360 | Development Platform

understanding services

360 | Understanding Services Programming Guide and Examples

Copyright

Disclaimer

Notice

Nuance Communications, Inc. is strongly committed to creating high quality voice and data management products that, when used in conjunction with your own company's security policies and practices, deliver an efficient and secure means of managing confidential information.

Nuance believes that data security is best maintained by limiting access to various types of information to authorized users only. Although no software product can completely guarantee against security failure, 360 | Understanding Servicessoftware contains configurable password features that, when used properly, provide a high degree of protection.

We strongly urge current owners of Nuance products that include optional system password features to verify that these features are enabled! You can call our support line if you need assistance in setting up passwords correctly or in verifying your existing security settings.

Published by Nuance Communications, Inc., Burlington, Massachusetts, USA

Visit Nuance Communications, Inc. on the Web at www.nuance.com.

Visit Nuance Healthcare on the Web at www.nuance.com/healthcare.

Contents

What is the 360 | Understanding Services SDK?

Nuance supports various clinical language understanding (CLU) features, including the ability to dictate and transcribe patient reports, process those reports, and extract pertinent data from them.

The 360 | Understanding Services SDK is a set of software components that allows you to integrate a wide range of client applications in the EHR domain, including transcription, dictation and document signing, with a variety of Nuance Healthcare services, supporting workflows in a Meaningful Use context.

The SDK offers simple, powerful programming interfaces to engage state of the art technology in medical document fact extraction. It is designed with extensibility in mind, accommodating the expansion of service offerings by Nuance Healthcare while providing familiar concepts and integration interfaces.

As Nuance Healthcare research and development progresses, client systems that consume services via the SDK can take advantage of improved service implementations without requiring major system updates.

What Does the SDK Include?

. The SDK consists of two parts:

- An API that allows you to connect to Nuance services and control the flow of documents through the document processing workflows. See *360 | Understanding Services Workflows* on page 47 for more information about the document processing workflows, and *The 360 | Understanding Services API* on page 9 for more information about the API.
- A set of XML templates that define the structure for various types of electronic medical records (EMRs). The structure of the templates is based on the Health Level 7 International (HL7) Clinical Document Architecture (CDA) v2 standard. See *About the XML Output* on page 67 for more information about the CDA XML output.

The software and documentation components included in the SDK are:

- Binaries:
  - `Nuance.NUS.Sdk.dll` – Core implementations of NUS services.
  - `Nuance.NUS.Sdk.Clu.dll` – CLU service implementation.
- Configuration and Licensing:

    - `CluSdk.config` – Contains the initial connection configuration for the partner organization.
    - `License.lic` – Contains licensing information for the partner organization.
- Documentation:
    - 360 | Understanding Services Programming Guide and Examples – This book; an overview of the SDK and a guide to common programming tasks.
    - `Nuance.NUS.Sdk.xml` – IntelliSense support for Visual Studio.
    - `Nuance.Nus.Sdk.Clu.xml` – IntelliSense support for Visual Studio.
    - One or more sample projects demonstrating CLU SDK use scenarios.

Environment Requirements

The SDK supports integration/programming environments based on .NET framework version 3.5 and higher, running on Microsoft Windows XP SP3 and later.

Security

Secure Session and Authentication

By default, connecting to Nuance Healthcare services requires SSL (https://...) security protocol, and all communication between the client application and Nuance services via SDK is conducted using SSL encryption.

Upon successful authentication, the server establishes a secure session and provides the security tokens necessary to verify every call to the service and maintain session lifecycle.

Session's duration is controlled either by explicit closing and disposal, or by exceeding maximum idle time. You can configure the maximum idle time via a server side setting. When the session is closed, no further API invocations are accepted.

The CLU SDK provides capabilities to validate the organization license for CLU SDK. To simplify the integration the CLU SDK creates users automatically if they do not already exist in the system. The APIs require minimum user information to complete this operation.

Authorization and License Verification

Each integration partner will receive separate licensing information, containing a unique license identifier, partner identifier and name.

The 360 | Understanding Services API

The 360 | Understanding Services API consists of several interfaces:

- Session, which allows you to create and control sessions.
- Configuration, which allows you to configure connection and logging settings.
- The CLU Service Interface, which supports the document processing workflows described in *360 | Understanding Services Workflows* on page 47.

The following sections describe these interfaces in greater detail.

Connecting to Nuance Healthcare Services

360 | Understanding Services supports the following mechanisms of establishing a connection to the Nuance Healthcare services:

- Explicit Host – A connection method to a static, well-known entry point URL.

  With this method, service location resolution is defined up front with each Nuance partner, and the partner receives the service entry point URL as part of SDK configuration.

- InfoServer – A connection method where standard eScription InfoServers dynamically determine the service entry point for each customer/partner account. This connection method allows integration partners to access high-availability, high-performance servers maintained by Nuance.

  With the InfoServer connection method, partner specific SDK configuration information contains references to pre-defined realm URLs as arguments to an InfoServer resolution of service entry point location.

360 | Understanding Services also supports a ServiceEmulator connection option, where service behavior is emulated within SDK implementation, but no actual connection occurs. This feature is particularly useful in the early stages of integration development, such as evaluation and preliminary testing of the SDK.

Accessing the 360 | Understanding Services Application

Complete the following steps to access the 360 | Understanding Services application:

Step One: Open Your Welcome Kit

After you purchase the 360 | Understanding Services SDK, Nuance will send you a welcome kit containing the following information:

- A partner GUID
- A license GUID
- Your institution name
- Your institution's URL
- The expiration date of your access to the application.

Make a note of the GUIDs, the institution name, and your institution's URL; you will need them to access the application.

You can access the 360 | Understanding Services application approximately 24 hours after you receive the welcome kit.

Step Two: Use the API to Create a Connection to the Application

Complete the following steps to use the API to connect to the 360 | Understanding Services application:

1. Create a Configuration object, supplying the URL from your welcome kit:

   See *Configuration* on page 20 for more information about the Configuration object.

2. Create a License object, supplying the Partner GUID, License GUID, and Institution Name from your welcome kit:

   See *Sessions* on page 12 for an example that includes creating a License object.

3. Create a SessionFactory object:

   See *Sessions* on page 12 for an example that includes creating a SessionFactory object, and *SessionFactory()* on page 15

4. Create an instance of the CLU service:

   See *Sessions* on page 12 and *Session Class* on page 14

Note: 360 | Understanding Services application uses a licensing model, and auto-provisions user IDs like the one used in the preceding example. There is a limitation of 8 distinct logins.

Sessions

A session is an essential SDK concept. Its responsibilities are:

- To maintain a secure session connection.
- To support authentication and authorization in SDK interactions.
- To provide access to Nuance service(s).
- To create and maintain a secure operational context for each API call within the accessed service.

SDK client code must maintain the scope of a session for any services accessed. Without a session context, you cannot use the service and its APIs.

To gain access to CLU services, you get a handle to a service object:

1. Create `SessionFactory()`, with the `Configuration` object as an argument.
2. Create a session using the `factory.Open()` method, providing user credentials and license information.
3. Create a CLU service instance via `Session`’s `GetService()` method.

The following sample code illustrates this procedure:

```
[Test]
public void SpecSessionAndServiceCreation()
{
        // configure connection; an emulated connection in this case
        var config = new Configuration
                {
                        SelectedConnection = new ConnectionConfig
                                {
                                        Type = ConnectionType.SessionEmulator,
                                        Name = "Emulated connection"
                                }
                };

        // populate license information
        var license = new License
                {
                        PartnerId = "50B1A16A-DEF7-48BE-9FA2-A49AC68BB72F",
                        Id = new Guid( "7B0E51D7-30F1-4315-A79D-3E57A60DA671" ),
                        CustomerName = "CustomerName",
                };

        // create session factory
        var factory = new SessionFactory( config );

        // create session
        using( var session = factory.Open( "userId", license ))
        {
                // create service
                using( var service = session.GetService<ICluService>())
                {
                        // utilize service
                        Assert.DoesNotThrow( () => service.IsAvailable() );

                } // disposal of service
        } // disposal of session
}
```

Note: 360 | Understanding Services application uses a licensing model, and auto-provisions user IDs like the one used in the preceding example. There is a limitation of 8 distinct logins.

The preceding example demonstrates the relationship between the session and service, as a service can be utilized only within a scope of a secure session.

Depending on specific use scenarios, session/service access can be maintained in number of variations to fit the needs of the integrating application. For example: a simple Windows form can create session and service instances as member variables during instantiation and/or loading, and upon closing, properly dispose service and session member variables.

Note: If any subsequent API throws an `AuthenticationException`, use the Session API to re-establish a session.

Session Class

The following table describes the `Session`class's properties and methods. See *Sessions* on page 12

Property/Method	Description
`SessionFactory()`	Establishes an authorized session with the CLU middle tier.
`INusSession Open(string userId, ILicense license)`	Opens a session with user ID and license.
`INusSession Open(string userId, string password, ILicense license)`	Opens a session with user ID, password, and license.
`GetService<ICluService>()`	Creates an instance of the NUS service.
`IsAvailable<ICluService>()`	Returns True or False, indicating whether the `ICluService` is available.

SessionFactory()

Description

Establishes an authorized session with the CLU middle tier.

Syntax

```
public SessionFactory( Configuration config )
```

Parameters

Parameter	Required?	Description
`config`	Y	A configuration object. See *Configuration* on page 20 for more information.

Returns

An instance of `SessionFactory`.

Example

```
var factory = new SessionFactory( config );

       using( var session = factory.Open( "userId", license )) // create session
       {
               // create service
               using( var service = session.GetService<ICluService>() )
               {
                       using( var docStream = GetDocStream() ) // open doc as stream
                       {
                               // execute extraction
                               var response = service.ExtractDocument( docStream,
                                       _docMetaData, _patient, _visit );

                               // verify results
                               Assert.IsTrue( response.DocumentId != -1,
                                       "Document id should be non-negative." );
                               Assert.IsTrue( response.ExtractedDocument.Length > 0,
                                       "Extracted document length should be
                                        greater than 0" );
                       }
               }
       }
```

INusSession Open(string userID, ILicense license)

Description

Opens a session with user ID and license.

Note: 360 | Understanding Services application uses a licensing model to authenticate users, and auto-provisions user IDs like the one used in the following example. There is a limitation of 8 distinct user IDs for a single license.

Syntax

```
INusSession Open(string userId, ILicense license);
```

Parameters

Parameter	Required?	Description
`userID`	Y	Login name identifying the user/application on the customer system.
`license`	Y	GUID used to identify the customer's licensing entity.

Returns

Returns a session instance.

Example

```
var factory = new SessionFactory( config );

       using( var session = factory.Open( "userId", license )) // create session
       {
               // create service
               using( var service = session.GetService<ICluService>() )
               {
                       using( var docStream = GetDocStream() ) // open doc as stream
                       {
                               // execute extraction
                               var response = service.ExtractDocument( docStream,
                                       _docMetaData, _patient, _visit );

                               // verify results
                               Assert.IsTrue( response.DocumentId != -1,
                                       "Document id should be non-negative." );
                               Assert.IsTrue( response.ExtractedDocument.Length > 0,
                                       "Extracted document length should be
                                        greater than 0" );
                       }
               }
       }
```

GetService<TService>()

Description

Creates an instance of an NUS service.

Syntax

```
TService GetService<TService>() where TService : INusService;
```

Parameters

Parameter	Description
`TService`	The service type to create.

Example

```
// create session factory
var factory = new SessionFactory( config );

       // create session
       using( var session = factory.Open( "userId", license ))
       {
               // create service
               using( var service = session.GetService<ICluService>())
                       {
                               // utilize service
                               Assert.DoesNotThrow( () => service.IsAvailable() );

                       } // disposal of service
       } // disposal of session
```

IsAvailable<TService>()

Description

Returns True or False, indicating whether an instance of the specified service type is available.

Syntax

```
bool IsAvailable<TService>() where TService : INusService;
```

Parameters

Parameter	Description
`TService`	The service type to check for.

Returns

`True` or `False`, indicating whether an instance of the specified service type is available.

Example

```
// create session factory
var factory = new SessionFactory( config );

        // create session
        using( var session = factory.Open( "userId", license ))
        {
                // create service
                using( var service = session.GetService<ICluService>())
                {
                        // utilize service
                        Assert.DoesNotThrow( () => service.IsAvailable() );

                } // disposal of service
        } // disposal of session
```

Configuration

System wide settings are contained within `Configuration` type. The Configuration type has two purposes:

- It specifies the connection details for the SDK, as described in *Connecting to Nuance Healthcare Services* on page 10.
- It allows the client application to configure various settings, including:
  - Maximum number of connection retries.
  - Maximum number of call retries.
  - Maximum connection idle time.
  - Log file location.
  - Log file size.
  - Logging thresholds.

The `Configuration` type allows multiple connection definitions. This allows you to specify connection details without having to recompile.

Figure 2 shows a class diagram of the `Configuration` type:

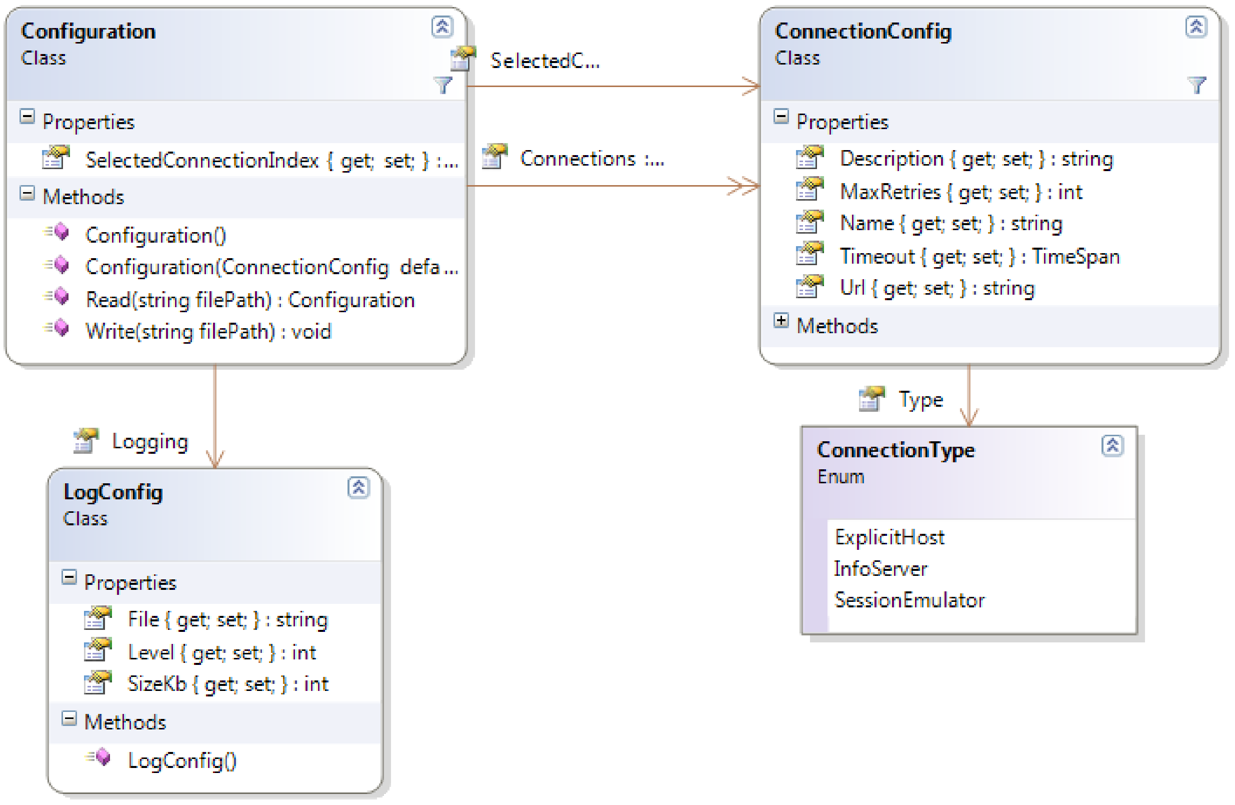
Configuration
Class
Properties
SelectedConnectionIndex { get; set; } :...
Methods
Configuration()
Configuration(ConnectionConfig defa...
Read(string filePath) : Configuration
Write(string filePath) : void
SelectedC...
Connections :...
ConnectionConfig
Class
Properties
Description { get; set; } : string
MaxRetries { get; set; } : int
Name { get; set; } : string
Timeout { get; set; } : TimeSpan
Url { get; set; } : string
Methods
Logging
LogConfig
Class
Properties
File { get; set; } : string
Level { get; set; } : int
SizeKb { get; set; } : int
Methods
LogConfig()
Type
ConnectionType
Enum
ExplicitHost
InfoServer
SessionEmulator

Figure 2: The `Configuration` Type

Each SDK ships with default configuration settings: a simple XML serialization of `Configuration`. Integrators can use the default XML serialization, or set the values programmatically, as in the following example:

```
using System;
using System.Linq;
using NUnit.Framework;

namespace Nuance.NUS.Sdk.Clu.Examples
{
    [TestFixture]
    public class ConfigurationExample
    {
        [Test]
        public void ConfigurationSpec()
        {
            var config = new Configuration();

            // Add connection configurations for specific purpose
            config.Connections.AddRange( new[]
                {
                    new ConnectionConfig

                        {
                            Type = ConnectionType.ExplicitHost,
                            Name = "ExplicitHostExample",
                            Description = "Connection to an explicit host",
                            Url = "https://[host]/[institution]",
                            MaxRetries = 5,
                            Timeout = TimeSpan.FromSeconds( 30 )
                        },
                    new ConnectionConfig
                        {
                            Type = ConnectionType.InfoServer,
                            Name = "Test",
                            Description = "Connection to Test InfoServer",

                            Url = "https://[host]/[institution]",
                        },
                    new ConnectionConfig
                        {
                            Type = ConnectionType.SessionEmulator,
                            Name = "CLU session emulator",
                        },
                } );

            // one of the configured connections should be selected
            // for use in SessionFactory session creation
            config.SelectedConnection = config.Connections.First(); // or
            config.SelectedConnection = config.Connections[0];

            // or it could be set explicitly:
            config.SelectedConnection = new ConnectionConfig
                {
                    Type = ConnectionType.ExplicitHost,
                    Url = "https://[host]/[institution]",
                };

            // or set any other property ...
            config.Logging.File = @"c:\Temp\MyLogFile.log";
        }
    }
}
```

Configuration Class

The following table describes the `Configuration` class's properties and methods. See *Configuration* on page 20

Property/Method	Description
`SelectedConnectionIndex()`	Gets/sets the selected connection index.
`LogConfig Logging { get; set; }`	Gets/sets a logging configuration object.
`List<ConnectionConfig> Connections { get; set; }`	Gets/sets a list of configured connections.
`ConnectionConfig SelectedConnection { get; set; }`	Gets/sets the selected connection.
`void Write(string filePath);`	Serializes a Configuration object to the specified file.
`Static Configuration Read( string filePath );`	De-serializes a Configuration object.

List<ConnectionConfig> Connections { get; set; }

Description

Sets or gets a list of configured connections. See *Connecting to Nuance Healthcare Services* on page 10 for additional information about connections.

Syntax

```
List<ConnectionConfig> Connections { get; set; }
```

Example

```
var config = new Configuration();

       // Add connection configurations for specific purpose
       config.Connections.AddRange( new[]
               {
                       new ConnectionConfig
                               {
                                       Type = ConnectionType.ExplicitHost,
                                       Name = "ExplicitHostExample",
                                       Description = "Connection to an explicit host",
                                       Url = "https://[host]/[institution]",
                                       MaxRetries = 5,
                                       Timeout = TimeSpan.FromSeconds( 30 )
                               },
                       new ConnectionConfig
                               {
                                       Type = ConnectionType.SessionEmulator,
                                       Name = "CLU session emulator",
                               } );

               }
```

ConnectionConfig SelectedConnection { get; set; }

Description

Sets or gets information about the selected connection. See *Connecting to Nuance Healthcare Services* on page 10 for additional information.

Syntax

```
ConnectionConfig SelectedConnection { get; set; }
```

int SelectedConnectionIndex { get; set; }

Description

Sets or gets the selected connection index. See *Connecting to Nuance Healthcare Services* on page 10 for additional information about connections.

Syntax

```
int SelectedConnectionIndex { get; set; }
```

Parameter

If setting the property, the index for the connection.

LogConfig Logging { get; set; }

Description

Sets or gets the logging configuration object.

Syntax

```
LogConfig Logging { get; set; }
```

Parameter

If setting the property, a `LogConfig` object. See *LogConfig Class* on page 37 for more information.

void Write(string filePath);

Description

Serialization of the connection object to file.

Syntax

```
void Write( string filePath );
```

Parameters

Value	Description
`filePath`	Path to the file that you are serializing to.

Static Configuration Read(string filePath);

Description

De-serialization of the connection object.

Syntax

```
Static Configuration Read( string filePath );
```

Parameters

Value	Description
`filePath`	Path to the serialized object.

ConnectionConfig Class

The following table describes the `ConnectionConfig` class's properties and methods. See *Configuration* on page 20

Property/Method	Description
`ConnectionType Type { get; set; }`	Gets/sets the connection type.
`string Url { get; set; }`	Gets/sets the connection entry point URL.
`TimeSpan Timeout { get; set; }`	Gets/sets the API call timeout.
`int MaxRetries { get; set; }`	Gets/sets maximum number of API call tries.
`string Name { get; set; }`	Gets/sets the connection name.
`string Description { get; set; }`	Gets/sets the connection description.

ConnectionType Type { get; set; }

Description

Sets or gets the connection type. See *Connecting to Nuance Healthcare Services* on page 10 for additional information.

Syntax

```
ConnectionType Type { get; set; }
```

Parameters

If you are setting the connection type, a value from the `ConnectionType` enumeration:

Value	Description
`ExplicitHost`	A connection method to a static, well-known entry point URL. With this method, service location resolution is defined up front with each Nuance partner, and the partner receives the service entry point URL as part of SDK configuration.
`InfoServer`	A brokered connection via an eScription InfoServer, which dynamically determines the service entry point for each customer/partner account. This connection method allows integration partners to access high-availability, high-performance servers maintained by Nuance.
`SessionEmulator`	A connection to emulated services without a connection to an external URL.

Example

```
var config = new Configuration();

       // Add connection configurations for specific purpose
       config.Connections.AddRange( new[]
               {
                       new ConnectionConfig
                               {
                                       Type = ConnectionType.ExplicitHost,
                                       Name = "ExplicitHostExample",
                                       Description = "Connection to an explicit host",
                                       Url = "https://[host]/[institution]",
                                       MaxRetries = 5,
                                       Timeout = TimeSpan.FromSeconds( 30 )
                               },
                       new ConnectionConfig
                               {
                                       Type = ConnectionType.SessionEmulator,
                                       Name = "CLU session emulator",
                                    } );

                }
```

string Name { get; set; }

Description

Sets or gets the connection name. See *Connecting to Nuance Healthcare Services* on page 10 for additional information.

Syntax

```
string Name { get; set; }
```

Parameter

If setting the property, the connection name.

Example

```
var config = new Configuration();

       // Add connection configurations for specific purpose
       config.Connections.AddRange( new[]
               {
                       new ConnectionConfig
                               {
                                       Type = ConnectionType.ExplicitHost,
                                       Name = "ExplicitHostExample",
                                       Description = "Connection to an explicit host",
                                       Url = "https://[host]/[institution]",
                                       MaxRetries = 5,
                                       Timeout = TimeSpan.FromSeconds( 30 )
                               },
                       new ConnectionConfig
                               {
                                       Type = ConnectionType.SessionEmulator,
                                       Name = "CLU session emulator",
                               } );

               }
```

string Description { get; set; }

Description

Sets or gets the connection description. See *Connecting to Nuance Healthcare Services* on page 10 for additional information about connections.

Syntax

```
string Description { get; set; }
```

Parameter

If setting the property, a string describing the connection.

Example

```
var config = new Configuration();

       // Add connection configurations for specific purpose
       config.Connections.AddRange( new[]
               {
                       new ConnectionConfig
                               {
                                       Type = ConnectionType.ExplicitHost,
                                       Name = "ExplicitHostExample",
                                       Description = "Connection to an explicit host",
                                       Url = "https://[host]/[institution]",
                                       MaxRetries = 5,
                                       Timeout = TimeSpan.FromSeconds( 30 )
                               },
                       new ConnectionConfig
                               {
                                       Type = ConnectionType.SessionEmulator,
                                       Name = "CLU session emulator",
                               } );

               }
```

string Url { get; set; }

Description

Sets or gets the connection entry point URL. See *Connecting to Nuance Healthcare Services* on page 10 for additional information.

Syntax

```
string Url { get; set; }
```

Parameter

If setting the property, the connection entry point URL.

Example

```
var config = new Configuration();

       // Add connection configurations for specific purpose
       config.Connections.AddRange( new[]
               {
                       new ConnectionConfig
                               {
                                       Type = ConnectionType.ExplicitHost,
                                       Name = "ExplicitHostExample",
                                       Description = "Connection to an explicit host",
                                       Url = "https://[host]/[institution]",
                                       MaxRetries = 5,
                                       Timeout = TimeSpan.FromSeconds( 30 )
                               },
                       new ConnectionConfig
                               {
                                       Type = ConnectionType.SessionEmulator,
                                       Name = "CLU session emulator",
                               } );

               }
```

int MaxRetries { get; set; }

Description

Sets or gets the maximum number of API call tries. See *Connecting to Nuance Healthcare Services* on page 10 for additional information.

Syntax

```
int MaxRetries { get; set; }
```

Parameter

If setting the property, the number of API call tries.

Example

```
var config = new Configuration();

       // Add connection configurations for specific purpose
       config.Connections.AddRange( new[]
               {
                       new ConnectionConfig
                               {
                                       Type = ConnectionType.ExplicitHost,
                                       Name = "ExplicitHostExample",
                                       Description = "Connection to an explicit host",
                                       Url = "https://[host]/[institution]",
                                       MaxRetries = 5,
                                       Timeout = TimeSpan.FromSeconds( 30 )
                               },
                       new ConnectionConfig
                               {
                                       Type = ConnectionType.SessionEmulator,
                                       Name = "CLU session emulator",
                               } );

               }
```

TimeSpan Timeout { get; set; }

Description

Sets or gets the API call timeout. See *Connecting to Nuance Healthcare Services* on page 10 for additional information.

Syntax

`TimeSpan Timeout { get; set; }`

Parameters

If setting the property, a `TimeSpan` object specifying the time out in seconds.

Example

```
var config = new Configuration();

       // Add connection configurations for specific purpose
       config.Connections.AddRange( new[]
               {
                       new ConnectionConfig
                               {
                                       Type = ConnectionType.ExplicitHost,
                                       Name = "ExplicitHostExample",
                                       Description = "Connection to an explicit host",
                                       Url = "https://[host]/[institution]",
                                       MaxRetries = 5,
                                       Timeout = TimeSpan.FromSeconds( 30 )
                               },
                       new ConnectionConfig
                               {
                                       Type = ConnectionType.SessionEmulator,
                                       Name = "CLU session emulator",
                               } );

               }
```

LogConfig Class

The following table describes the `LogConfig` class's properties and methods. See *Configuration* on page 20

Property/Method	Description
`LogLevel Level { get; set; }`	Gets/sets the type of information that gets logged.
`string File { get; set; }`	Gets/sets the path to the log file.
`int SizeKb { get; set; }`	Gets/sets the log size.

LogLevel Level { get; set; }

Description

Sets or gets the type of information written to the log. See *Connecting to Nuance Healthcare Services* on page 10 for additional information about connections and logging.

Syntax

```
LogLevel Level { get; set; }
```

Parameters

If you are setting the logging level, a value from the `LogLevel` enumeration:

Value	Description
`All`	All log statements.
`Debug`	Debug log statements.
`Info`	Info log statements.
`Warn`	Warning log statements.
`Error`	Error log statements.
`Fatal`	Fatal log statements.
`Off`	Disables logging.

string File { get; set; }

Description

Sets or gets the path to the log file,

Syntax

```
string File{ get; set; }
```

Parameters

Path to the log file.

Example

```
                 // or set any other property ...
                config.Logging.File = @"c:\Temp\MyLogFile.log";
```

int SizeKb { get; set; }

Description

Sets or gets the log size. See *Connecting to Nuance Healthcare Services* on page 10 for additional information.

Syntax

```
int SizeKb { get; set; }
```

Parameters

If setting the log size, the desired log segment size, in Kbytes.

Exception Handling Overview

The 360 | Understanding Services CLU SDK exception type hierarchy is designed to:

- Distinguish communication and protocol exceptions from system / application exceptions.
- Classify the origin of exceptions, for example, client and server side exceptions.

All CLU SDK exception types are derived from .NET framework's `Application Exception` class, where all CLU SDK specific data is contained within an existing interface. The only extension to the interface is the `IsRetryable()` method, applicable to Communications exceptions group. See Microsoft's .NET documentation for more information about the .NET framework and exception handling.

All exceptions are recorded in the CLU SDK log (call stack with complete exception data and a history of method call retries), unless the logging level has been set to `LogLevel.Off` (`LogConfig` class, `LogLevel` property).

Exception Handling Class Hierarchy

The SDK uses the following hierarchy for exception handling:

The exception hierarchy contains several types that are grouped as follows:

- Communication Exceptions group:
  - `NusCommunicationException`
  - `NusTimeoutException`
- System/Application Exceptions group:
  - `NusException`
  - `NusSystemException`
  - `NusAuthorizationException`

Each `NusSystemException` and its descendants contain `Response` property of type `SystemResponse`, which encapsulates the specific failure code (`RequestStatus` enumeration) and a text description. The `Response` property will also carry the name of the API method (`MethodName`) in which the exception occurred.

How the SDK Handles Exceptions

The SDK's internal exception handling consists of several steps:

- Based on the source exception type and other contextual information (workflow, session state, API call specific information), the SDK evaluates the exception state as retryable or non-retryable and classifies the exception type as either communication or timeout.
  - If the exception is non-retryable, the SDK throws appropriate communication exception type, where the `InnerException` property contains the source exception, and its `IsRetryable()` method returns False.
  - If the exception is retryable, the SDK attempts remedial action and retries the call in question (the number of retries is defined in the `ConnectionConfig.MaxRetries` property). If the retry is successful, the SDK will not throw an exception.
  - If retry is unsuccessful, the SDK throws the appropriate communication exception type, where the `InnerException` property contains the source exception, and its `IsRetryable()` method returns True.

CommunicationException Group

CLU SDK Communication exceptions are primarily utility classes providing encapsulation for multiple possible sources of communication errors you may encounter while completing an API call. The initial (source) exception can come from a number of underlying .NET framework components such as the http(s) protocol, or other exceptions in the `System.ServiceModel` or `System.Net` namespaces. See Microsoft's .NET documentation for more information about the .NET framework.

NusCommunicationException Type

NusCommunicationExceptions come from variety of sources, including exceptions from the `System.ServiceModel` and `System.Net` namespaces. The most common source type is the standard .Net type `System.Net.WebException`, where its `Response` and `Response.Status` properties contain critical information about the error state encountered.

As described in default retry strategy above, if the exception is classified as a communication type, the SDK will determine whether it is retryable, and if so, retry the offending call. If unsuccessful, the SDK throws a `NusCommunicationException`, where the `InnerException` property contains the source exception.

NusTimeoutException Type

Timeout exceptions are a kind of communication exception where the trigger exception is either `System.TimeoutException` or `System.Net.WebException`. The exception's `Response.StatusCode` property is set to `WebExceptionStatus.Timeout`.

Once the SDK classifies the source exception as a Timeout type, the system employs the default retry strategy. If the retry is unsuccessful, it throws a `NusTimeoutException` where the `InnerException` property contains the source exception.

System/Application Exceptions Group

NusSystemException Type

This exception type encapsulates errors returned by the NUS server per session or service API call as a result of violation of business and/or data integrity rules. The exception data contains details on a specific request status and its details. By default, for exceptions in application group, the `IsRetryable ()` method returns False.

This is the most frequent type of exception to encounter during integration development.

GetExtractedDocument() Example

Executing `GetExtractedDocument()` for a non-existing document ID throws an exception with the following message:

```
Nuance.NUS.Sdk.NusSystemException: web service call GetExtractedDocument failed:

response code: REQUEST_ERROR

response description: CLU Document does not exist
```

SessionFactory.Open() Example

Supplying invalid credentials to `SessionFactory.Open()` throws an exception with the following message:

```
Nuance.NUS.Sdk.NusSystemException: web service call OpenSession failed: response
code: AUTHENTICATION_ERROR

response description: Generic application error. Please contact system administrator
```

NusException Type

These exceptions are generated by the CLU SDK client side code, and are specific to internals of creating session and service instances and service type resolution. Typically, such exceptions are rare and likely an indicator of SDK deployment issues, such as missing service specific dll files and/or resources.

NusSystemException Type

These exceptions cover errors returned by the NUS server related to internal message exchange protocol and data formats. The `Exception.Message` property contains a complete description of error encountered, and any additional data if available. Typically, such exceptions are rare and an indication of internal CLU SDK messaging problems.

NusAuthorizationException Type

These exceptions occur as a consequence of compromised authorization data and/or process, such as corrupt security context data, or prolonged inactivity of the client system resulting in exceeding the server idle timeout limitation.

Exception Handling Guidelines

The `try/catch/finally` pattern is the safest approach for handling processing errors and allowing your client application to either maintain its valid state or exit in an orderly fashion while preserving all information about the erroneous state.

While handling `NusSystemException` cases (violations of business and/or data rules), you might choose to inform the user about possible remedial action. For example, if a user tries to retrieve a non-existing document ID, you might want to prompt for correction of input data.

In case of `NusTimeoutExceptions` , you should increase the setting for maximum timeout per API call (as defined in `ConnectionConfig.Timeout` property) before examining other influencing factors such as network latency and the size and number of documents processed via SDK.

Handling Exceptions Programmatically

You can handle some of exceptions programmatically, attempting remedial action in your application code. For example, programmatic exception handling is a good choice for dealing with `NusAuthorizationExceptions`, when an instance of session went through a long period of inactivity and exceeded the NUS server idle timeout.

Exceeding the timeout causes every subsequent API call to throw a `NusAuthorizationException`. Its `Response.Code` property will always be `RequestStatus.AuthorizationError`, while `Response.Description` and `Response.MethodName` vary depending on the source invocation context.

To handle such exceptions, use a three step strategy:

1. Detect the `NusAuthorizationException` during an API call.
2. Recreate the session and respective service instance.
3. Retry the original API call.

The following example shows this strategy:

360 | Understanding Services Workflows

The 360 | Understanding Services SDK supports the following workflows:

- Synchronous fact extraction, where you submit a document to Nuance's fact extraction engine via a single API call, and must wait for that processing to complete before you can continue.
- Asynchronous fact extraction, where you can better control latency in the system by using different API calls to submit, check, and retrieve documents from the fact extraction engine.

Synchronous Fact Extraction

Synchronous fact extraction is the simplest workflow. At its core, it is a single API call executing multiple operations; it:

- Submits a single document (in text form) and its metadata to the engine.
- Waits for the extraction process to complete.
- Receives the extraction results in CDA XML format. The data returned from this call contains a unique document identifier within the processing queue. This identifier is an integral part of the Document Management functionality. See *Document Management* on page 52 for more information.

Depending on size and complexity of the document in process, response times vary.

The client application is responsible for handling the latency of sending the file, and processing and receiving the extraction results.

The extraction results are returned as a document stream in CDA format, so it is responsibility of the client application to parse and reconcile resulting CDA document into their specific representation of the updated document.

See *About the XML Output* on page 67

When to use the Synchronous Fact Extraction Workflow

This workflow is primarily geared towards integration scenarios where a separate process executes synchronous extraction calls, isolating the client application UI (if any) from potential adverse effects. An example would be processing documents in batch fashion, either sequentially, or as concurrent tasks[1].

[1]Concurrent processing refers to client application's ability to launch and manage execution of either multiple threads or multiple processes.

Synchronous Fact Extraction Example

The following sample shows how to implement the Synchronous Fact Extraction workflow:

```
        using System;
        using System.IO;
        using System.Text;
        using Nuance.NUS.Sdk.Model;
        using NUnit.Framework;

        namespace Nuance.NUS.Sdk.Clu.Examples
        {
            [TestFixture]
            public class SynchronousWorflowExample
            {
                #region arguments
                // minimal contents of required arguments
                // metadata
                private readonly DocumentMetaDataDetail  _docMetaData =
                    new DocumentMetaDataDetail
                        {
                            ExternalDocumentId = "123",
                            DocumentType = DocumentType.Original,
                            AuthorUserName = "author",
                            BusinessEntityShortName = "Training",
                            WorkTypeCode = "41"
                        };
                // patient
                private readonly Patient  _patient =

                new Patient
                    {
                        LastName = "Doe",
                        BirthDate = new DateTime( 1955, 12, 5 ),
                        MedicalRecordNumber = "MRN-123"
                    };
                // visit
                private readonly Visit _visit = new Visit {VisitCode = "4321"};
                #endregion

                [Test]
                public void SynchronousWorkflow()
                {
                    // configure connection; an emulated connection in this case
                    var config = new Configuration
                        {
                            SelectedConnection = new ConnectionConfig
                                {
                                    Type = ConnectionType.SessionEmulator,
                                    Name = "Emulated connection"
                                }
                        };

                    // populate license information
                    var license = new License
                        {
                            PartnerId = "50B1A16A-DEF7-48BE-9FA2-A49AC68BB72F",
                            Id = new Guid( "7B0E51D7-30F1-4315-A79D-3E57A60DA671" ),
                            CustomerName = "CustomerName",
                        };

                    var factory = new SessionFactory( config );

                    using( var session = factory.Open( "userId", license )) // create session
                    {
                        Assert.IsNotNull( session );
                        using( var service = session.GetService<ICluService>() ) // create service

                        {
                            Assert.IsNotNull( service );
                            using( var docStream = GetDocStream() ) // open doc as stream
                            {
                                var response = service.ExtractDocument( docStream, _docMetaData, _patient, _visit );
                        // execute extraction

                        // verify results
                        Assert.IsTrue( response.DocumentId != -1,
                            "Document id should be non-negative." );
                        Assert.IsTrue( response.ExtractedDocument.Length > 0,
                            "Extracted document length should be greater than 0" );

                    }
                }
            }
        }

        private static Stream GetDocStream()
        {
            // create stream

            // either as simple as from file system
            // File.OpenRead( "your file name / path here" )

            // or memory stream,
            var stream = new MemoryStream();
            var buffer = Encoding.ASCII.GetBytes( "Document to be extracted text ... " );
            stream.Write( buffer, 0, buffer.Length );
            stream.Position = 0;
            return stream;
        }
    }
}
```

Asynchronous Fact Extraction

The asynchronous fact extraction workflowoffers more flexibility in managing latency, but is more complex than the synchronous workflow.With low execution overhead per API call, it is easier to manage timeliness and responsiveness of client application UI.

Asynchronous fact extraction requires three distinct operations and corresponding API calls to comple the fact extraction sequence for a document:

- Submit–Places the document and its metadata into a service processing queue, and immediately returns control to the calling process. The data returned from this call contains a unique document identifier within the processing queue. This identifier is an integral part of the Document Management functionality. See *Document Management* on page 52 for more information.
- Check Status –Using the document identifier from the submission method as an argument, the API returns the status of the specified document.
- Retrieve –Upon receiving a status of completion in previous step, the API call returns the extraction results.

As in the synchronous workflow, the extracted document results are delivered as a document stream in CDA format. The client application is responsible for parsing and reconciling the resulting CDA document into their specific representation of the updated document.

See *About the XML Output* on page 67

Asynchronous Fact Extraction Example

The following sample code shows how to implement the Asynchronous Fact Extraction workflow:

```
using System;
using System.Linq;
using System.IO;
using System.Text;
using System.Threading;
using Nuance.NUS.Sdk.Model;
using Nuance.NUS.Sdk.Util;
using NUnit.Framework;

namespace Nuance.NUS.Sdk.Clu.Examples
{
        [TestFixture]
        public class AsynchronousWorkflowExample
        {
                #region arguments
                // minimal contents of required arguments
                // metadata
                private readonly DocumentMetaDataDetail  _docMetaData =
                        new DocumentMetaDataDetail
                                {
                                        ExternalDocumentId = "123",
                                        DocumentType = DocumentType.Original,
                                        AuthorUserName = "author",
                                        BusinessEntityShortName = "Training",
                                        WorkTypeCode = "41"
                                };
                // patient
                private readonly Patient _patient =
                        new Patient
                                {
                                        LastName = "Doe",
                                        BirthDate = new DateTime( 1955, 12, 5 ),
                                        MedicalRecordNumber = "MRN-123",
                                };
                // visit
                private readonly Visit _visit = new Visit {VisitCode = "4321"};
                #endregion

                // the methods below illustrate steps and code needed in client application
                // to engage asynchronous document extraction

                // workflow variables
                private INusSession _session;
                private ICluService _service;
                private int _docId; // document id of the document in workflow

                [TestFixtureSetUp] // workflow startup routine
                public void StartWorkflow()
                {
                        // configure connection; an emulated connection in this case
                        var config = new Configuration
                                {
                                        SelectedConnection = new ConnectionConfig
                                                {
                                                        Type = ConnectionType.SessionEmulator,
                                                        Name = "Emulated connection"
                                                }

                                };

                        // populate license information
                        var license = new License
                        {
                                PartnerId = "50B1A16A-DEF7-48BE-9FA2-A49AC68BB72F",
                                Id = new Guid( "7B0E51D7-30F1-4315-A79D-3E57A60DA671" ),
                                CustomerName = "CustomerName",
                        };

                var factory = new SessionFactory( config );
                _session = factory.Open( "userid", license );
                _service = _session.GetService<ICluService>();
        }
        [TestFixtureTearDown] // workflow cleanup routine
        public void EndOfWorkflow()
        {
                _service.Dispose();
                _session.Dispose();
        }

        [Test(Description = "Step1: Submit document")]
        public void AStep1()
        {
                using( var docStream = GetDocStream() )
                {
                        var response = _service.ImportDocument( docStream, _docMetaData, _patient, _visit );

                        Console.WriteLine("jobResponse: {0}", response );
                        Assert.IsTrue( response.DocumentId != -1 );

                        _docId = response.DocumentId; // record document id returned
                }
        }

        [Test( Description = "Step2: Check status of the document")]
        public void BStep2()
        {
                var status = JobStatus.Unknown;

                while (status != JobStatus.Complete )
                {
                        var statusArr = _service.GetDocumentListStatus( new[] {_docId} ); // retrieve status array

                        status = statusArr.First().JobStatus;
                        Console.WriteLine( "Status check: {0}", statusArr.First() );

                        Thread.Sleep( 300 ); // pause before another poll
                }

                CStep3(); // continue to step 3
        }

        [Test( Description = "Step 3: Retrieve extracted document"), Ignore]
        public void CStep3()
        {
                var response = _service.GetExtractedDocument( _docId ); // retrieve workflow  document


                // verify results
                Assert.IsTrue( response.DocumentId != -1,
                        "Document id should be non-negative." );
                Assert.IsTrue( response.ExtractedDocument.Length > 0,
                        "Extracted document length should be greater than 0" );
        }

        private static Stream GetDocStream()
        {


                // create stream

                // either as simple as from file system
                // File.OpenRead( "your file name / path here" )

                // or memory stream,
                var stream = new MemoryStream();
                var buffer = Encoding.ASCII.GetBytes( "Document to be extracted text ... " );
                stream.Write( buffer, 0, buffer.Length );
                stream.Position = 0;
                return stream;
        }
}
```

Document Management

Document management (DM) functionality is an integral part of every document submission/extraction cycle. It applies to both the synchronous and the asynchronous fact extraction workflows.

DM facilitates document lifecycle management by tracking document versions, and by tracking the relationships between documents that are processed at various stages of client application workflow.

The DM identifies and treats documents according to their type:

- Source Document– The initial version of medical document entering the system.
- Amendment – Update(s) to a Source Document that has not been signed yet. An amended document replaces its previous version.
- Addendum – A distinct document that is related to a previously submitted document, but which does not replace the previous document.

The document lifecycle sequence requires that a Source Document exist before before any Amendment or Addendum documents can be submitted. The client application logic is responsible for adhering to this sequence.

The sequence is also important in maintaining the relationship between subsequent document submissions. Each time you use the APIs to submit a document, the CDA output contains a unique document identifier. The client application is then responsible for specifying a related/predecessor document identifier for each subsequent submission of amendments or addendums to the source document.

The CLU Service Interface

The CLU Service interface is a collection of API methods that support the workflows described in *360 | Understanding Services Workflows* on page 47. Figure 2 shows a class diagram for this interface, and Figure 3 shows a diagram of the interface's arguments:

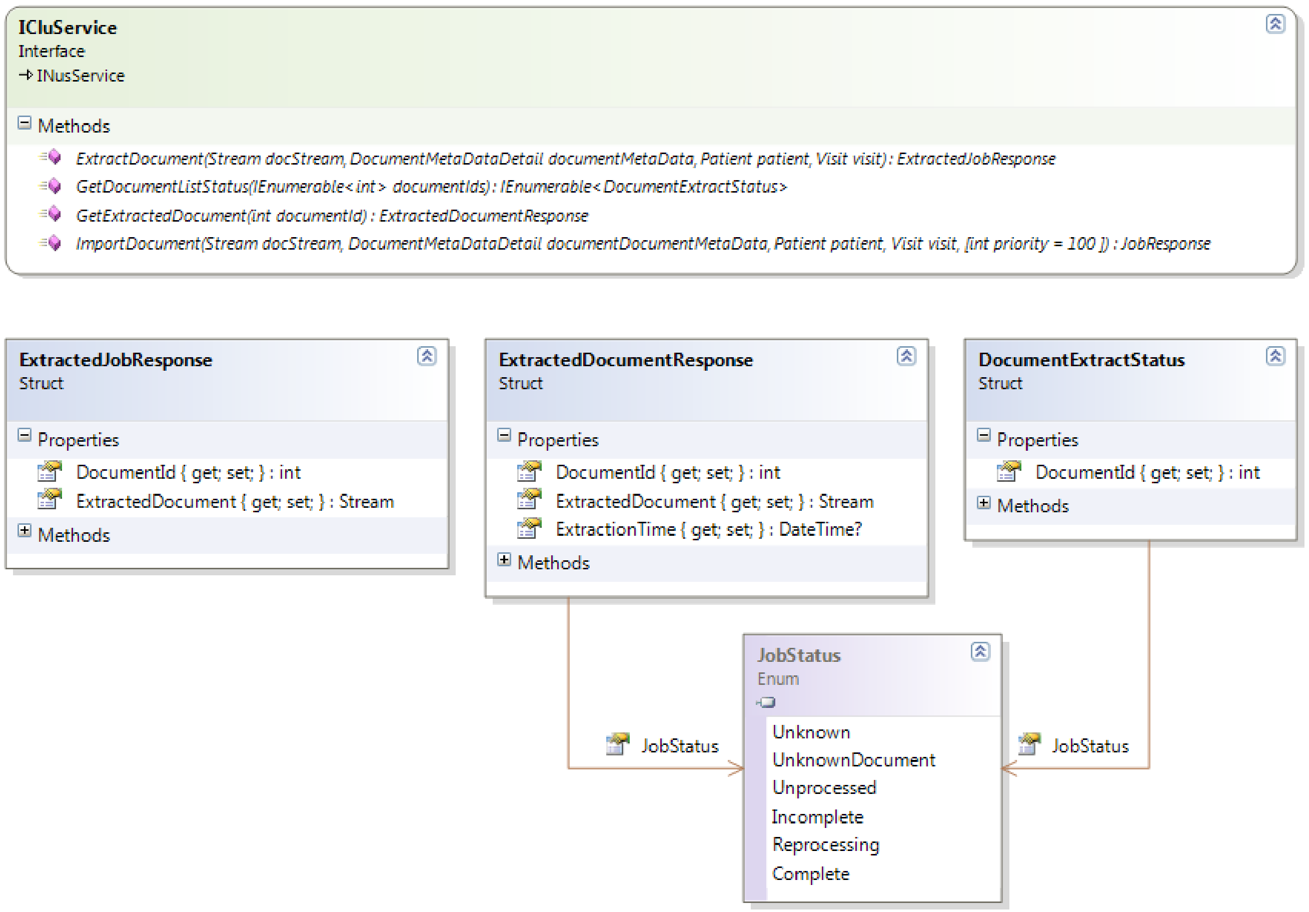
ICluService
Interface
INusService
Methods
ExtractDocument(Stream docStream, DocumentMetaDataDetail documentMetaData, Patient patient, Visit visit) : ExtractedJobResponse
GetDocumentListStatus(IEnumerable<int> documentIds) : IEnumerable<DocumentExtractStatus>
GetExtractedDocument(int documentId) : ExtractedDocumentResponse
ImportDocument(Stream docStream, DocumentMetaDataDetail documentDocumentMetaData, Patient patient, Visit visit, [int priority = 100]) : JobResponse
ExtractedJobResponse
Struct
Properties
DocumentId { get; set; } : int
ExtractedDocument { get; set; } : Stream
Methods
ExtractedDocumentResponse
Struct
Properties
DocumentId { get; set; } : int
ExtractedDocument { get; set; } : Stream
ExtractionTime { get; set; } : DateTime?
Methods
DocumentExtractStatus
Struct
Properties
DocumentId { get; set; } : int
Methods
JobStatus
JobStatus
Enum
Unknown
UnknownDocument
Unprocessed
Incomplete
Reprocessing
Complete
JobStatus

Figure 2: The CLU Service Interface

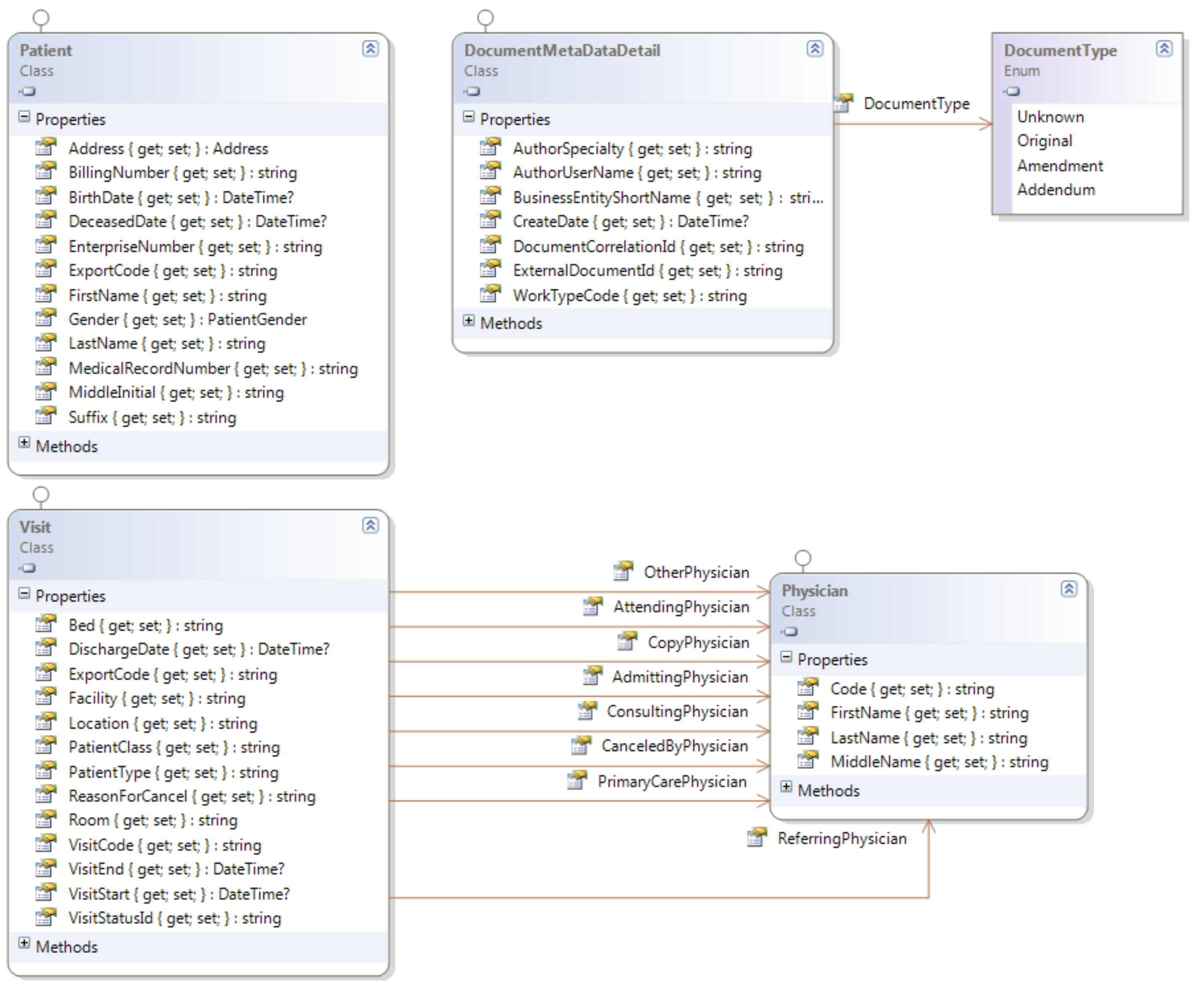
Patient
Class
Properties
Address { get; set; } : Address
BillingNumber { get; set; } : string
BirthDate { get; set; } : DateTime?
DeceasedDate { get; set; } : DateTime?
EnterpriseNumber { get; set; } : string
ExportCode { get; set; } : string
FirstName { get; set; } : string
Gender { get; set; } : PatientGender
LastName { get; set; } : string
MedicalRecordNumber { get; set; } : string
MiddleInitial { get; set; } : string
Suffix { get; set; } : string
Methods
DocumentMetaDataDetail
Class
Properties
AuthorSpecialty { get; set; } : string
AuthorUserName { get; set; } : string
BusinessEntityShortName { get; set; } : stri...
CreateDate { get; set; } : DateTime?
DocumentCorrelationId { get; set; } : string
ExternalDocumentId { get; set; } : string
WorkTypeCode { get; set; } : string
Methods
DocumentType
DocumentType
Enum
Unknown
Original
Amendment
Addendum
Visit
Class
Properties
Bed { get; set; } : string
DischargeDate { get; set; } : DateTime?
ExportCode { get; set; } : string
Facility { get; set; } : string
Location { get; set; } : string
PatientClass { get; set; } : string
PatientType { get; set; } : string
ReasonForCancel { get; set; } : string
Room { get; set; } : string
VisitCode { get; set; } : string
VisitEnd { get; set; } : DateTime?
VisitStart { get; set; } : DateTime?
VisitStatusId { get; set; } : string
Methods
OtherPhysician
AttendingPhysician
CopyPhysician
AdmittingPhysician
ConsultingPhysician
CanceledByPhysician
PrimaryCarePhysician
ReferringPhysician
Physician
Class
Properties
Code { get; set; } : string
FirstName { get; set; } : string
LastName { get; set; } : string
MiddleName { get; set; } : string
Methods

Figure 3: CLU Service Interface Arguments

The following tables describe the CLU Service Interface's methods and arguments:

Workflow Type	Method	Description
Synchronous	`ExtractDocument()`	Performs document extraction in a single call. This includes submitting a document, awaiting processing, and returning extracted document results.
Asynchronous	`ImportDocument()`	Submits a document into service processing queue and returns the submitted document identifier.
Asynchronous	`GetDocumentListStatus ()`	Checks the document status

Workflow Type	Method	Description
		in the service processing queue and returns an indication of whether the document is processed or not.
Asynchronous	`GetExtractedDocument()`	Retrieves the extracted document.

Patient Attibutes

Attribute	Type	Character Constraint	Optional	Description
`MedicalRecordNumber`	string		N	Medical record number in the EHR.
`ExportCode`	string		Y	Patient code for the EHR alternate system.
`EnterpriseNumber`	string		Y	Identifier used by the EHR to reference the patient regardless of facility.
`BillingNumber`	string		Y	The patient's billing account number.
`LastName`	string		Y	The patient's last name.
`MiddleInitial`	string		Y	The patient's middle initial.
`Suffix`	string		Y	Patient's title. For example, Jr. or M.D.
`PatientGender`	enumeration		Y	The patient's gender. Valid values are `Male` and `Female`, or `Unknown` if the value is not set.
`BirthDate`	dateTime		Y	The patient's birth date.
`Address`	Address		Y	The patient's address.
`DeceasedDate`	dateTime		Y	The patient's death date.

Address Attibutes

Attribute	Type	Optional
`street1`	string	Y
`street2`	string	Y
`city`	string	Y
`state`	string	Y
`zip`	string	Y
`country`	string	Y
`phone`	string	Y

Physician Attibutes

Attribute	Type	Optional	Description
`code`	string	Y	Unique physician identifier in CLU system.
`lastName`	string	Y	Physician's last name.
`firstName`	string	Y	Physician's first name.
`middleName`	string	Y	Phisician's middle name.

Visit Attributes

Attribute	Type	Character Constraint	Optional	Description
`PatientClass`	string		Y	The EHR code for describing classes of patients. For example, inpatient, emergency, pre-admit.
`PatientType`	string		Y	The EHR code for describing types of patients.
`ExportCode`	string		Y	The EHR visit code for an alternate system.
`VisitCode`	string		Y	The EHR identifier for this patient visit.
`VisitStart`	dateTime		Y	The date and time of the start of the visit.
`VisitEnd`	dateTime		Y	The date and time of the end of the visit.
`VisitStatusId`	string		Y	Current status of patient visit.
`Location`	string		Y	For in-patient stays, nursing station or other location of patient.
`Facility`	string		Y	For multi-facility institutions, the facility is where the patient is located.
`Room`	string		Y	For in-patient stays, patient room number.
`Bed`	string		Y	For in-patient stays, patient bed identifier.
`ReasonForCancel`	string		Y	Reason for canceling.
`DischargeDate`	dateTime		Y	The date and time that the patient was discharged.
`AdmittingPhysician`	physician		Y	The admitting physician.
`AttendingPhysician`	physician		Y	The attending physician.
`ReferringPhysician`	physician		Y	The referring physician.
`ConsultingPhysician`	physician		Y	The consulting physician.
`PrimaryCarePhysician`	physician		Y	The primary care physician.

Attribute	Type	Character Constraint	Optional	Description
CanceledByPhysician	physician		Y	
CopyPhysician	physician		Y	
OtherPhysician	physician		Y	

Document Metadata Attributes

Name	Type	Optional	Character Constraint	Description
externalDocumentId	String	N	255	The calling application's document ID.
documentCorrelationId	String	Y	255	Identifies the externalDocumentId of the previous/latest version of the document required for amendments or addendums.
documentType	Enumeration	N		Specifies if the document is a source, amendment, or addendum with a document's version trail.
createDate	dateTime	Y		The report creation date in the document management system.
businessEntityShortname	String	N	25	The business entity name within the institution, important for audit trail reconciliation. This is a provisioned value dependency.
WorktypeCode	String	N	4	The institution worktype corresponding to the imported document that will map to a MUSE worktype within the CLU middle tier.[1]Note if this value is not known 'Unknown' may be specified.
authorUsername	String	N	30	The author of the document. This username will be auto-provisioned if it does not exist.
authorSpecialty	String	Y		The author's specialty. The values are defined in the CLU middle tier and must be an existing valid value as such. They rarely change and are exchanged manually with document management systems,

[1]The source worktype may be a CLU worktype with a 1-1 mapping in the middle-tier.

Name	Type	Optional	Character Constraint	Description
				such as is the case with work-types.

The following sections describe the methods in greater detail.

ExtractDocument()

Description

Supports the Synchronous Document Extraction workflow. Performs document extraction in a single call. This includes submitting a document, awaiting processing, and returning extracted document results.

Syntax

```
ExtractDocument(Stream docStream, DocumentMetaDataDetail
docMetaData, Patient patient,Visit visit):ExtractedJobResponse
```

Parameters

Parameter	Description
`docStream`	Stream of the document that you want to process.
`docMetaData`	A `DocumentMetadataDetail` object containing metadata for the document that you want to process. See *The CLU Service Interface* on page 54 for more information about the `DocumentMetadataDetail` class and its arguments.
`patient`	A `patient`object containing information about the patient associated with the document that you are processing. See *The CLU Service Interface* on page 54 for more information about the `patient` class and its arguments.
`visit`	A `visit`object containing information about the visit associated with the document that you are processing. See *The CLU Service Interface* on page 54 for more information about the `visit` class and its arguments.

Returns

`ExtractedJobResponse`, which includes:

- `documentId` - A unique identifier for this document.
- The CDA XML stream of `ExtractedDocument`. See *About the XML Output* on page 67

Example

The `ExtractDocument()` method is shown in yellow:

```
var factory = new SessionFactory( config );

        using( var session = factory.Open( "userId", license )) // create session
        {
                 // create service
                 using( var service = session.GetService<ICluService>() )
                 {
```

```
                using( var docStream = GetDocStream() ) // open doc as stream
                {
                        // execute extraction
                        var response = service.ExtractDocument( docStream,
                                _docMetaData, _patient, _visit );

                        // verify results
                        Assert.IsTrue( response.DocumentId != -1,
                                "Document id should be non-negative." );
                        Assert.IsTrue( response.ExtractedDocument.Length > 0,
                                "Extracted document length should be
                                 greater than 0" );
                }
        }
}
```

ImportDocument()

Description

Submits a document into service processing queue. When processing is complete, returns a document identifier.

Syntax

```
ImportDocument(Stream docStream, DocumentMetaDataDetail doc-
umentMetaData, Patient patient, Visit visit, [int pri-
ority=100]): JobResponse
```

Parameters

Parameter	Description
`docStream`	Stream of the document that you want to process.
`docMetaData`	A `DocumentMetadataDetail` object containing metadata for the document that you want to process. See *The CLU Service Interface* on page 54 for more information about the `DocumentMetadataDetail` class and its arguments.
`patient`	A `patient`object containing information about the patient associated with the document that you are processing. See *The CLU Service Interface* on page 54 for more information about the `patient` class and its arguments.
`visit`	A `visit`object containing information about the visit associated with the document that you are processing. See *The CLU Service Interface* on page 54 for more information about the `visit` class and its arguments.
`priority`	Sets the document processing priority. An integer from 1 to 100. 100 is the lowest priority, 1 the highest. The default value is 100.

Returns

A `JobResponse` object containing the following information:

- `documentId` - A unique identifier for this document.
- `ExtractedDocument` - The CDA XML stream of `ExtractedDocument`. See *About the XML Output* on page 67

Example

```
[Test(Description = "Step1: Submit document")]
       public void AStep1()
       {
               using( var docStream = GetDocStream() )
               {
```

```
                        var response = _service.ImportDocument( docStream,
                            _docMetaData, _patient, _visit );

                        Console.WriteLine("jobResponse: {0}", response );
                        Assert.IsTrue( response.DocumentId != -1 );

                        _docId = response.DocumentId; // record document id returned
                }
        }
```

GetDocumentListStatus()

Description

Retrieves the status of the specified document.

Syntax

```
GetDocumentListStatus(IEnumerable<int> doc-
umentIds):IEnumerable<DocumentExtractStatus>
```

Parameters

Parameter	Description
`documentIDs`	An enumeration of document identifiers. Document identifiers are returned by *ImportDocument()* on page 62.

Returns

An enumeration containing the status of the specified documents. Valid values are:

Value	Description
`Unknown`	The document's status is unknown.
`UnknownDocument`	The document ID does not correspond to a known document.
`Unprocessed`	The document is unprocessed.
`Incomplete`	The document processing is incomplete.
`Reprocessing`	The document is being reprocessed.
`Complete`	Document processing is complete.

Example

```
[Test( Description = "Step2: Check status of the document")]
                public void BStep2()
                {
                        var status = JobStatus.Unknown;

                        while (status != JobStatus.Complete )
                        {
                                var statusArr = _service.GetDocumentListStatus(
                                        new[] {_docId} ); // retrieve status array
                                status = statusArr.First().JobStatus;

                                Console.WriteLine( "Status check: {0}",
                                        statusArr.First() );
```

```
                    Thread.Sleep( 300 ); // pause before another poll
                }

                CStep3(); // continue to step 3
            }
```

GetExtractedDocument()

Description

Retrieves the document extraction results in CDA XML format.

Before calling `GetExtractedDocument()`, you must first check the document status with *GetDocumentListStatus()* on page 64

Syntax

```
ExtractedDocumentResponse GetExtractedDocument( int documentId );
```

Arguments

Parameter	Description
`documentID`	The `documentId` returned by *ImportDocument()* on page 62.

Returns

The `ExtractedDocumentResponse` enumeration, which includes:

- `documentId` - A unique identifier for this document.
- `JobStatus` - The status of this document's processing. See *GetDocumentListStatus()* on page 64 for a table of valid statuses.
- `ExtractionTime` - Date and timestamp when the document completed processing.
- `ExtractedDocument` - The CDA XML stream of `ExtractedDocument`. See *About the XML Output* on page 67

Example

```
[Test( Description = "Step 3: Retrieve extracted document"), Ignore]
       public void CStep3()
       {
               // retrieve workflow document
               var response = _service.GetExtractedDocument( _docId );

               // verify results
               Assert.IsTrue( response.DocumentId != -1,
                       "Document id should be non-negative." );
               Assert.IsTrue( response.ExtractedDocument.Length > 0,
                       "Extracted document length should be greater than 0" );
       }
```

About the XML Output

The data extraction platform contains a number of components that can be assembled in a pipeline to perform a specific extraction task. The actual execution of that task is performed by a workflow engine that takes this pipeline as a configuration parameter.

When a document is processed by the engine, the result is captured in an XML structure. The structure of the engine's XML output conforms to the Clinical Document Architecture (CDA) v2 standard.

You use this XML output in conjunction with a set of XML templates that define the structure for various types of electronic medical records (EMRs).

The 360 | Understanding Services is a software development kit that supports eScription's data extraction capability

This document describes the XML templates, including:

- The general structure of the XML document.
- The standards used to create that structure.
- The information that the document contains.
- The standards by which this information is structured.
- Where this structure differs from the standards.
- How to retrieve this information using XPath.

The *eScription CLU SDK API Reference* document, included with the SDK, describes the CLU SDK API.

About the CDA Standard

The CDA standard is maintained by HL7's Structured Documents Technical committee. They define the CDA as:

“A document markup standard that specifies the structure and semantics of "clinical documents" for the purpose of exchange.”

CDA documents are expressed in XML.

Not all features of CDA are used in the CLU framework output format, so we will focus on those parts that are used. See the HL7 website at http://www.hl7.org/ for the full specifications.

About CDA Levels

CDA documents can be one of three levels. The CDA level of a document indicates the amount of structure in that document:

- Level 1: A document containing a structured header and a body part. The header contains information about the document itself (metadata), and the body describes the content of the document (the clinical report).
- Level 2: A Level 1 document in which the body part is subdivided into standardized sections.
- Level 3: A Level 2 document in which all clinical statements found in the report are added to the correct section.

The CLU engine outputs CDA level 3 documents, with sections and structured information.

It is important to note that all structured facts (level 3) CDA output, regardless of the document type used, is structured according to the HITSP/C32 standards. For example, a discharge summary will also contain structured facts according to the C32 standards.

About the Structured Header

The header of the Level 1 document contains the metadata of the document required for document discovery, management, and retrieval.

The metadata describes the document, and includes information like:

- Data about the document itself (unique ID, document type classification, version).
- Data about people and groups who are related to the record (providers, authors, patients).
- Data about document relationships (other documents, external images).

About the Body

The body structure contains the content of the clinical document itself. It consists of a sequence of sections that represent the sections in the clinical document.

Each section in the body structure contains a required textual part, composed of the free text as given in the original input document, and an optional structured part, composed of any information extracted from the textual part during processing of that document.

This structured part represents the information retrieved from the textual part in a consistent way, relying on standard coding systems such as SNOMED, LOINC, ICD-9, etc. It can also reference exactly which part of the textual block it represents by means of token/content annotations used in the textual part.

The textual part may contain content annotations used for reference in the structured part, as well as layout tags such as paragraphs, lists, items, emphasis, etc.

Data Extracted by the CLU Pipeline

The CLU pipeline can extract many types of data, including:

- problems (findings)
- procedures
- vital signs
- laboratory results
- social history
- allergies
- medications

Guidelines for Using XPath in .NET

The .NET framework offers a convenient way to parse XML documents via its XPath implementation's key classes: XPathDocument, XPathNavigator, and XPathNodeIterator.

When parsing a document with multiple namespaces declared, such as CDA document, namespace name (alias) must to be used in XPath queries to correctly resolve XML node traversals.

For example, namespace `"urn:hl7-org:v3"` is a default namespace, but it does not have an alias declared per se:

To ensure proper node traversal and query execution in that namespace, an alias has to be declared prior to query execution by using the `XmlNamespaceManager` class facility, and then used as an argument in `XPathNavigator.Select()` method.

The following example demonstrates a query of all "component" elements contained in CDA's "`structuredBody`" element: