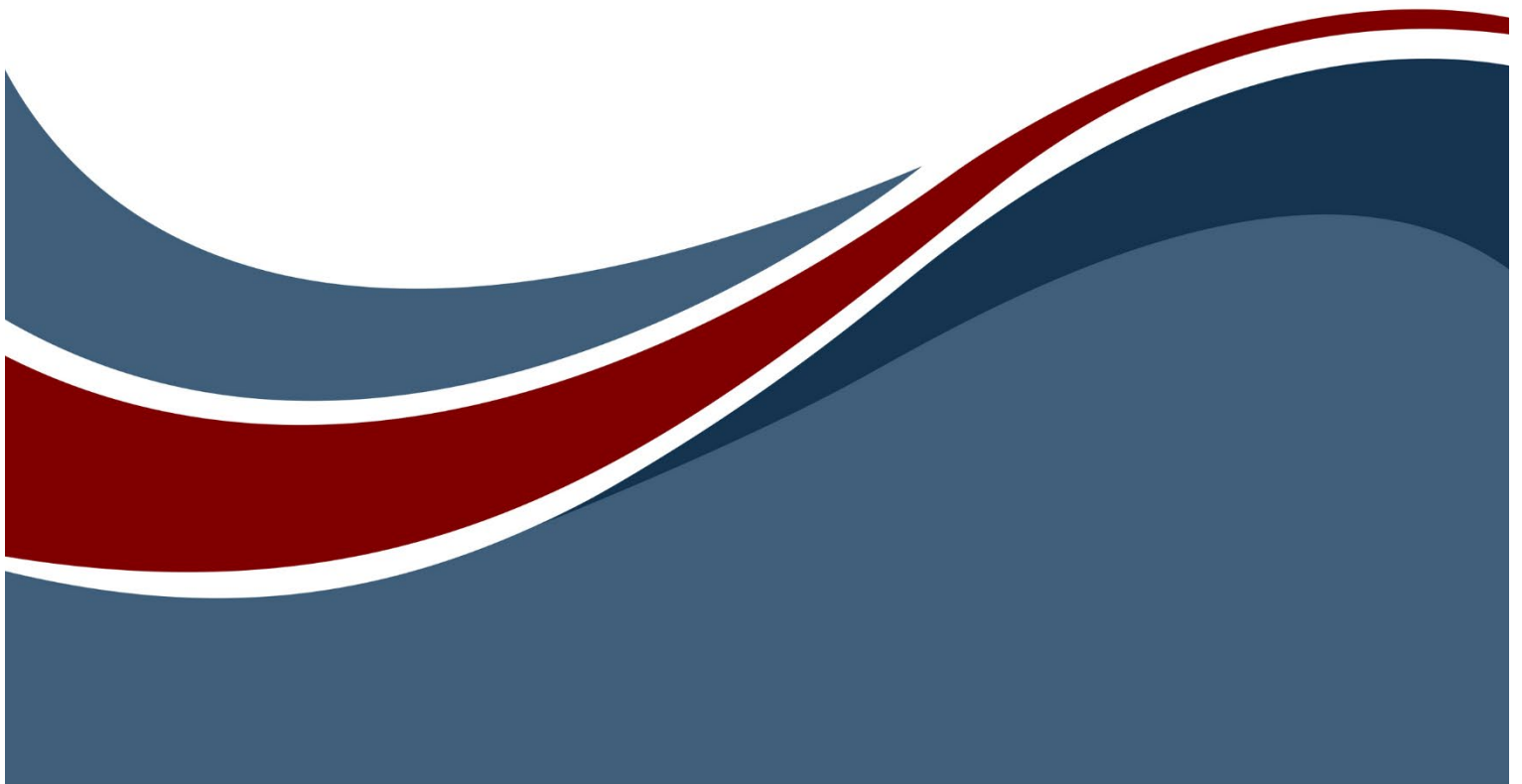




Getting Started with the GENESYS™ API



Getting Started with GENESYS™ API

Copyright © 2014-2024 Zuken Vitech Inc. All rights reserved.

No part of this document may be reproduced in any form including, but not limited to, photocopying, language translation, or storage in a data retrieval system, without Vitech's prior written consent.

Restricted Rights Legend

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in the applicable GENESYS End-User License Agreement and in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.277-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights at 48 CFR 52.227-19, as applicable, or their equivalents, as may be amended from time to time.

Zuken Vitech Inc.

2270 Kraft Drive, Suite 1600
Blacksburg, Virginia 24060
+1 540 951 3322 | Fax: +1 540 951 8222
www.vitechcorp.com

Customer Support:

+1 540 951 3999 | support@vitechcorp.com



is a trademark of Zuken Vitech Inc. and refers to all products in the GENESYS software product family.

The license and/or entitlement management portions of GENESYS are based upon one or more of the following copyrights: Sentinel® EMSaaS, Sentinel® LDK. Copyright © 2024 Thales. All rights reserved.

Sentinel® is a registered trademark of Thales. Other product names mentioned herein are used for identification purposes only and are trademarks of their respective companies.

Publication Date: December 2024

TABLE OF CONTENTS

Getting Started 1

Note to Programmers 1

Basic GENESYS Architecture 2

Basic GENESYS Data Structures Overview 3

Goals and Objectives 4

Logging Into GENESYS 5

Navigating the GENESYS Model 7

Reading a GENESYS Entity 8

Creating a GENESYS Entity 9

Modifying a GENESYS Entity 11

Logging Out of GENESYS 12

Deployment 12

Where to Find Further Help 12

APPENDIX A: PreRequisites, Assemblies, and Configuration 13

 .NET Framework Prerequisites 13

 Key GENESYS API Assemblies 13

 Vitech.Genesys 13

 Vitech.Genesys.Client 13

 Vitech.Genesys.Common 13

 Vitech.Genesys.Logging 13

 Vitech.Genesys.Random 13

 Licensing Assemblies 13

 App.config 14

 Upgrading to a new version 14



CUSTOMER RESOURCE OPTIONS

Supporting users throughout their entire journey of learning model-based systems engineering (MBSE) is central to Vitech’s mission. For users looking for additional resources outside of this document, please refer to the links below. Alternatively, all links may be found at www.vitechcorp.com/online-resources/.



[Webinars](#)

Immense, on-demand library of webinar recordings, including systems engineering industry and tool-specific content.



[Screencasts](#)

Short videos to guide users through installation and usage of GENESYS.



[A Primer for Model-Based Systems Engineering](#)

Our free eBook and our most popular resource for new and experienced practitioners alike.



[Help Files](#)

Searchable online access to GENESYS help files.



[Technical Papers](#)

Library of technical and white papers for download, authored by Vitech systems engineers.



[Technical Support](#)

Frequently Asked Questions (FAQ), support-ticket web form, and information regarding email, phone, and chat support options.

GETTING STARTED

The GENESYS™ architecture is designed with an extensive public API as its fundamental architecture. The external application developer has full access to the same interfaces which the GENESYS thick client uses, providing full programmatic access to repository and project model data. Vitech follows standard, accepted Microsoft® .NET™ framework practices when it comes to managing public interfaces, resulting in a familiar look and feel to external application developers.

Vitech is a strong supporter of API development efforts, both within individual customer organizations and as open-source examples across the user community, in order to extend and fine-tune the benefits offered by the GENESYS platform. Repository and model data are fully available for virtually any application, whether it is reports, dashboards, “connectors” to other data models, or other user-facing solutions that add additional value to an organization’s systems engineering efforts.

The *Requirements Commenter Page* is a simple, but useful example of one of the many ways to add value to the fundamental GENESYS model. By narrowly focusing on one aspect of the systems model—**Requirements**—the *Requirements Commenter Page* allows a non-technical stakeholder to add to the model without the overhead of learning details of the highly technical modeling tool.

Specific requirements for writing a GENESYS API application are in Appendix A. All the necessary files, including the Microsoft® Visual Studio project files, necessary to compile this example can be found in the GENESYS install folder under the Samples\API Samples\ sub folder.

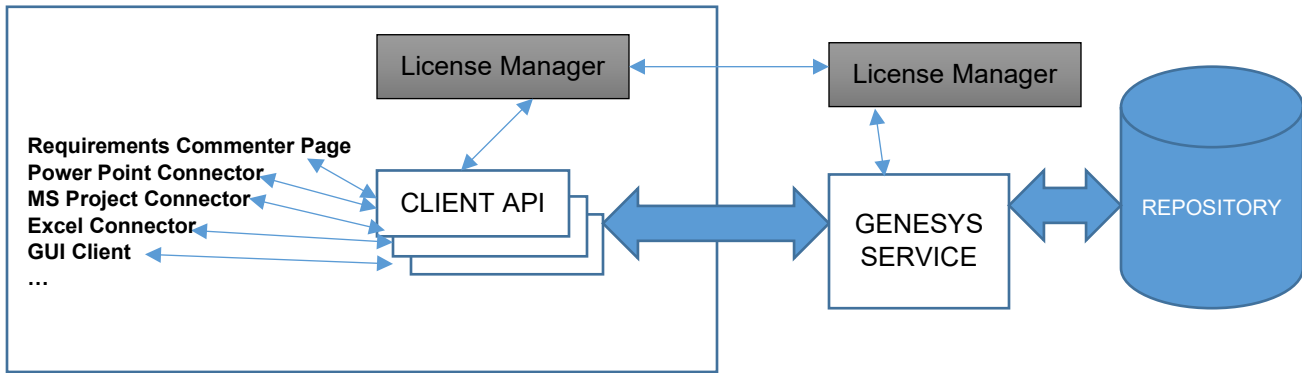
NOTE TO PROGRAMMERS

This document and the corresponding sample application are intended to give a basic introduction to the GENESYS API. As such, the code is written for brevity and clarity; the *try/catch* structures are minimal, and basic input and error checking may not be complete. This code is provided as a model for your own application; you can use this example by copying and pasting relevant sections from the document for your own purposes. The sample code referenced in this document can be found in any GENESYS installation, in the install folder, under the Samples\API Samples\RequirementCommenter\ sub folder. The GENESYS install directory path is by default C:\Program Files (x86)\Vitech\GENESYS {version} {edition}.

Since systems engineering and software engineering both have the concept of a class, which are related but distinct, this document will use the term **model class** when referring to a systems engineering class in the GENESYS schema, and simply **class** when referring to the software engineering concept of class. Another potential area of confusion is the term **entity**. In this document, **entity** will always refer to the model-based systems engineering concept of an entity, which is a particular instantiation of any model class within the model.

When developing an application for GENESYS using the API, it is useful to have a GENESYS client running on your second screen. Any updates you make to the model will show up immediately in the GENESYS client. To save the time required to build a test model, it is recommended that you import one of the sample projects—either the *SAMPLE: Geospatial Library*, or *SAMPLE: Fast Food*. This will give you a well-developed model on which to experiment. Make sure you have full permissions on the model you intend to use for testing.

BASIC GENESYS ARCHITECTURE



Your Machine

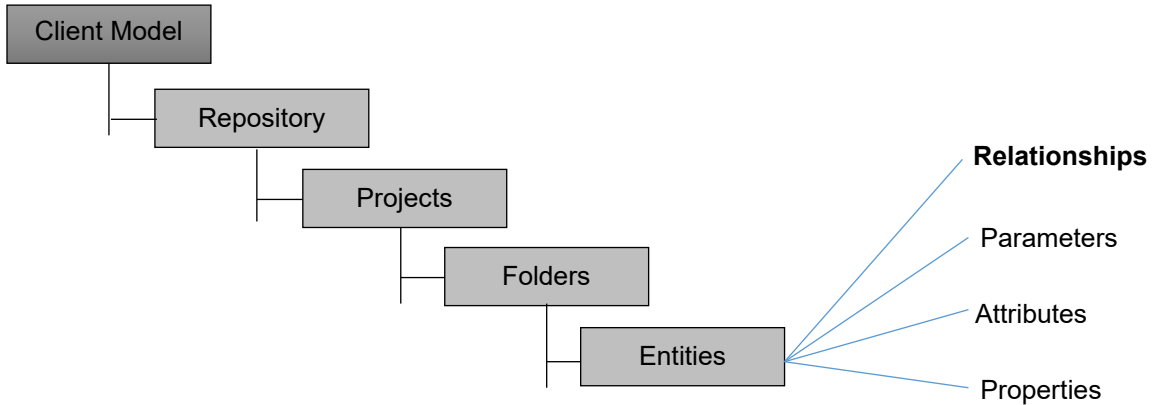
The *Requirements Commenter Page* program will use the same client API used by all other GENESYS clients. The Client API (a set of .NET 4.8 DLL files), and the Sentinel License Manager (a third-party component installed with GENESYS) need to reside on your development machine and need to be installed on the end user's machine when the application is deployed. The GENESYS host service, repository, and license may reside on any machine in your network domain. We recommend that you install GENESYS 2024 on your development machine. This will install all the components needed to create an ideal development environment. For specifics on the files and their installation, refer to Appendix A.

The Sentinel License Manager is third party software used to manage licenses for the GENESYS family of products. There are two licensing modes used by the client API: local and network. *Local* licensing forces the client to use only licenses local to the client API's machine. *Network* allows the user to select a license from any machine on your domain. Use the License Manager installed with your GENESYS product to switch between local and network modes. Whichever method you choose, the Sentinel License Manager is required to be running on the client API machine. To inspect the disposition of both local and network GENESYS licenses, point an internet browser at <http://localhost:1947>.

As a programmer, you will be working with the Client API. The Client API is accessed using the ClientModel object. These objects will allow you to connect to the known repositories on the network and provide the login method, which returns a RepositoryConfiguration object. This object is then used to access the GENESYS model data.

BASIC GENESYS DATA STRUCTURES OVERVIEW

The repository object is your entry point into the GENESYS models stored on that repository. There are several ways to navigate/view the model data in repositories. For this example, we will demonstrate the basic hierarchical view.



For the current project, the diagram above provides sufficient understanding to locate the entities you need to access.

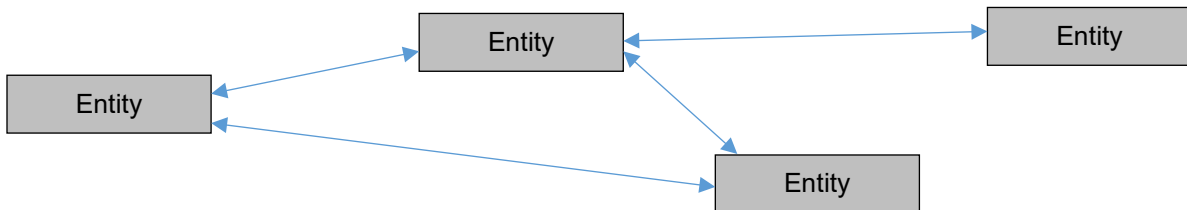
Project and top-level folder names are always unique. Folder names are also unique within a parent folder. Depending on how the individual project is set up, names of entities may **not** be unique. Entity IDs are **always** unique.

If you attempt to read an attribute of an entity you do not have permission to read, the phrase “Insufficient Privileges” will be returned.

Folders contain specific model classes. A top-level folder and all of its subfolders contain only one type of model class. An entity’s model class (type) is determined by which folder it was created in. Entities in a class can be moved between subfolders within the class but cannot be moved between classes (or folders in the data structure) without transforming the entity.

GENESYS has a sophisticated and finely grained permissions system, to the point that individual users may have permissions limited to particular entities. An entity for which you do not have read permissions as a GENESYS user will either not appear in the application dialog window or you will not be allowed to read of the entity’s attribute data, depending on what you are trying to read. GENESYS also has the ability to relate entities in one project to entities in another project. The entities referenced in another project will follow a different naming convention than the entities in the project on which you are working.

The other major way of navigating the GENESYS model is by navigating the relationships between entities. The entities and their relationships form a graph which can be navigated.

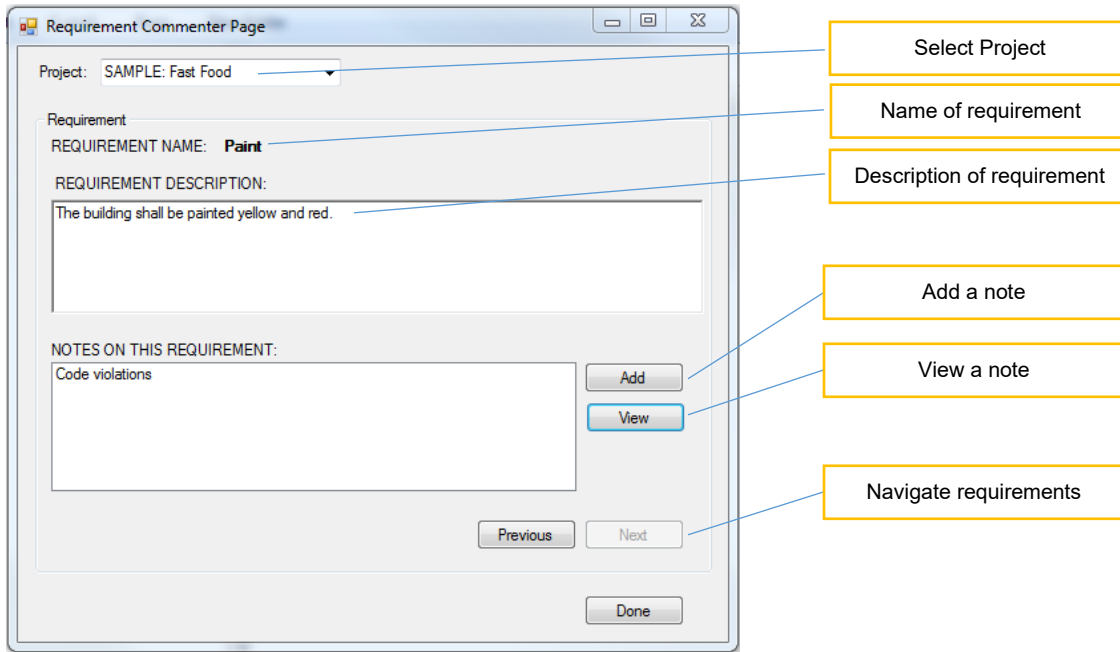


GOALS AND OBJECTIVES

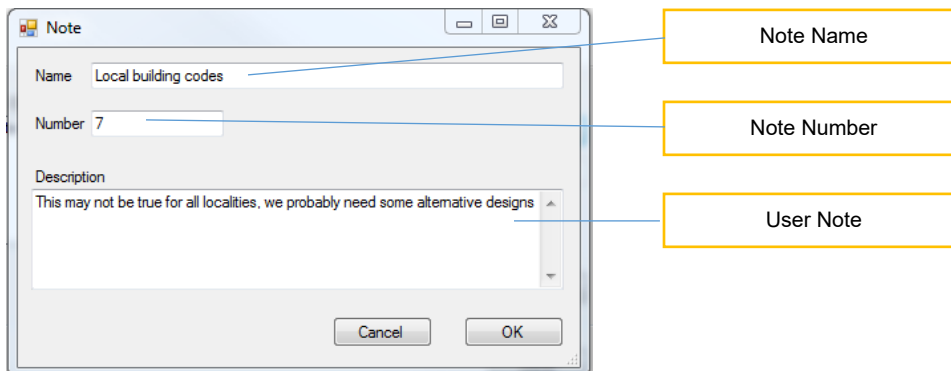
This example revolves around the following user story:

As a nontechnical project stakeholder, I want to review and comment on my project's requirements without having to learn any systems engineering or the GENESYS tool. I want to do this at my convenience.

To accomplish this, we will use the following screen to view requirements and any existing notes in turn, adding notes if needed.

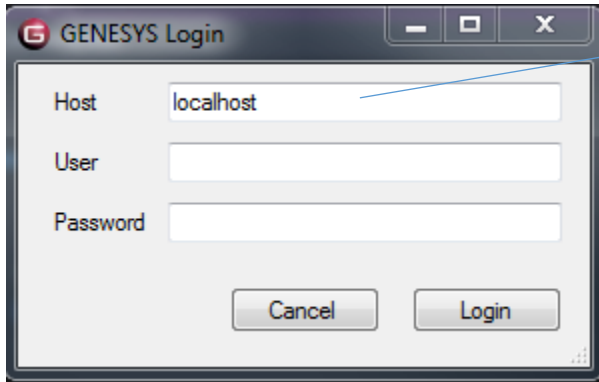


The dropdown at the top will allow the user to select any project the user has permissions to view. The "Next" and "Previous" buttons at the bottom will allow the user to view each requirement in sequence. The "View" button will allow the user to view the details of the note listed on its left. And the "Add" button will create a new note. The dialog presented to view/add a note is shown below.



LOGGING INTO GENESYS

The first task we need to do to accomplish our user story is to log our user into a GENESYS repository. The different repositories on your network are located using a host machine name and a host machine port.



Allows user to select different repositories by entering the host's machine name

The default port is 39151 and is rarely changed. If you have difficulty accessing a remote repository, check the firewall on both machines. If that does not help, verify the GENESYS service's default port by looking at the DefaultPort tag in the GenesysService.exe.config file located in the main GENESYS installation directory.

The following code shows the steps needed to obtain a valid *RepositoryConfiguration* object. This is the root object allowing access to the GENESYS repository which then gives you all you need to access the GENESYS repository to which you have just connected.

Getting Started with GENESYS™ API

Located in File *LoginDialog.cs*, method *btnLogin_Click()*.



```
private void btnLogin_Click(object sender, EventArgs e)
{
    string host = txtHost.Text.Trim();
    string userName = txtUser.Text.Trim();
    string password = txtPassword.Text;

    if (string.IsNullOrEmpty(userName) || string.IsNullOrEmpty(password) || string.IsNullOrEmpty(host))
        System.Windows.MessageBox.Show("A username and password must be specified.", "Login",
            System.Windows.MessageBoxButton.OK, System.Windows.MessageBoxImage.Error);

    if(host.Equals("localhost"))
    { // Get the configuration for the local repository.
        repositoryConfig = client.GetKnownRepositories().LocalRepository;
    }
    else
    { // Get the configuration for a remote repository.
        ConnectionInfo connInfo = new ConnectionInfo(host, 39151, ConnectionInfo.ProtocolType.NetTcp);
        repositoryConfig = client.GetKnownRepositories().Add(connInfo, false, "remote", userName);
    }

    // Set the credentials.
    // NOTE: Currently this is hard coded to use GENESYS AuthenticationType.
    GenesysClientCredentials credentials = new GenesysClientCredentials(userName, password,
        AuthenticationType.GENESYS);
    string errorMessage = null;

    try
    {
        // This operation will call the service to authenticate the user.
        repositoryConfig.Login(credentials);
    }
    catch(FaultException<UserNotFoundFault> dupFault) { errorMessage = "user not found"; }
    catch(FaultException<PasswordIncorrectFault> dupFault) { errorMessage = "bad password"; }
    catch(EndpointNotFoundException) { errorMessage = "Endpoint not found"; }
    catch(Exception ex) { errorMessage = "Invalid Login" + ex.Message; }

    if(errorMessage != null)
    {
        System.Windows.MessageBox.Show(errorMessage, "Login", System.Windows.MessageBoxButton.OK,
            System.Windows.MessageBoxImage.Error);
    }
    else
    {
        this.DialogResult = DialogResult.OK;
        Close();
    }
}
```

NAVIGATING THE GENESYS MODEL

Now that we have a *RepositoryConfiguration* object, the next thing needed for our story is to present the user with a list of projects from which to choose. To do this in the *Requirement Commenter Page*, we will display all projects in a repository and allow the user to select one from a drop-down box. The code to retrieve a list of projects from the repository is shown below.

Located in File *RequirementCommenterDialog.cs*, method *LoadProjectList()*.

```
//Get repository from our configuration object
Repository repository = Globals.RepositoryConfig.GetRepository();
cmbProjectList.DataSource = repository.GetProjects(); // works w/ data binding
```

The user selects one of the projects from the combo-box, and our next job is to read all **Requirements** from the selected project and add them to our class requirements list *m_requirements*. We will do this by stepping down the tree structure illustrated in figure {x}. We have already navigated down to the project level, so from there we will retrieve the folder holding all of the **Requirement** entities, and then query that folder for all of the entities it contains.

There are several ways of referencing entities in GENESYS. In this example we will start with the most direct method—referencing by name. The code required to reference all the entities from the **Requirement** class is below:

Located in File *RequirementCommenterDialog.cs*, method *Project_Changed()*.

```
// Here's how to get a project with a project name
// string projectName = cmbProjectList.SelectedItem.ToString();
// Repository repository = Globals.RepositoryConfig.GetRepository();
// IProject project = repository.GetProject(projectName);

IProject project = cmbProjectList.SelectedItem as IProject; // we're using databinding
if (project == null)
    return;
IFolder requirementsFolder = project.GetFolder("Requirement"); // Get requirements folder in that
project
// Load all requirements in this folder
m_requirements.Clear();
m_requirementIndex = 0;
foreach(IEntity req in requirementsFolder.GetEntities())
{
    m_requirements.Add(req);
}

DisplaySelectedRequirement();
```

READING A GENESYS ENTITY

We are now at the point in our story where we can start reading individual requirements from our user's project and displaying them to him. The next section shows how to read some of the commonly used information associated with GENESYS entities.

Individual requirements are entities of the **Requirement** class. Entities contain information in the form of attributes, properties, parameters, and relationships. The name attribute is somewhat special as it has its own accessor method as shown here from the *DisplaySelectedRequirements* method:

Located in File *RequirementCommenterDialog.cs*, method *DisplaySelectedRequirement()*.

```
IEntity req = m_requirements[m_requirementIndex]; // get selected requirement
lblRequirementName.Text = req.GetName(project); // gets full name
// Or just this if you don't care about cross-project extensions
// lblRequirementName.Text = req.Name;
```

Name may not be unique within a model class, depending on how the project was set up when it was created. It is recommended for displaying an entity's name to the user, but generally not for uniquely identifying an entity. The entity's ID is unique within the GENESYS model, and the ID is recommended as the way to uniquely identify an entity.

Other attributes have no special accessor and must be accessed differently. In the following example, we are retrieving the string value of the description attribute for the requirement as retrieved above:

Located in File *RequirementCommenterDialog.cs*, method *DisplaySelectedRequirement()*.

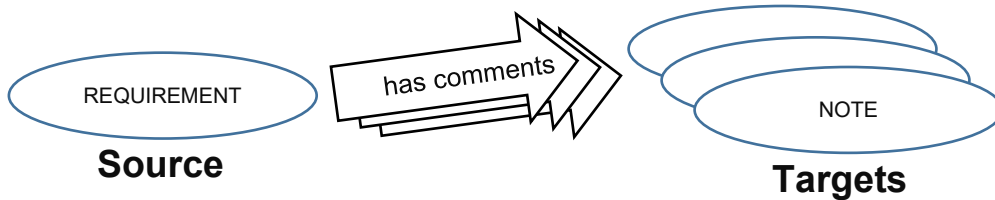
```
// Add description of the requirement -----
string description = req.GetAttributeValueString("description");
rtbDescription.Text = description;
```

It is important to draw a distinction here between **names** and **aliases** in GENESYS. Localizations and minor schema naming deviations between engineering disciplines are handled in GENESYS by giving the user the ability to create **aliases** for all classes and class datum **names**. Many times, there is no alias for a class or its associated data, so these are the same. When there is an alias, it will be displayed in the GENESYS client. If you try to use the alias to access the entity's data, you will get an error. The easiest way to see if an entity class or any of its associated data has been aliased is to view the schema in the GENESYS client.

Another way of navigating the GENESYS model is by following the relationships between entities. In the following example, we have the ability to get all **Notes** connected to a specific **Requirement** via the *has comments* relation.

Getting Started with GENESYS™ API

In the **GetRelationshipTargets** call below, we are requesting that **Requirement** entity **req** return all entities which are **targets** of the *has comments* relation, of which it is the **source**.



Located in File *RequirementCommenterDialog.cs*, method *DisplaySelectedRequirement()*.

```
// Add concerns to requirement -----  
// Specifically Requirements related to Concerns via the "has comments" relationship  
  
lstvNotes.Items.Clear();  
IEnumerable<IEntity> notes = req.GetRelationshipTargets("has comments");  
foreach(IEntity note in notes)  
{  
    ListViewItem item = lstvNotes.Items.Add( note.DisplayName );  
    item.Tag = note.Id; // store GUID of note for easy retrieval  
}
```

It is important to note that relationships have bidirectionality: **Relationships** consist of a pair of inverse **relations**. So, the *has comments/comments on relationship* consists of the *has comments relation* and its inverse, the *comments on relation*. The schema of a project determines which types of entities are valid sources and targets of each particular relation in the relationship. The *has comments* relation is in the base schema, and we know it has only one target entity type: **Note**. If there were more than one type of entity which could be a target of the *has comments* relation, we would need to remove those from the entity list returned by the **GetRelationshipTargets** call if we only wanted a list of **Notes**.


CREATING A GENESYS ENTITY

The key part of the user story is allowing our user to comment on project requirements. We do this in GENESYS by creating a **Note** entity and then linking it to the existing **Requirement** entity being commented on by creating the appropriate relationship.

GENESYS entities are created into the model as model class objects such as **Requirement**, **Component**, **Note**, **Function**, etc. Remember, folders hold particular model class types, so which model class is created depends on the folder in which it is created. Component entities are created in the Component folder or its subfolders, **Requirement** entities in the Requirement folder or its subfolders, etc. A name is required when creating a new entity; depending on how the project is set up in GENESYS, trying to create an object with a name that already exists may or may not fail due to name uniqueness rules. To be safe, assume that duplicate names will fail. When creating a new entity, the GENESYS API provides a facility to retrieve a guaranteed unique name from a particular class.

Getting Started with GENESYS™ API


Located in File *RequirementCommenterDialog.cs*, method *buttonNoteAdd_Click()*.



```
//Get repository from our configuration object
IFolder notesFolder = project.GetFolder("Note"); // Get Notes folder in that project
String defaultName = project.GetDefaultEntityName( notesFolder.EntityDefinitionId );
IEntity newNote = null;
try
{
    newNote = project.CreateEntity(notesFolder, defaultName); // Create new note w/ default name
}
catch(FaultException dupFault)
{
    System.Windows.MessageBox.Show( dupFault.Message, "Create Entity", System.Windows.MessageBoxButton.OK,
        System.Windows.MessageBoxImage.Error);
    return;
}
```

GENESYS entities are usually connected to other entities in the model via relationships. In our case, the requirement *has comments* of a new **Note**, so it is important to reflect that in the model. To do that we will create a new relationship using the *has comments* relation in the project between the **Requirement** and the **Note**. The following code continues the segment above and creates the new relationship by providing the *CreateRelationship(...)* call with the "has comments" relation, the source entity (the user-selected requirement); and the target entity we just created:

Located in File *RequirementCommenterDialog.cs*, method *buttonNoteAdd_Click()*.



```
// Connect note to our requirement using the generates Relationship
IEntity req = m_requirements[m_requirementIndex]; // get selected requirement
Try
{
    project.CreateRelationship( project.Schema.GetRelationDefinition("has comments"), req, newNote);
}
catch(FaultException fault)
{
    System.Windows.MessageBox.Show(fault.Message, "Create Relation", System.Windows.MessageBoxButton.OK,
        System.Windows.MessageBoxImage.Error);
    return;
}
```


Notice that in the *CreateRelationship(...)* call above we could have used a *relationDefinitionConstant* instead of calling *project.Schema.GetRelationDefinition(...)* to retrieve a relationship definition. Use the constant when referring to a relation that is part of the GENESYS-defined schema. You **must** use the *project.Schema.GetRelationDefinition(...)* call when retrieving a custom relation definition that is not part of the predefined base schema.

When working in GENESYS, it is important to note the difference between a **relation** and a **relationship**. A **relationship** is made of a pair of inverse **relations**. When you create the relationship above using the **has comments** relation, the inverse relation, **comments on**, is also created in the database. So, the **Requirement has comments** of a **NOTE** at the same time the **Note comments on** the **Requirement**. This is important when calling *GetTarget()* or *GetSource()* on a **relationship** to retrieve the entity at the other end.

MODIFYING A GENESYS ENTITY

Our user is almost done with adding his comment to the project model. The final step is adding the user's comment text to the comment object we created above. When the **Note** entity was created, all its attributes, parameters, and properties were set to their default values. The following code shows how to set a named attribute of an entity:


Located in File *NoteDialog.cs*, method *SetEntityAttributeStr()*.



```
private void SetEntityAttributeStr( IEntity entity, String Attr, String attrVal )
{
    IEntityAttributeValue av = entity.Attributes[Attr];
    DataTypeDefinition typeDef = av.AttributeDefinition.DataType;
    object outVal;
    if (typeDef.TryConvertValue(attrVal, out outVal))
        av.SetValue(outVal);
    else
        MessageBox.Show("Attribute conversion failed.");
}
```

Also important to note here is that the *Update()* method must be called to write any changes made to an entity to the model as done here when the user clicks OK:

Located in File *NoteDialog.cs*, method *OkBtn_Click()*.



```
private void OkBtn_Click(object sender, EventArgs e)
{
    try
    {
        m_Note.Update(); // force write to db
    }
    catch (FaultException<InsufficientPrivilegesFault> fault)
    {
        System.Windows.MessageBox.Show(fault.Message, "Set Entity Attribute",
            System.Windows.MessageBoxButton.OK, System.Windows.MessageBoxImage.Error);
        return;
    }

    this.Close();
}
```

While our example does not allow the user to delete entities from the model, this can be accomplished by calling the *Delete()* method on an entity for which you have delete privileges.

LOGGING OUT OF GENESYS

Remember to log out of the GENESYS repository before switching repositories or when your code exits. The following code shows how to gracefully log out from the GENESYS repository.

Located in File *NoteDialog.cs*, method *OkBtn_Click()*.

```
try
{
    if(Globals.RepositoryConfig != null )
    {
        if(Globals.RepositoryConfig.Status == Vitech.Genesys.Common.AvailabilityStatus.Available)
        {
            Globals.RepositoryConfig.Logout();
        }
        Globals.RepositoryConfig = null;
    }
    Globals.Client = null;
}
catch(Exception)
{
    MessageBox.Show("Failed to logout of the GENESYS client");
}
```

When exiting GENESYS, it is important to use the *Dispose()* method so that licenses, sessions, and other resources can be reused as soon as possible.

DEPLOYMENT

Finally, we come to the last part needed to complete our user story—deploying the new API application to our user.

The first thing needed to install a GENESYS API application is the licensing runtime service. If the machine has a GENESYS product installed on it, it will already have this installed. If not, you will need to install it. Again, the easiest way of doing this is by installing the latest GENESYS edition. If this is not possible, the runtime service can be downloaded from the Vitech website here: https://vitechcorp-webdownload.s3.amazonaws.com/support/Sentinel_LDK_Run-time_setup_823.zip.

You will also need a valid GENESYS license. If the machine you are installing on is connected to other machines with a valid license, you will be able to use one of these. If your machine is not so connected, you will have to install a license locally. You will need the GENESYS *License Manager* to switch between local and remote licensing modes.

If the licensing service is correctly installed, you will be able to browse to <http://localhost:1947> on the machine and see a list of all licenses available.

The simplest way to deploy is to include the executable and all the linked .NET DLL files in one folder. Copy the entire folder on the user's machine and provide a shortcut to the executable.

WHERE TO FIND FURTHER HELP

The additional samples in the Samples\API Samples directory demonstrate a WPF style example and a web service example and may be a better starting point for your particular application.

If you are a visual learner, you might try Vitech's API webinars series which can be found on Vitech's website under Support (<https://www.vitechcorp.com/webinar-videos-on-demand/>).

APPENDIX A: PREREQUISITES, ASSEMBLIES, AND CONFIGURATION

.NET Framework Prerequisites

GENESYS is built via Microsoft's .NET 4.8 framework and API applications must utilize the .NET 4.8 or greater runtime to directly reference and use GENESYS API assemblies. For applications using .NET runtimes that are versioned prior to 4.8, service-based approaches can be used.

Sample API projects are available in your GENESYS install directory's Sample folder to assist in understanding key API methods, properties, and techniques. The GENESYS samples provide Microsoft Visual Studio® 2017 solution and project files.

Key GENESYS API Assemblies

Several assemblies are available for reference, offering comprehensive public interfaces to project model data. For full descriptions of public assembly methods, properties, and other functionality, please visit the API reference in the GENESYS online help files.

Vitech.Genesys

Required for API application development. The Vitech.Genesys assembly provides GENESYS product and version data. While it may or may not be used explicitly, other API assemblies require it.

Vitech.Genesys.Client

Required for API application development. The Vitech.Genesys.Client assembly is the workhorse for working with repository data. It provides access to key data structures such as repositories, projects, folders, entities, attributes, and relationships, as well as to schema meta-data such as entity definitions, relation definitions, and attribute definitions. It also contains repository connection and authentication functionality.

Vitech.Genesys.Common

Required for API application development. The Vitech.Genesys.Common assembly contains key enumerations and constants that are used throughout the application, such as entity definition identifiers and type descriptors. It also contains definitions used to convert values to GENESYS types, as well as exception classes specific to GENESYS actions. While it may or may not be used explicitly, other API assemblies require it.

Vitech.Genesys.Logging

Required for API application development. The Vitech.Genesys.Logging assembly contains the ability to read warnings and errors from other Vitech assemblies. Logging levels are set in the App.config (see below). While it most likely would not be used explicitly, other API assemblies require it.

Vitech.Genesys.Random

Required for API application development. The Vitech.Genesys.Common assembly hosts algorithms and other functionality used across the GENESYS framework. While it most likely would not be used explicitly, other API assemblies require it.

Licensing Assemblies

The GENESYS licensing assemblies would typically not be used directly for API application development efforts. However, they provide licensing services and validation for the GENESYS framework and are required references for an API application.

Getting Started with GENESYS™ API

Note that while GENESYS API assemblies are compiled for “Any CPU,” some licensing functions require independently compiled 32-bit and 64-bit versions. Including both versions in an application will ensure it works seamlessly in either environment.

- 1) *Vitech.Genesys.License*
- 2) *Vitech.Genesys.License.Provider*
- 3) *hasp_net_windows*
- 4) *hasp_windows_82194*
hasp_windows_x64_82194
- 5) *apidsp_windows*
apidsp_windows_x64

Note: The Sentinel HASP DLL files (#3 – 5) must each be output as a file with your API application. They should not be directly referenced. Instead, they should each simply be added to a .NET GENESYS API project as existing files. Each .dll file’s “Build Action” property should be set to “Content” and its “Copy to Output Directory” property should be set to “Copy if newer.”

App.config

An API application’s app.config file should contain the following nodes in the appSettings section.

- The LoggingLogLevel value provides different levels of transaction and error reporting. Possible logging levels are: off, error (default value), info, and verbose.
- The ProductCode value provides information to the GENESYS services that is required to authenticate a user’s license against the GENESYS version in use.

```
<appSettings>
  <add key="LoggingLogLevel" value="Off"/>
  <add key="ProductCode" value="G03"/>
</appSettings>
```

Upgrading to a new version

The version of the GENESYS DLL files included in the API project should match the GENESYS installation where the API is being used. Therefore, when upgrading your GENESYS installation, remember to copy over the following files into your solution:

- apidsp_windows.dll
- apidsp_windows_x64.dll
- hasp_net_windows.dll
- hasp_windows_82194.dll
- hasp_windows_x64_82194.dll
- Vitech.Genesys.Client.dll
- Vitech.Genesys.Common.dll
- Vitech.Genesys.dll
- Vitech.Genesys.License.dll
- Vitech.Genesys.License.Provider.dll
- Vitech.Genesys.Logging.dll
- Vitech.Genesys.Random.dll

THIS PAGE INTENTIONALLY BLANK



2270 Kraft Drive, Suite 1600
Blacksburg, Virginia 24060
+1 540 951 3322 | Fax: +1 540 951 8222
www.vitechcorp.com

Customer Support:
+1 540 951 3999 | support@vitechcorp.com