffers the data and then sends it to the provider. The framework usually sends data in 8K chunks. It is up to the plugin developer to determine the optimal transfer size. If that happens to be larger than 8K, each chunk can be buffered until the transfer size has been reached and then the contents of the buffer can be uploaded to the service provider. The framework does not care whether the data in each write request was actually written to the provider. Returning Success from the Write() call simply means that the plugin is prepared to receive the next write request. It is the plugin's responsibility to make sure that the data is successfully uploaded, and determine how the framework will be notified of the completion. Further details are provided in the description of CloseWrite() that is given below. If we are blocking the Write() until it is committed to the provider, then returning Success after the last write is committed is sufficient. In this case, CloseWrite() will simply return Success. If we write asynchronously , m_Context.OnOperationComplete() must be called once the last write has been committed, to let the framework know that the write has been completed.

CloseWrite()

Parameters: None

Returns: GladResult

Description:

Used by the framework to let the plugin know that the framework has written all of its data. The plugin is then required to either return GladResult.Success or GladResult.Pending. If the plugin returns Pending, the framework will wait for a call to m_Context.OnOperationComplete() before it considers the write to be complete. Alternatively, if the plugin is done when CloseWrite() is called , then Success can be returned. If the writes have all been done synchronously, then CloseWrite() can always return Success. If however, the Write() calls are allowed to return before the data is actually committed to the provider, then the Pending return will let the framework know that there may be outstanding writes and it should wait for the OnOperationComplete() call.

Getting Started

Gladinet has provided a plugin for the local file system as a reference implementation. The easiest way to get started is with a copy of this implementation. First copy ..\GladPluginFileSys to ..\YourPluginName

The reference implementation contains the following files: GladinetLocalFileSysPlugin.cs, GladinetLocalFileSysPluginFactory.cs, PluginModule.cs and localfilesys.gladinetsp. In turn, these contain implementations of the interfaces which will create a new plugin, namely: IStoragePluginModule, IStoragePluginRequestHandler, IStoragePluginInterfaceExt, IStoragePluginFactory

IStoragePluginModule

This is implemented in PluginModule.cs as shown below

```
namespace Gladinet.Plugin
{
        ...
        public
        IStoragePluginFactory
        GetStoragePluginFactory()
        {
            return new GladinetLocalFileSysPlugInFactory();
        }

        public
        string
        PluginFQDN {
            get
            {
                return "Gladinet.Plugins.Storage.GladinetLocalFileSysPlugIn";
            }
        }

        public
        string
        PluginDescription
        {
            get
            {
                return "Gladinet Local File System Plugin";
            }
        }


    }

}
```

Simply modify the member function and properties to return the appropriate data for your plugin. For example, GetStoragePluginFactory() may end up looking like this:

```
        public
        IStoragePluginFactory
        GetStoragePluginFactory()
        {
            return new CustomerDefinedPlugInFactory();
        }
```

This brings us to our next interface implementation.

IStoragePluginFactory

The definition of CustomerDefinedPluginFactory () lives in this interface which can be found in GladinetLocalFileSysPluginFactory.cs. Here are some excerpts from the implementation:

```
namespace Gladinet.Plugins.Storage
{
    class GladinetLocalFileSysPlugInFactory : IStoragePluginFactory
    {
        private IVirtualDirectoryMgr m_VDirMgr;

        public
        object
        CreateNewRequestHandler(IStorageSessionCtx Ctx)
        {
            return new GladinetLocalFileSysPlugIn(Ctx, this);
        }

       ...

        public
        string
        GetPhysicalPath(
            string uri
            )
        {
         ...

            VDir = m_VDirMgr.GetVirtualDirectory(VDirName);
            Root = VDir.GetSafeCardFieldValue("{4A3C8DF8-D2E5-4b99-AF00-63D228FFF86A}");
       ...

            return PhysicalPath;
        }

        public
        string
        ProviderId
        {
            get
            {
                return "{A99CCB1F-6FBD-4fd8-AF9C-A897A24F37EF}";
            }
        }

       ...

    }
}
```

Three changes are needed here:

1. Change GladinetLocalFileSysPlugInFactory to CustomerDefinedPlugInFactory.

2. Modify the GUIDs which identify the path field and the provider id.
3. Modify CreateNewRequestHandler() as follows:

```
public
object
CreateNewRequestHandler(IStorageSessionCtx Ctx)
{
    return new CustomerDefinedPlugIn(Ctx, this);
}
```

The GUIDS are found in the localfilesys.gladinetsp file which will be renamed to store some properties for the plugin under construction. A tool like uuidgen will be needed to create new GUIDs for the plugin under development. For reference, note the location of the GUIDs above in localfilesys.gladinetsp. Note that the GUID returned by the ProviderId property is actually called NetPluginId in the.gladinetsp file and that the SPID should be the same for all plugins.

IStoragePluginRequestHandler and IStoragePluginInterfaceExt

Now that the plugin framework has been defined, we are ready to implement the core interfaces which will define how the Gladinet framework communicates with the service provider. This is simply a matter of implementing the API described in the section above.

```
namespace Gladinet.Plugins.Storage
{
    class GladinetLocalFileSysPlugIn : IStoragePluginRequestHandler,IStoragePluginInterfaceExt
    {
        ...

        public
        GladinetLocalFileSysPlugIn(
            IStorageSessionCtx Context,
            GladinetLocalFileSysPlugInFactory Factory
            )
        {
            m_Context = Context;
            m_Factory = Factory;
        }

        public
        GladResult
        BeginRequest(

            )
        {
        }

        public
        GladResult
        EndRequest(

            )
        {
        ...
        }

        public
        GladResult
        ListDirectory(
            string Uri
            )
        {
```

```
...
}

public
GladResult
CreateDirectory(
    string Uri
    )
{
...
}

public
GladResult
RemoveDirectory(
    string Uri
    )
{
...
}

public
GladResult
DeleteFile(
    string Uri
    )
{
...
}
public
GladResult
Move(
    string SrcUri,
    string DstUri,
    bool   Overwrite
    )
{
...        }

public
GladResult
OpenFileForWrite(
    string Uri,
    bool Overwrite,
    UInt32 Offset,
    UInt32 Length
    )
{
...
}

public
GladResult
Write(
    byte[] Data,
    uint DataSize
    )
{
...
}

public
GladResult
CloseWrite()
{
...
}
```

```
        public
        OpenReadResult
        OpenFileForRead(
            string Uri,
            UInt32 Offset,
            UInt32 Length
            )
        {
        ...
        }

        public
        GladResult
        Read()
        {
        ...
        }

        ...
    }
}
```

Here GladinetLocalFileSysPlugin needs to be changed to CustomerDefinedPlugin and each of the calls will need to be implemented. Note that m_Context allows data to be passed back to the framework. For example, ListDirectory will call m_Context.AddDirectoryItem() to add each directory that the provider returned.

IStorageSessionContext and m_Context

Throughout this document we have been leveraging this interface through the m_Context member of our IStoragePluginInterfaceExt imlpementation. The interface definition looks like this:

```
    public interface IStorageSessionCtx
    {
        /*
         * Request
         */
        string
         GetRequestHdr(string name);

        /*
         * Response
         */
        bool
        AddDirectoryItem(
            GladNodeInfo FInfo,
            ulong Size,
            byte[] VendorData
            );

        bool
        EndDirList();


        GladResult
        OnReadData(
            byte[] Data,
            UInt32 DataSize
            );

        void
        OnOperationComplete(
            GladResult gResult,
```

```
            byte[]      OperationResult
            );
        ...
    }
```

This interface allows the plugin to communicate with the framework as follows:

AddDirectoryItem() and EndDirList()allow ListDirectory to define a directory listing. Each file or directory is returned by calling AddDirectoryItem() and the framework is notified that the listing has been completed by calling EndDirList().

OnReadData() is used by Read() to transfer data returned from the service provider to the framework.

OnOperationComplete() is used by Write() to notify the framework that the write has completed. This is needed when CloseWrite() returns a pending status.

Finally, vendor data can be set for each directory item. It is an array of bytes which can contain metadata for the file or directory. Alternatively, it can contain any vendor or file specific information selected by the plugin developer.

Gladinet SP File Creation and Mounting

In order to register a plugin with Gladinet Cloud Desktop, an XML file must be created with the extension .gladinetsp. For example, localfilesys.gladinetsp looks like this:

```
<VirtualStorage>
        <DisplayName>.Net File System</DisplayName>
        <SPID>{E910767C-28D6-46ad-890B-FCE357D9952D}</SPID>
        <NetPluginId>{A99CCB1F-6FBD-4fd8-AF9C-A897A24F37EF}</NetPluginId>
        <PluginModuleLocation>
                file://D:/gladinet/src/GladinetPluginHost/GladPluginFileSys/bin/Debug/GladPluginFileSys.dll
        </PluginModuleLocation>
        <LocalHost>allowed</LocalHost>
        <StorageType>Generic</StorageType>
        <Icon>http://www.gladinet.com/r/cashbundle.gif</Icon>
        <PublishedBy>Gladinet Inc.</PublishedBy>
        <PublishedDate>06/23/2009</PublishedDate>
        <Description>A sample implementation of a storage plugin using the Gladinet Storage SDK</Description>
        <PublisherWeb>http://www.gladinet.com</PublisherWeb>
        <SignupURL>http://www.gladinet.com</SignupURL>
        <ConfigurationURL>http://www.gladinet.com</ConfigurationURL>
        <Capability>
                <AllowRootFile>True</AllowRootFile>
                <MaxAllowedFileSize>0</MaxAllowedFileSize>
                <AllowedFileExtension></AllowedFileExtension>
                <DisallowedFileExtension></DisallowedFileExtension>
        </Capability>
        <SafeCardTemplate>
                <Field>
                        <Name>Root Path</Name>
                        <Encrypted>False</Encrypted>
                        <FieldId>{4A3C8DF8-D2E5-4b99-AF00-63D228FFF86A}</FieldId>
                </Field>
        </SafeCardTemplate>
</VirtualStorage>
```

Once this file is created, a virtual directory for the plugin can be mounted by invoking the FTA for the file type. When Gladinet Cloud Desktop is installed, this association is created with the Gladinet Virtual

Directory Manager executable (GVDirMgr.exe). Following are a few notes on the elements in the XML file which are not self explananatory:

- SPID contains a GUID which should be the same for all plugins. Copy this GUID to your file.
- NetPluginID is unique for each plugin. Use uuidgen to create a new one.
- PluginModuleLocation is the installed location of the plugin
- Capability contains several sub elements: AllowRootFile indicates whether the provider allows files to be added at the root level. Writes will fail and return an appropriate error if MaxAllowedFileSize is exceeded. If set to 0, there is no maximum. Etc…
- SafeCardTemplate describes the fields needed to login. Examples of fields include usernames, access tokens, etc… Each field is identified by a GUID which is used by the plugin factory as described above.

The Gladinet Directory

Gladinet has created a directory to serve as a central storage location for all plugins. Plugins from this directory will be made readily available to Gladinet Cloud Desktop users. The directory can be found at http://www.gladinet.com/p/umdirectory.aspx