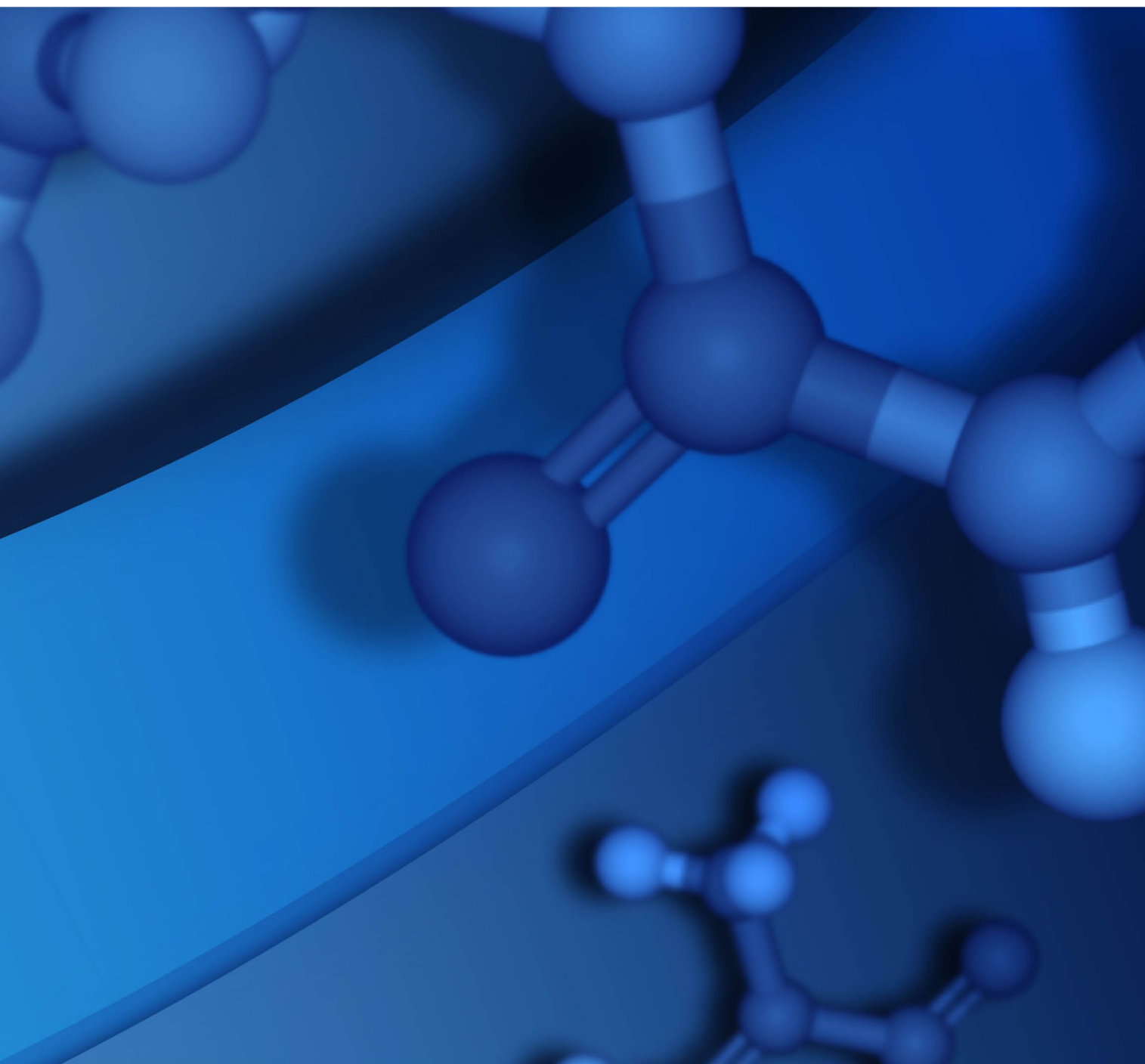


SDK DEVELOPERS GUIDE

BIOVIA WORKBOOK 2021



Copyright Notice

©2020 Dassault Systèmes. All rights reserved. 3DEXPERIENCE, the Compass icon and the 3DS logo, CATIA, SOLIDWORKS, ENOVIA, DELMIA, SIMULIA, GEOVIA, EXALEAD, 3DVIA, 3DSWYM, BIOVIA, NETVIBES, IFWE and 3DEXCITE, are commercial trademarks or registered trademarks of Dassault Systèmes, a French "société européenne" (Versailles Commercial Register # B 322 306 440), or its subsidiaries in the U.S. and/or other countries. All other trademarks are owned by their respective owners. Use of any Dassault Systèmes or its subsidiaries trademarks is subject to their express written approval.

Acknowledgments and References

To print photographs or files of computational results (figures and/or data) obtained by using Dassault Systèmes software, acknowledge the source in an appropriate format. For example:

"Computational results were obtained by using Dassault Systèmes BIOVIA software programs. BIOVIA Workbook was used to perform the calculations and to generate the graphical results."

Dassault Systèmes may grant permission to republish or reprint its copyrighted materials. Requests should be submitted to Dassault Systèmes Customer Support, either by visiting <https://www.3ds.com/support/> and clicking **Call us** or **Submit a request**, or by writing to:

Dassault Systèmes Customer Support
10, Rue Marcel Dassault
78140 Vélizy-Villacoublay
FRANCE

Contents

Chapter 1: BIOVIA Workbook SDK	1
BIOVIA Workbook and Vault Server Architecture	2
Framework Applications Overview	2
Symyx Framework Class Libraries	3
Chapter 2: BIOVIA Vault Server	5
Accessing Vault	5
VaultWorkspace Class	6
VaultServer Class	6
Connecting to a Vault Server Endpoint	7
Verifying that Vault Services are Running	7
Authentication and Logging in to Vault	7
Security Permissions	8
Log in to Vault Example	9
VaultRepository Class	11
User Repository	11
Folders	11
Retrieve Vault Users List Example	12
Vault Objects	13
Vault Object Base Classes	14
Vault Object Flags Property	15
Vault Object Core Properties	15
Set Properties in VaultObject.ExtendedProperties	25
Use Data Scope to Set Information	25
Retrieve Vault Users Example	27
Retrieve a Vault Object using Vault ID or Vault URI	28
Create a Batch For Identity Requests	28
Retrieve and Cast a Vault Object	29
Identity Requests Scope	29
Retrieve Vault Objects by Object Type	29
Search for a Vault User Example	30
Retrieve Vault Assemblies	30
Use GetMembers For Context Retrieval	30
Retrieve Vault Folders	31

Retrieve Hierarchy of Vault Folders Example	33
Use Vault Query Service to Get Vault IDs	33
Convert Vault IDs	35
Vault Object References and Associations	36
Vault URIs	37
Scheme	37
Authority	37
Path	38
Query	39
Fragment	39
Vault Object Annotations	39
Add and Remove an Annotation	40
Create and Delete Vault Objects	41
Package Vault Objects in VOZIP Files	41
Create a Vault Object Package Example	42
Add an Object to a Vault Object Package	42
List Objects in a Vault Object Package	43
Publish Using a VOZIP File	43
Delete an Object From a Vault Object Package	44
Read and Write to a Vault Object	45
Display DLL Assembly Dependencies	45
Chapter 3: Users and Security	48
Classes Supporting Users Settings and Preferences	48
Access User Profiles	48
Permissions	48
Explicit Permissions	49
Implicit Permissions	50
Application Permissions	50
Chapter 4: Properties	51
Property Set Editor	51
Property Set Definitions	51
Property Event Handlers	52
Property and Vault Object Variables	53
Property Class and PropertySetDefinition Variables	53
Variable Aliases	53

Validation Script Variables for Value Changing Handlers	53
CalculateValueHandler Script Variables	54
ValueSelectionsProvider Script Variables	55
Property Dictionaries	55
Dictionary Providers Types	55
Dictionary Scripts Examples	56
ValueSelectionsProvider Script Variables	57
Chapter 5: Materials	59
Material Classes and Interfaces	59
Material Class	59
Material Properties	59
Material as a Mixture	60
DensityCalculation Class	61
Materials Calculations	62
Conversions	63
Calculating Molecular Weight Example	64
Materials Sections C# Example	65
Measurement Class	66
Validate a Measurement	66
Convert Amounts	67
Container Class	69
Preparation Class	69
Chapter 6: Documents	71
Find a Template Example	72
Create a Document from a Template	72
Get a Document	73
Adding, Inserting, and Removing Document Sections	73
Insert a Section Between Other Sections	74
Remove a Section	74
Text Sections Example	74
Add Text Sections and Set Plain Text	75
Add Data to a Text Section	76
Check the Section Type	78
Scale an Image	78
Chapter 7: Query Service for Searching	79

RAS Data Schema	80
Query Form Data	82
Search Results	83
Create a Custom Query Builder Using Metadata	83
Custom Vault Objects Indexing	84
Full-text Search Indexing	84
IIndexableText Implementation Example	85
Custom Indexing	85
Creating a New Search Type	85
SampleIDSearchExtension	86
Build Queries	86
Search Extension Development Best Practices	89
Search Extension Configuration	90
Chapter 8: Scripting in BIOVIA Workbook	92
Workbook Objects	92
Python Scripting	92
Script Performance Profile	93
Document Toolbar Scripting	93
Optimize Scripts	93
Form Editor Scripting	94
FormSection Events	94
FormSection Script Variables	95
FormSection Events Script Variables	96
Access Widgets	98
OnReview Script Example	99
OnValidate Script Example	100
Add Scripts to an Experiment Template	101
Experiment Editor Events	101
Experiment Editor Event Scripts	104
Experiment Editor Events Script Variables	105
Get the Active Section	106
Access Menu Items	106
Menu Item Property Changes	106
Menu Item Names	107
Access Workbook Toolbar Items	110

Section Toolbars and Toolbar Items	110
Access Workbook Toolstrips	113
Check User Permissions and Disable a Section Example	113
Remove the Active Section	114
Rename the Active Section	114
Add a Button to a Toolstrip	115
Custom Toolbar Scripting	115
Custom ToolStripButton Example	115
Assign a Script to a Toolbar Button	116
Interaction Between Scripts	117
Insert an Excel File	117
Add a Section to an Experiment	118
Error Handling in Scripts	119
Cancel an Action	119
Raising an Exception	120
sys.exit	120
Generate Unique IDs	120
Sequence Name for Unique IDs	120
ID Formatting	121
Generate SampleID Example	121
List Variables in Scope	122
Add a Dictionary to a Recipe Section	122
Content History for a Control	122
Form Control Content History	123
ELN Assembly Cache	124
Release Memory	126
Omit Vault Object Content Compression	127
Prevent Concurrent Updates	127
Debug the Framework	127
Debug a Remote Service	127
WCF Tracing for Vault Diagnostics	127
Script From External Assemblies	128
Use an External .NET Assembly	128
CreateInstanceFromLatestAssembly Method	129
Create Custom .NET Assembly	129

Create a .NET project	130
Sign Your Assembly	130
Writing Classes and Methods	130
Naming Conventions	130
Adding references to Workbook assemblies	131
Define a Class	131
Define a Method	132
Chapter 9: Sections	133
Clone an Experiment to the Latest Template Version	135
Sections in a New Document	135
Forms and Tables	136
Insert Forms	136
Import Forms	136
Populate Form Controls	137
Populate a List Using Vault Vocabulary	138
Form Examples	138
Populate Widgets in Forms	139
References	140
Property Set Definitions	140
Clone to Latest Limitations	141
File Sections	141
Add and Remove Files	142
Visualizations	143
Required Software for Visualizations Utility	143
Create File Section with Table Rows	144
List the Property Set Definitions for a Table	144
List Values from a Table	145
Invoke a Form and Add Rows	145
Set Values in a Table	146
Add a Property Set Definition	146
Insert Rows	147
TableSection Script Variables	147
Script with Table Section Properties	148
Access a Table and its Rows	149
Import and Export Data	149

Import Summary Data	150
ImportExportData to Update Data	151
Lock Imported Rows	152
Prevent the Removal of Locked Rows	152
Export or Import All Table Rows	152
Request Column Dictionary Event	153
Table Section Script Events	153
Table Section Event Variables	155
Table Section Script Examples	156
Material Section Script Variables	158
Material Property Set Definitions	159
Nullable for Primitive Types	162
Material Section Script Examples	162
Access the Material Structure	163
Create a Review Message	163
Scripting Material Import	164
BeforeImportMaterials Event Script Example	164
AfterImportMaterials Event Script Example	165
Testing Examples	165
Script Variables for Experiment and Common Section Events	165
Script Variables for Events	166
Workbook Sections	168
Materials Section Event Script Variables	169
Sample Preparation Section Script Variables	170
Export Preparation Section Data	174
Data Exported as CSV File	179
Change the Scale Used In Calculations	179
Script for Custom Scale	180
Change the Scale of Calculated Values for a Formulation	181
Unit Types	181
DataCreation Example	182
CreateDocument Method	183
Reaction Scheme C# Examples	183
Locate the Reaction Scheme Section	184
Locate a Reaction Step	185

Add a Reaction Step	185
Add a Reaction From a File	185
Link Corresponding Materials Section	186
Add a Material Using AddRow	186
Modify a Material	186
Chapter 10: Build and Debug a Custom .NET Assembly	187
IronPython Script For Calling a Custom Assembly Example	187
C# Code for Importing Custom Data	190
Publish a Custom .NET Assembly	191
Call an External Assembly with IronPython	191
AssemblyCache.Publish Method	192
List Assemblies in Vault	194
Publishing Referenced .NET Assemblies	194
Publish a New Version of a .NET Assembly	195
Unpublish an Assembly	195
List of Assemblies in Vault	195
In Visual Basic .NET	195
In C#	195
Chapter 11: Workflow Designer	196
Vault objects	196
Custom Workflow Activities	196
Prerequisites	196
Create a Custom Workflow Activity	197
Configure the Build Location for Your DLL	198
Custom Activity OnExecute Method	198
Configure Workflow Designer to Use a Custom Activity	200
Add a Custom Activity to a Workflow	200
Custom Workflow Activity Example	201
Compile and Publish a Workflow	202
Change the Vault Logging Level	202
Appendix A: Potentially Breaking API Changes in Workbook 2018	204
Namespace Replacements	206
Changes to Menu Item List Creation	206
GridView Methods Require Additional Parameters	206
Deprecated Events and Methods	207

EditorContainer.RepositoryItems Renamed as ExternalRepository	207
RepositoryItemCheckedComboBoxEdit.ShowAllItemCaption Renamed as SelectAllItemCaption	207
tree.OptionsBehaviour.DragNodes Renamed and Changed to Boolean	207
Exceptions Sometimes Thrown when Operating on non-UI Thread	207
Some Grid Methods Now Clear Status Data	207

Chapter 1:

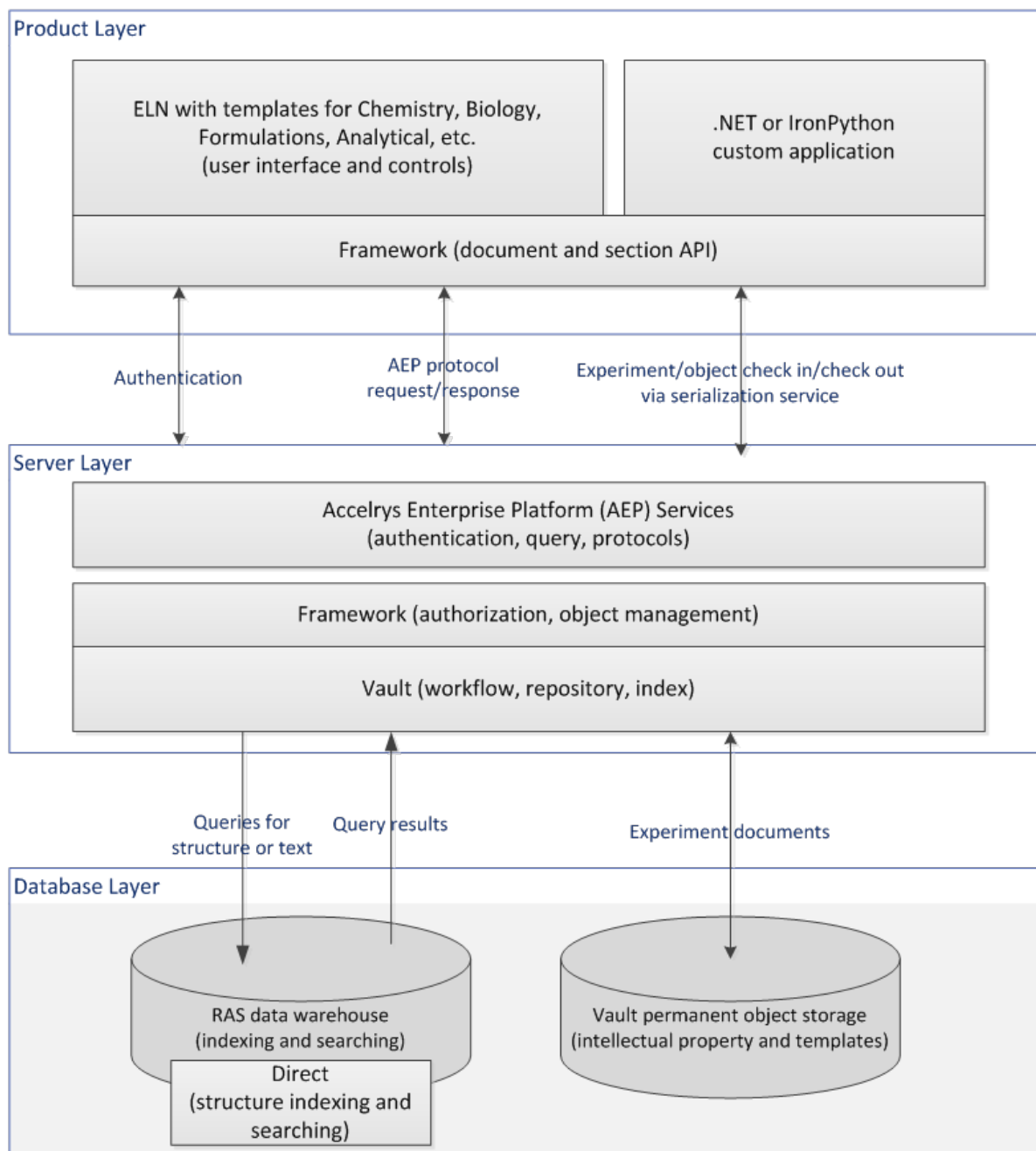
BIOVIA Workbook SDK

The Workbook SDK is a set of class libraries ("Framework") for creating applications that extend Workbook capabilities and integrate Workbook with other applications. Software developers can use the Framework to create applications that access or capitalize on the following Workbook services and features:

- Object persistence, versioning, and history
- Data warehousing and search
- Workflow management
- Security and signing
- Object properties and content
- Common dialogs and controls
- Reports
- Base objects like materials, units and conversions, and quantities

The Framework can be used to create client and server applications. Several of its namespaces facilitate client-side interaction with BIOVIA Vault Server.

BIOVIA Workbook and Vault Server Architecture



Framework Applications Overview

To develop a Framework application using Visual Studio, you should create a project to implement your customizations. The type of project depends on what you are building. You can:

- Add references to the appropriate Framework assemblies, that are located in the lib folder of the SDK installation.

- Add a reference to `Symyx.Framework.dll`; this has the component name of Symyx Framework: Core.
- Add references to the Microsoft's assemblies, `System` and `System.Core`.

If you need to use the Windows Forms controls, add references to the following assemblies:

- `Symyx.Framework.Controls.dll`
- `Symyx.Windows.dll`

If you need to use the Materials object model, add references to the following assemblies:

- `Symyx.Framework.Materials.dll`
- `Symyx.Framework.Quantity.dll`

In your application code, define your class to implement the required interfaces and extend the classes in the Framework API.

Symyx Framework Class Libraries

The table shows the class libraries that are included when you install Symyx Framework. The class libraries are located in the Symyx Framework lib folder.

Assembly	Namespaces
<code>Symyx.Framework.dll</code>	<code>Symyx.Framework</code> <code>Symyx.Framework.ApplicationManagement</code> <code>Symyx.Framework.Cache</code> <code>Symyx.Framework.Chemistry</code> <code>Symyx.Framework.Collections</code> <code>Symyx.Framework.Extensibility</code> <code>Symyx.Framework.History</code> <code>Symyx.Framework.IO</code> <code>Symyx.Framework.Logging</code> <code>Symyx.Framework.Properties</code> <code>Symyx.Framework.Properties.ImportExport</code> <code>Symyx.Framework.RAS</code> <code>Symyx.Framework.Scripting</code> <code>Symyx.Framework.TabularData</code> <code>Symyx.Framework.User</code> <code>Symyx.Framework.Vault</code> <code>Symyx.Framework.Vault.Exceptions</code> <code>Symyx.Framework.Vault.Packaging</code> <code>Symyx.Framework.Vault.Security</code> <code>Symyx.Framework.Vault.Signing</code> <code>Symyx.Framework.Workflow</code>
<code>Symyx.Framework.Controls.dll</code>	<code>Symyx.Framework.Controls</code> <code>Symyx.Framework.Controls.Dialogs</code> <code>Symyx.Framework.Controls.Events</code>
<code>Symyx.Framework.Quantity.dll</code>	<code>Symyx.Framework.Quantity, Unit, Value</code>

Assembly	Namespaces
Symyx.Framework.Reporting.dll	Symyx.Framework.Reporting
Symyx.Framework.Materials.dll	Symyx.Framework.Materials
Symyx.Windows.dll	Symyx.Windows Symyx.Windows.ApplicationManagement Symyx.Windows.ToolStripManagement Symyx.Windows.ToolWindowManagement

The Symyx Framework lib folder also contains third-party class libraries that are used and bundled with Symyx Framework and Workbook.

Chapter 2:

BIOVIA Vault Server

BIOVIA Vault Server is an object management system that supports BIOVIA Workbook. Vault Server provides:

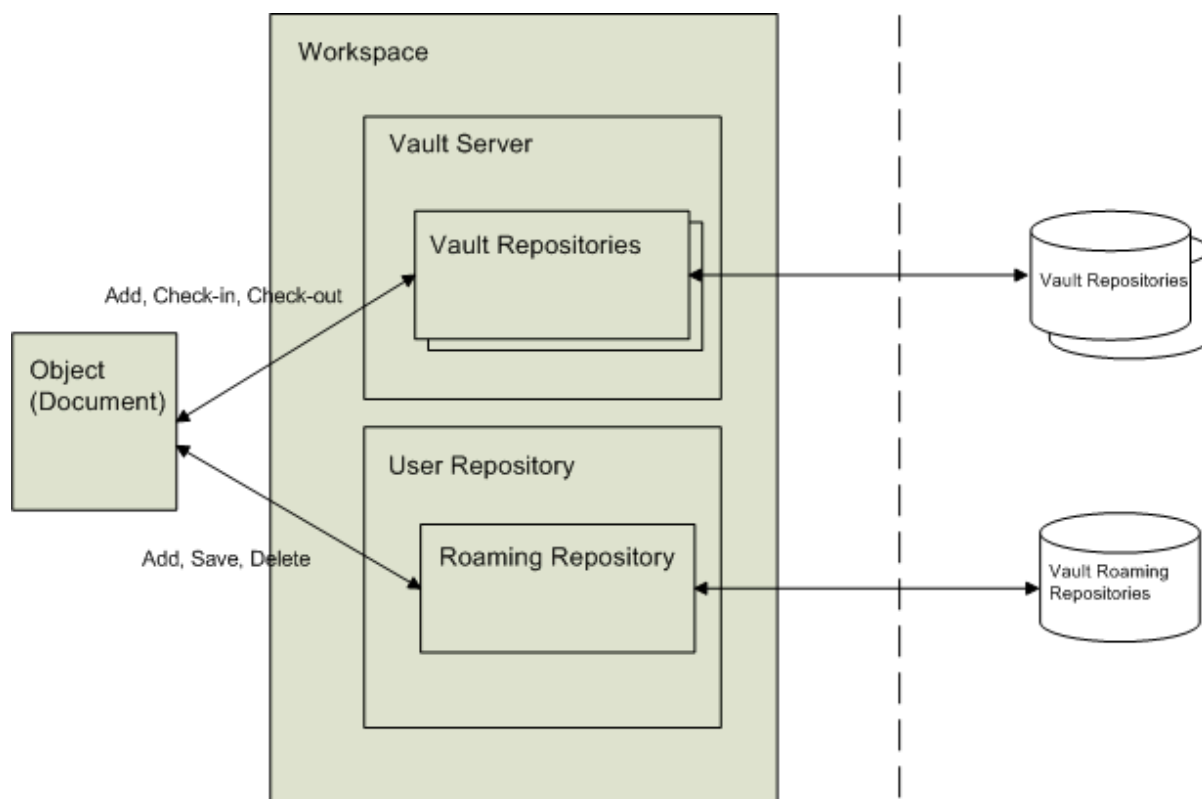
- A searchable database of research and development data.
- Integrated lab device data acquisition with Workbook data.
- Analytical capabilities through on-demand data warehousing in batch or real-time.
- Workflow capabilities for streamlining the document approval process.
- Enterprise level security and auditing.
- Versioned object data management.

Vault uses a site repository to store the common data among the versioned repositories. The common data includes information about the system security, federation tables, the Workbook assembly cache, and directory service.

Vault supports user repositories that store Vault objects relevant to an application, user, or group. Each repository exposes endpoints that clients use to send requests to the repository. Multiple repositories can share endpoints.

Accessing Vault

You use the classes and interfaces in the `Symyx.Framework.Vault` namespace to access Vault. The `Symyx.Framework.Vault` namespace provides client-side classes for interacting with Vault. The following diagram shows the relationships between the classes in the Vault API object model:



VaultWorkspace Class

A workspace is the top-level abstraction through which client programs access Vault. The workspace allows clients to log in to a Vault server to perform data persistence and retrieval operations.

The `Symyx.Framework.Vault.VaultWorkspace` class contains methods that:

- Authenticate and log in a user to Vault
For examples of how to log in to a Vault workspace, see [Authentication and Logging into Vault](#).
- Manage objects or object collections in Vault such as methods that get, add, delete, move, update, and get the history of Vault objects.

VaultServer Class

A Vault server is the middle-tier engine that services client requests to access the Vault system. A Vault server can access multiple repositories. The `Symyx.Framework.Vault.VaultServer` class contains properties and methods that:

- Log in and authenticate a user.
- Set permissions, workflow security, and the signature on Vault objects.
- Initialize and access a file system repository, local storage folder, or roaming repository for the user.
- Manage objects or object collections in Vault, such as methods that get, add, delete, move, update, and get the history of Vault objects.

Connecting to a Vault Server Endpoint

Vault uses the Windows Communication Foundation (WCF) service to enable a client application to connect to the Vault server endpoint. An endpoint is where messages are sent or received between clients and servers. An endpoint defines all of the information required for the message exchange, and consists of the:

- Location where messages are sent to; the URI that represents the address of the service.
- Binding that specifies the message communication protocol such as `BasicHttpBinding`, `WsHttpBinding`, `NetNamedPipeBinding`, `NetMsmqBinding`.
- Specification for the messages that can send the service contract.

To connect to a Vault server endpoint, you specify the endpoint when you invoke a `VaultServer` or `VaultWorkspace` constructor. If the specified endpoint does not have WCF binding information in the .NET configuration file, the Symyx Framework creates a default endpoint for the caller. The Vault server uses basic HTTP binding, which means that the only thing that the caller needs to do is to pass the endpoint into the workspace when invoking the `VaultServer` or `VaultWorkspace` constructor.

Verifying that Vault Services are Running

To ensure the Vault services are running:

1. Log in to the computer on which Vault is running as a Windows administrator.
2. Choose **Start > All Programs > Administrative Tools > Services**.
3. Click **Services**.
4. Verify that the following services are running:
 - Vault Message Processing Service
 - Workflow Service

Authentication and Logging in to Vault

The Symyx Framework supports the following types of authentication:

- The `Symyx.Framework.Vault.UsernameCredentials` class represents the username/password credentials.
- Extensible binary authentication such as Public Key Infrastructure (PKI) and biometrics. To implement binary authentication, use or derive from the `System.Framework.Vault.BinaryCredentials` class.

A user can log into Vault through a workspace. The `Symyx.Framework.Vault.VaultWorkspace` class provides `Login` methods enable the user to log in to Vault.

The following C# example shows the simplest way to log in:

```
// create a vault workspace
VaultWorkspace workspace = new VaultWorkspace("myVaultServer");

// attempt to log in workspace.Login(@"myDomain\myUser", "myPassword");
Console.WriteLine("Login to server {0}", workspace.IsAuthenticated ?
"Succeeded": "Failed");
```

In the code example, the `myVaultServer` parameter is the same Vault server name for which the SSL certificate is issued. In the call to the `Login` method, the username is combined with the domain to form `myDomain\myUser`.

To connect to the server, you do not need to set up Windows Communication Foundation (WCF) configuration. You can use configuration information in the current `app.config` or `web.config` file, however, the Symyx Framework uses a set of default bindings.

Most Workbook users have one Workspace that is encapsulated in the `VaultWorkspace.Current` property. If a workspace is instantiated and there is no current workspace, it will assign itself as the Current workspace.

You also can log in using an object of one the following classes that derive from `Symyx.Framework.Vault.SecurityCredentials`:

- `UsernameCredentials` represents username and password credentials.
- `BinaryCredentials` represents binary security credentials to encapsulate generic user tokens such as PKI.

The following C# example shows the `UsernameCredentials` object:

```
// create a vault workspace
VaultWorkspace workspace = new VaultWorkspace("myVaultServer");

// attempt to log in to the workspace
workspace.Login(new UsernameCredentials
                ("myDomain", "myUser", "myPassword"));
Console.WriteLine("Login {0}", VaultWorkspace.Current.IsAuthenticated ?
                  "Succeeded" : "Failed");
```

Security Permissions

The `Symyx.Framework.Vault.Security` namespace contains classes that encapsulate the permissions that users and groups have on Vault objects. The `Permissions` property of the `Symyx.Framework.Vault.VaultObject` class returns the `Symyx.Framework.Vault.Security.ObjectPermissions` for that Vault object. The permissions enumeration contains the different types of permissions for low-level operations that you can grant to a Vault object. Multiple low-level permissions are aggregated into higher level privileges that are encapsulated in the `Privilege` class.

The following C# example returns false if a Vault object does not have the specified permissions:

```
if (vaultObject.Permissions != null)
{
    // Must have the checkout permission.
    if (!vaultObject.Permissions.HasPermission
        (Permissions.Checkout))
    { return false; }
    // Must have the write data permission.
    if (!vaultObject.Permissions.HasPermission
        (Permissions.WriteData))
    { return false; }
    // Must have the write properties permission.
    if (!vaultObject.Permissions.HasPermission
        (Permissions.UpdateProperties))
    { return false; }
    // Must have the transition permission.
```

```

    if (!vaultObject.Permissions.HasPermission
        (Permissions.WorkflowTransition))
    { return false; }
}

```

Log in to Vault Example

This section shows a complete C# example program that you can compile and run. The program demonstrates how to log into a Vault server.

// Do not run this example in a production environment.

```

using System;
using Symyx.Framework.Vault;
namespace Symyx.SDK.Framework.Examples
{
    class LogInToVault
    {
        static void Main(string[] args)
        {
            // declare strings to store the vault server name, domain, username, and
            password
            string endpoint; string
                domain; string user;
                string password;
            if (args.Length == 4)
            {
                // if the vault server name, domain, username, and password are
                entered on the command line,
                // use those entries
                endpoint = args[0];
                domain = args[1];
                user = args[2];
                password = args[3];
            }
            else
            {
                // otherwise, prompt the user to enter the vault server name, domain,
                username, and password
                endpoint = GetConsoleValue("Server");
                domain = GetConsoleValue("Domain");
                user = GetConsoleValue("User");
                password = GetConsoleValue("Password");
            }
            // attempt to log in to the vault server
            Console.WriteLine("Attempting to log in to vault server {0}...",
                             endpoint);
            VaultWorkspace workspace = new VaultWorkspace(endpoint);
            workspace.Login(new DomainUsernameCredentials(domain, user, password));
            // check the user workspace is authenticated if
            (workspace.IsAuthenticated)
            {
                Console.WriteLine(string.Format("Logged in to the vault server"));
                Console.WriteLine(workspace.IsOnline ? "Server is online" : "Running

```

```
in offline mode");
    Console.WriteLine("Hello user " + workspace.CurrentUser);
}
else
{
    Console.WriteLine(string.Format("Could not log in to the vault
server"));
}
}
// prompt the user to enter a string
private static string GetConsoleValue(string name)
{
    Console.Write("{0}: ", name); return Console.ReadLine();
}
}
```

Before compiling, add a reference to Symyx Framework, Core, the `Symyx.Framework.dll`.

To run the example use the following statement:

```
LogIntoVault myVaultServer myDomain myUser myPassword
```

Log in to Workspace Example

The following C# example shows a more complex example of how to log in and add the Vault server to the workspace. The example checks the endpoint string and exits the loop if the endpoint is blank. The example shows the use of the `VaultWorkspace CreateServer` method, which creates a server endpoint object:

```
// create a vault workspace
VaultWorkspace workspace = new VaultWorkspace();

// while the workspace does not have an active server set...
while (!workspace.HasActiveServer)
{

    // prompt the user to enter a vault server name string
    endpoint = GetConsoleValue("Server");

    // if the string is null or empty, break from the loop
    if (string.IsNullOrEmpty(endpoint)) { break; }

    // create a server object
    IServer server = workspace.CreateServer(endpoint);

    // prompt the user to enter the domain string
    domain = GetConsoleValue("Domain");

    // prompt the user to enter the vault username and password string
    user = GetConsoleValue("User");
    string password = GetConsoleValue("Password");

    // attempt to log in to vault
    Console.WriteLine("Attempting login to server {0}...", endpoint);
```

```

server.Login(new DomainUsernameCredentials(domain, user, password));

// if authenticated, add the server to the workspace
if(server.IsAuthenticated)
    {workspace.Add(server);}
}

//Assume the example GetConsoleValue method prompts the user to enter a
string:
// prompt the user to enter a string
private static string GetConsoleValue(string name)
{
    Console.Write("{0}:", name); return Console.ReadLine();
}
Repositories

```

The Symyx Framework provides two types of repositories for documents and objects:

- Vault repository
- User repository

VaultRepository Class

The Vault repository is a centralized repository managed by the Vault server. The `Symyx.Framework.Vault.VaultRepository` class provides the abstraction for the repository.

There are several types of repositories managed by Vault. You can determine the type of repository by getting the `VaultRepository.RepositoryType.Name` property. The `RepositoryBehaviors` property describes the behavior of each repository.

Vault provides versioning capabilities and provides control over object updates. The `VaultRepository` class provides methods that enable you to add, modify, copy, move, delete, and get objects from the repository. You can also soft-delete an object to mark an object as deleted but is not removed from the repository.

User Repository

The user repository contains user-specific data and documents. This is similar to the `C:\Documents` and `Settings\Username` folder in Microsoft Windows.

User profiles are similar to the contents of `C:\Documents` and `Settings\Username\Application Data`. User storage is similar to `C:\Documents` and `Settings\Username\My Documents`.

The user repository allows access from multiple clients to the same document.

For example, a user creates a document on their office computer, saves the document to the repository. The same user can access that document from a different computer.

In Vault, the user repository provides access to the repository from any connected computer using a roaming repository profile. The objects stored in the user repository are not saved with version control. A user can delete the contents in their user repository.

Folders

Users can organize documents and objects in a repository by grouping them in folders. A folder is synonymous with a directory. Repository folders provide a thin layer of organization such as requesting and grouping around objects in a repository. Properties and methods in the

`Symyx.Framework.Vault.Folder` class allow navigation of the folder hierarchy and enumeration of documents within a folder. Object operations are performed in the `VaultRepository` or `UserRepository`, rather than the folder.

Retrieve Vault Users List Example

The following example shows a complete C# application that you can compile and run. The program demonstrates how to retrieve the list of users from Vault.

```
using System;using Symyx.Framework.Vault;
namespace Symyx.SDK.Framework.Examples
{
    class DisplayUsers
    {
        static void Main(string[] args)
        {
            // declare strings to store the vault server name, domain, user
            // name, and password
            string endpoint; string domain; string user; string password;
            if (args.Length == 4)
            {
                // if the vault server name, domain, username, and password
                // are entered on the command line, use those entries
                endpoint = args[0];
                domain = args[1];
                user = args[2];
                password = args[3];
            }
            else
            {
                // otherwise, prompt the user to enter the vault server name,
                // domain, username, and password
                endpoint = GetConsoleValue("Server");
                domain = GetConsoleValue("Domain");
                user = GetConsoleValue("User");
                password = GetConsoleValue("Password");
            }
            // attempt to connect to the vault server
            Vaultworkspace workspace = new Vaultworkspace(endpoint);
            workspace.Login(new DomainUsernameCredentials
                (domain, user, password));
            // check that the user workspace is authenticated
            if (workspace.IsAuthenticated)
            {
                Console.WriteLine("List of Vault users:");
                // Get the list of vault users with
                // Symyx.Framework.Vault.VaultObjectTypes.User. Note that
                // Symyx.Framework.Vault.VaultObjectType.User creates
                // a new instance of type User.
                VaultObjectList users =
                    workspace.Current.SiteRepository.Get
                        (VaultObjectTypes.User,
                         DataScope.Properties,
                         RetrievalOptions.None);
            }
        }
    }
}
```



```

        // for each user, display some of the user properties
        foreach (User vaultUser in users)
        {
            Console.WriteLine("User ID: " + vaultUser.VaultId);
            Console.WriteLine("Title: " + vaultUser.Title);
            Console.WriteLine("Path: " + vaultUser.VaultPath);
            Console.WriteLine("Type: " + vaultUser.Type);
        }
    }
    else
    {
        Console.WriteLine(string.Format("Could not log in to vault"));
    }
}
// prompt the user to enter a string
private static string GetConsoleValue(string name)
{
    Console.Write("{0}: ", name); return Console.ReadLine();
}
}
}

```

Example output

```

List of Vault users:
User ID: User.005c4ccd-71d8-4a56-91d0-1e595ab07ec2
User name: vm-vault64\Dara.Ward
Path: Site\vm-vault64\Dara.Ward
Object type: User
User ID: User.01875c4c-0ef1-4636-9b4e-18c78d57c031
User name: vm-vault64\Rebecca.Jones
Path: Site\vm-vault64\Rebecca.Jones
Object type: User
...

```

Vault Objects

You can do the following with a Vault object:

- Address the object independently with URL-style references
- Maintain a history of its changes
- Secure the object
- Participate in workflows
- Maintain a series of flags to indicate the object status such as system, hidden, or remove from the table without deleting from the database (soft delete)

Vault objects are serialized in XML-format and stored in repositories. Framework also allows for customized serialization. Some repositories allow the deletion of objects stored in them, others allow soft deletion only.

The `Symyx.Framework.Vault.VaultObject` class is a base class for Vault objects. Actual Vault objects have a specific type that is specified using `Symyx.Framework.Vault.VaultObjectType`

such as User, Folder, Form, Document. The `VaultObjectType` exposes a number of methods and properties that allow programs to use the specific Vault object types.

The parts of a Vault object are:

- **Vault object properties**

The Vault object properties are name/value pairs, represented by the Framework classes `Property` and `Property<T>`, where T is any value that is serialized by the `NetDataContractSerializer`. Properties are grouped together into a `PropertyCollection`.

`VaultObjectProperty` extends `Property` by adding additional attributes that indicate:

- If the property is hidden.
- Who assigns the property; influences ability to edit the property.
- Whether the property is searchable.
- Adaptations to use the `CoreProperty` enumeration to access the core properties.

- **Vault object content**

The Vault object content is a set of bytes defined by each specific Vault object implementation. A Vault object is not required to have content. The content bytes are compressed by default when saved.

When saving to a managed repository, saving content creates a new version of the Vault object. In the User Repository, because only one instance of a Vault object exists at a time, saving content updates the content of the single version in the repository.

- **Vault object preview image**

Every version of the Vault object can have an associated preview image. There is a one-to-one relationship between a preview image and the content. The preview is accessible via the `Preview` core property.

- **Vault object members**

A Vault object can have a membership in another Vault object. A Vault object can have members. The Vault container type has an elevated strong relationship to its members.

The main distinction between properties and content is that the non-server assigned properties are not versioned. There is one and only one copy of the properties. The properties apply to every version of the vault object, even if a particular property did not exist or was different when that earlier version of the vault object was created. There are server properties that are supplied when retrieving a Vault object that are possibly specific to a particular version, but these are not editable and are provided by the server at retrieval time.

Vault Object Base Classes

There are three base classes that are derived from the `VaultObject` class:

- **`VaultElement`**

A Vault element is an object of the `VaultElement` class. A Vault element is a leaf object (file) that cannot contain any other Vault objects. The `VaultElement` class describes a non-container Vault object.

- **`VaultObjectCollection`**

A Vault collection is an object of the `VaultObjectCollection` class. A Vault collection represents a collection of objects and provides a hierarchy in a Vault repository. The objects in a collection need not have a common type and they may belong to multiple collections. An example is a list of a user's favorite Vault folders.

■ VaultObjectContainerBase

A Vault container is an object of the `VaultObjectContainerBase` class. A Vault container knows about the versions of the objects it contains, while a Vault collection does not know about the versions of the object that it contains. An example of a container is a Vault document. A Vault object in a container cannot be included in any other container, that is, a container owns all of its contained objects. However, you can include an object in one container, the only one, and in one or more collections. The Vault container represents a relationship between a parent object and its children. The children do not generally make sense outside of the context of their parent, although they may be operated on in isolation. Updating a child object causes the parent object to increment its version.

The `VaultContainer` class, derived from `VaultObject`, is an ordered or unordered collection of other Vault objects. The `VaultContainerTypes` enumeration lists the different types of Vault containers.

The retrieval of a Vault object is controlled by two parameters:

■ RetrievalOptions

Specifies how the retrieval should be performed. For example, it specifies whether or not an exception is thrown if an object is not found. For details, see the enumeration [Symyx.Framework.Vault.RetrievalOptions](#).

■ DataScope

Specifies how much or which part of the data to retrieve. For example, it specifies whether to retrieve only the properties but not the contents of an object. For details, see the enumeration [Symyx.Framework.Vault.DataScope](#).

Vault Object Flags Property

The `VaultObject.Flags` property contains the `VaultObjectFlags` enumeration that contains the complete set of flags assigned to the object. To set the value of a core property of a `VaultObject`, you use the `VaultWorkspace.UpdateFlags` method that takes as parameters, a `VaultObject` and the `VaultObjectFlags`.

The following C# example sets the `ReadOnly` flag of a Vault object:

```
workspace.UpdateFlags
    (vaultObject, vaultObject.Flags | VaultObjectFlags.ReadOnly);
```

Vault Object Core Properties

A Vault object has metadata consisting of core properties that describe the basic properties of a Vault object. The core properties are based on standards set by the [Dublin Core Metadata Initiative](#). Core properties are assigned by the Framework, the repository, a client application, or an end-user. Some core properties are read-only.

The `VaultObject.CoreProperties` property contains instances of the core properties. Each property is represented by the `Symyx.Framework.Properties.Property` class that is also the base class for all other Vault object property classes. The property keys are defined in `Symyx.Framework.CoreProperty`.

The following C# example gets the `Title` core property of a Vault object:

```
string title = vaultObject.CoreProperties.GetValue<string>
    (CoreProperty.Title);
```

or

```
Symyx.Framework.Properties.Property  
    titleProperty = vaultObject.CoreProperties[CoreProperty.Title];
```

To get the Title core property as a string, use:

```
string title = vaultObject.Title;
```

The following C# example sets the Title core property of a Vault object:

```
vaultObject.CoreProperties.SetValue(CoreProperty.Title, "New Title");
```

Or

```
vaultObject.CoreProperties[CoreProperty.Title].Value = "New Title";
```

Display Name	Property Key	Value Data Type Data Type	Description	Usage
Associations	Associations	Associations	The associations assigned to the object.	Assigned by the Framework.
Association Target	AssociationTarget	Boolean	True if the object can be the target of a user-defined association	Assigned by user
Auto Name	Autoname	String	The auto name string mask	Assigned by a client application. The string mask is used to generate a corporate identifier on Vault.
Check-in Signature Policy	CheckInSignaturePolicy	CheckInSignaturePolicies	The signature policies that apply when the object is checked in.	Assigned by a client application.
Availability	CheckoutState	CheckoutState	Check-out state of the object at the time it was retrieved from the server. Read-only.	Assigned by Vault managed repository. Includes state (New, Available, ExclusivelyCheckedOut, NotAvailable), user name, user ID, and timestamp. Populated by the server at the time that the object is retrieved from the server; might not necessarily reflect the current checkout state. Set to CheckoutState.New prior to the object's first check-in.
Creation Date	ClientCreationDate	DateTimeOffset	A point or period of time associated with an event in the lifecycle of the resource. Read-	Assigned by the Framework. The date and time that the object was originally created on the client, in UTC. Populated by the VaultObject constructor

Display Name	Property Key	Value Data Type Data Type	Description	Usage
			only.	
Size	ContentSize	Integer	Size of the object data. Read-only	Assigned by the source repository. The size of the object's content stream, as computed by Vault. For experiments, this is the size of the document object itself as stored in the database, excluding the sections within the experiment, which are stored as separate objects in the database.
Contributor(s)	Contributor	String	An entity responsible for making contributions to the resource.	Assigned by user. User description of others contributing to the object's content, for example, technicians and investigators.
Scope	Coverage	String	The spatial or temporal topic of the resource, the spatial applicability of the resource, or the jurisdiction under which the resource is relevant.	Assigned by user. User description of the scope of the object.
Created By	Creator	VaultId	An entity primarily responsible for making the resource. Read-only.	Assigned by Vault managed repository. The ID of the user making the first Vault check-in of the object. Assigned by the server; empty for objects that have not been checked into Vault.

Display Name	Property Key	Value Data Type Data Type	Description	Usage
Description	Description	String	An account of the resource	Assigned by user. User description of the object, such as an abstract, a table of contents, or a free-text account of the content.
Flags	Flags	Integer	Describes the system-level flags that can be applied on a Vault object.	Assigned by the source repository. The ReadOnly flag overrides the security permissions to write, update, or delete the object, and it may require a signing policy to unlock. The Hidden flag has no bearing on security settings; clients can handle hidden objects as appropriate. The System flag has no bearing on security settings; clients can handle system objects as appropriate. The Archive flag is frequently used by backup software to do incremental backups. It identifies an object that has reached a point in its lifecycle (workflow) that makes it eligible for archiving.
Format	Format	String	The file format, physical medium, or dimensions of the resource. Read-only.	Assigned by the Framework. The media type of the object in MIME-type format.
ID	Identifier	VaultId	An unambiguous reference to the resource within a given context. Read-only	Assigned by the Framework. The system GUID for the object. Populated by the VaultObject constructor.

Display Name	Property Key	Value Data Type Data Type	Description	Usage
IsLocked	IsLocked	Boolean	Indicates whether or not the Vault object is locked by the user	Assigned by user
Language	Language	String	A language of the resource	Assigned by user. User description of the language used by the object.
ObjectData	ObjectData	Byte[]	Hidden object properties	For sections, an XML representation of DocumentSection base class properties
PendingContentHistory	PendingContentHistory	MutableContentHistory	A list of pending content history entries	Assigned by the Framework. Used to capture content history entries before they are committed to Vault
Permissions	Permissions	ObjectPermissions	Current permissions of the object. Read-only.	Assigned by the source repository. Contains both explicit and implicit permissions. Populated by the server at the time that the object was retrieved from the server; may not necessarily reflect the current permissions.
Preview Section	Preview	Image	An implementation-defined preview image of the object	Assigned by a client application. For sections, an implementation-specific rendering of the section's contents. For documents, a rendering of the section designated as the preview section.
PropertySets	PropertySets	List<PropertySetIdentifier>	A list of property set identifiers	Assigned by the Framework. The PropertySetIdentifier is a VaultUri if the PropertySetDefinition is managed, or otherwise a key.
Published By	Publisher	String	An entity responsible for	Assigned by the Framework. The name and version of the application creating

Display Name	Property Key	Value Data Type Data Type	Description	Usage
			making the resource available. Read-only.	the object, for example, Workbook.
Related To	Relation	String	A related resource	Assigned by user. User description of related work, for example, a literature reference or something semi-abstract like <i>the experiments I did as a postdoc</i> .
Referenced	References	AssociationList	A collection of associations that contain references	Assigned by the Framework. A reference is a type of association in which the target object is a reference to the source object.
RepositoryVersion	RepositoryVersion	String	The version of a repository	Assigned by the source repository.
Rights	Rights	String	Information about rights held in and over the resource	Assigned by user. User description of rights held to the object. This is distinct from permissions, which describe the accessibility of the object to users within the system.
Initial Check-In Date	ServerCreationDate	DateTimeOffset	Creation date and time of the object from the server	Assigned by Vault managed repository. The date and time that the first version of the object was saved to Vault, in UTC
Source	Source	String	The resource from which the described resource is derived	Assigned by the Framework. For documents: title of DocumentTemplate from which the document was created. For sections: title of SectionTemplate from which the section was created. For users: the user name in the

Display Name	Property Key	Value Data Type Data Type	Description	Usage
				underlying directory, for example, Active DirectoryEmpty for other object types.
Subject	Subject	String	The topic of the resource	Assigned by user. User description of the object's topic keywords and key phrases.
Name	Title	String	A name given to the resource	Assigned by user. Title of the object, for display purposes. For users: the full name of the user, for example, <i>John Smith</i> not <i>symyx\john.smith</i> .
Total Content Size	TotalContentSize	Long	The size of the object data and all its children	For experiments, this is the size of the document and its sections, as stored in the database. For file content stored in file sections, this figure includes only size of the metadata about the file, not the file content size itself, which is stored in a separate database schema.
Type	Type	String	The type of the resource	Assigned by the Framework. For documents, <i>Document</i> For document templates, <i>Document Template</i> For sections, <i>Document Section</i> For section templates, <i>Document Section Template</i> For folders, <i>Folder</i> For repositories, <i>Repository</i> For a favorites collection, <i>Favorites</i> For signature policies, <i>Signature Policy</i> For users, <i>User</i> For groups, <i>Group</i> For forms, <i>Form</i>

Display Name	Property Key	Value Data Type Data Type	Description	Usage
Vault Path	VaultPath	String	Location of the object in Vault in directory-style	Assigned by a Vault managed repository. A path-style concatenation of the object's containing folder and its parent folders. Populated by the server; only affected by saves to an authoritative Vault repository, that is not a user repository. Empty if not saved to Vault.
Current Version	Version	Integer	Version number of the object. Read-only.	Assigned by Vault managed repository. An integer value incremented with each content change. Initialized to zero by the Framework; non-zero value assigned by the server.
Version Comment	VersionComment	String	Description of the current version of the object content.	Assigned by Vault managed repository. Corresponds to the user-specified check-in comment for that version.
Version Creation Date	VersionCreationDate	DateTimeOffset	Creation date and time of the object from the server. Read-only	Assigned by Vault managed repository. The date and time that this version of the object was originally checked into Vault, in UTC.
Version Created By	VersionCreator	VaultId	User ID responsible for the current version of the object content. Read-only.	Assigned by Vault managed repository
Workflow Stages	WorkflowSummary	String	A delimited list of the current stage for each of the	Assigned by Vault managed repository.

Display Name	Property Key	Value Data Type Data Type	Description	Usage
			workflows in which the object is currently participating	

Set Properties in VaultObject.ExtendedProperties

The property `VaultObject.ExtendedProperties` contains custom or extended properties of a Vault object. `ExtendedProperties` is a `Symyx.Framework.Properties.ExtendedPropertySet` object that allows you to set the property values.

The following C# example sets an extended property named `key` to the value `myvalue`:

```
vaultObject.ExtendedProperties.SetValue("key", "myvalue");
```

You can also create an enumeration and use the contents as keys. The following enumeration defines keys.

```
named Size and InsertableThings: private enum MyKeys
{
    Size, InsertableThings,
}
```

The following `ExtendedVaultObject` class contains a property that gets and sets `Size`.

```
public class ExtendedVaultObject : VaultObject
{
    public ExtendedVaultObject() : base(VaultObjectType.User)
    {
        // additional constructor code
    }
    // create a Size property public int Size
    {
        // get and set methods get
        {
            return this.ExtendedProperties.GetValue<int>(MyKeys.Size, this.Size);
        }
        set
        {
            this.ExtendedProperties.SetValue(MyKeys.Size, value);
        }
    }
}
```

Use Data Scope to Set Information

The Framework provides various strategies for retrieving Vault objects. Before you see those strategies, you need to understand how to specify the information to be retrieved from a Vault object by setting the Data Scope.

Data Scope allows you to control the information retrieved for a Vault object.

The `Symyx.Framework.Vault.DataScope` enumeration contains options that correspond to the parts of a Vault object that are retrieved.

```
public enum DataScope {
    Minimal,
    Properties,
    Content,
    Members,
    Preview,
    All = Properties | Preview | Content | Members,
}
```

`DataScope.Minimal` specifies that only the following information about the vault object is returned which includes: `Title`, `Description`, `VaultObjectType`, `Class`, and `SourceRepositoryId`.

`DataScope.All` is a shortcut for combining all the available data scopes.

Experiments could contain a large amount of content. Performing gets using `DataScope.Content` or `DataScope.All` can cause significant performance problems and can lead to out of memory errors.

- Do not execute a `DataScope.Content` or `DataScope.All` get methods unless the experiment content is needed.
- Avoid executing `DataScope.Content` or `DataScope.All` get methods on more than one document a time.

In a few cases a repository might return more information than is requested. For example, `VaultRepository` returns properties even if you only `DataScope.Content` was specified.

The following C# example shows how to use `DataScope.Content` to retrieve the Vault object content, and displays the `DataScope` and retrieval properties.

```
VaultObject vaultObject = repository.Get(vaultId, DataScope.Content);
Console.WriteLine(vaultObject.DataScope);
// the DataScope
Console.WriteLine(vaultObject.PropertiesRetrieved);
// will be false
Console.WriteLine(vaultObject.ContentRetrieved);
// will be true
Console.WriteLine(vaultObject.PreviewRetrieved);
// will be false
Console.WriteLine(vaultObject.MembersRetrieved);
// will be false
```

The following C# example shows how to use multiple `DataScope` combinations to retrieve the Vault object properties and preview image:

```
VaultObject = repository.Get
    (vaultId, DataScope.Properties | DataScope.Preview);
Console.WriteLine(vaultObject.DataScope);
// the DataScope
Console.WriteLine(vaultObject.PropertiesRetrieved);
// will be true
Console.WriteLine(vaultObject.ContentRetrieved);
// will be false
Console.WriteLine(vaultObject.PreviewRetrieved);
// will be true
Console.WriteLine(vaultObject.MembersRetrieved);
// will be false
```

`DataScope` applies to the objects to retrieve. In the case of a `GetMembers()` method call, `DataScope` applies to the members of the object.

The following C# example retrieves the subfolders of a folder (all of the members for subfolders are also retrieved):

```
Folder subFolder = (Folder) repository.Get
    (folder.VaultId, DataScope.Properties | DataScope.Members);
foreach (VaultObject folder2 in subFolder)
{
```

```
    Console.WriteLine(folder2.MembersRetrieved);
}
```

When retrieving a Vault object that is a container such as documents and folders, `DataScope.Content` applies to the members when `DataScope.Members` is specified. On other types, the content for members is not retrieved. For example, in C#:

```
// the members of the container will be retrieved
VaultObject vaultObject = repository.Get(documentId, DataScope.All);
// the members of the folder will not be retrieved
vaultObject = repository.Get(folderId, DataScope.All);
```

Retrieve Vault Users Example

The following C# method retrieves the Vault users.

```
private static void RetrieveVaultUsers(Repository repository)
{
    // get the list of vault users
    VaultObjectList users = VaultWorkspace.Current.SiteRepository.Get
        (VaultObjectType.User, DataScope.Properties,
        RetrievalOptions.None);
    foreach (User user in users)
    {
        // display some of the core properties
        Console.WriteLine("Title: " + user.CoreProperties.GetValue
            (CoreProperty.Title));
        Console.WriteLine("Content size: " + user.CoreProperties.GetValue
            (CoreProperty.ContentSize));
        Console.WriteLine("Description: " + user.CoreProperties.GetValue
            (CoreProperty.Description));
        // set a new title
        Symyx.Framework.Properties.Property titleProperty =
            user.CoreProperties[CoreProperty.Title];
        user.CoreProperties.SetValue(CoreProperty.Title, "New Title");
        Console.WriteLine("Title: " + user.CoreProperties.GetValue
            (CoreProperty.Title));
        // set an extended vault property
        user.ExtendedProperties.SetValue("key", "myvalue");
        Console.WriteLine("Key: " + user.ExtendedProperties.GetValue
            ("key"));
        // use a different DataScope and re-retrieve the contents
        VaultObject vaultObject = repository.Get
            (user.VaultId, DataScope.Content);
        Console.WriteLine(vaultObject.DataScope);
        Console.WriteLine(vaultObject.PropertiesRetrieved);
        Console.WriteLine(vaultObject.ContentRetrieved);
        Console.WriteLine(vaultObject.PreviewRetrieved);
        Console.WriteLine(vaultObject.MembersRetrieved);
        // use a different DataScope and re-retrieve the contents
        vaultObject = repository.Get
            (user.VaultId, DataScope.Properties | DataScope.Preview);
        Console.WriteLine(vaultObject.DataScope);
        Console.WriteLine(vaultObject.PropertiesRetrieved);
        Console.WriteLine(vaultObject.ContentRetrieved);
    }
}
```

```
        Console.WriteLine(vaultObject.PreviewRetrieved);
        Console.WriteLine(vaultObject.MembersRetrieved);
    }
}
```

Retrieve a Vault Object using Vault ID or Vault URI

The following C# example gets a Vault object using a Vault ID:

```
VaultObject vaultObject = repository.Get
    (vaultId, DataScope.Properties);
```

In the above code, repository represents a Vault Repository object.

A Vault URI can have the version number specified as part of the VaultId.

The C# method calls are:

```
VaultObject Get(VaultId vaultId, Version version, DataScope dataScope);
VaultObject Get(VaultUri vaultUri, DataScope dataScope);
```

For example:

```
VaultObject vaultObject = repository.Get
    (vaultId, new Symyx.Framework.Version(1), DataScope.Properties);
vaultObject = repository.Get
    (new VaultUri
        (vaultId, new Symyx.Framework.Version(1)), DataScope.Properties);
```

A version that is empty (Version.Empty) is the same as only specifying the Vault ID. A request without a version returns the latest version of that object. When a version is specified, the local object cache might determine that object if it has been retrieved before.

The first version of a Vault object has the version number of 1. The following C# example retrieves a document using a Vault ID. The DataScope.All parameter specifies that the retrieved object includes its properties and contents. The RetrievalOptions.SuppressNotFoundExceptions parameter specifies that an exception is not thrown if the specified Vault ID is not found.

```
VaultObject vaultObject = repository.Get
    (vaultId, DataScope.All, RetrievalOptions.SuppressNotFoundExceptions);
```

Create a Batch For Identity Requests

You can specify multiple VaultIds or VaultUris identities and retrieve the Vault objects in a single server call. The following C# shows the Get method signatures.

```
VaultObjectList Get
    (IEnumerable <VaultId> vaultIds, DataScope dataScope);
VaultObjectList Get
    (IEnumerable <VaultUri> vaultUris, DataScope dataScope);
```

The following C# example shows how the first Get method is called to retrieve multiple Vault objects, and return the objects in a VaultObjectList.

```
List<VaultId> vaultIds = GetVaultIds();
VaultObjectList list = repository.Get(vaultIds, DataScope.Properties);
```

When supplying versions using VaultUris, the Vault objects that can be retrieved from the local object cache are retrieved. The other Vault objects are retrieved from the Vault server.

Retrieve and Cast a Vault Object

All of the identity Get methods that return a single object allow you to return the object as a specific type. For example, in C#:

```
TVaultObject Get<TVaultObject>(VaultId vaultId, DataScope dataScope) where
TVaultObject : VaultObject;
```

The following returns a Group object when the Get method is called:

```
VaultId groupId = vaultObject.VaultId;
Group group = repository.Get<Group>(groupId, DataScope.Properties);
```

The Get method throws an error if the retrieved type cannot be cast to the supplied generic parameter type. You can specify any valid base class in the object's inheritance hierarchy. The following C# example casts the retrieved object to a Notebook.DocumentSection:

```
Symyx.Notebook.DocumentSection section =
repository.Get<DocumentSection>(sectionId, DataScope.Properties);
```

This example is valid for a TextSection, FormsSection, or any custom section class that derives from the DocumentSection class.

Identity Requests Scope

Requests sent to a specific repository find only the Vault objects that are located in that repository. Requests using VaultWorkspace.Current use the Vault object location service on the Vault Server machine to determine the appropriate repository. For example, in C#:

```
VaultObject vaultObject = VaultWorkspace.Current.Get
(vaultId, DataScope.Properties);
```

The call can only locate objects in managed repositories. Items in the User Repository are not found. The exception is that a mixed list of VaultUri's can be passed to the current workspace. An example is a set of VaultShortcuts that might point to objects in many different repositories.

A VaultUri that contains a supplied repository VaultId always causes Vault Server to look in that repository only for that Vault object. If a such a VaultUri is passed to a specific repository that is different, an error is thrown. For example, in C#:

```
VaultUri vaultUri = new VaultUri
(vaultworkspace.Current.Repositories[0].VaultId, vaultId);
VaultObject vaultObject = VaultWorkspace.Current.Repositories[1].Get
(vaultUri, DataScope.Properties);
```

Retrieve Vault Objects by Object Type

To retrieve Vault objects by a Vault object type, you use the Get method that accepts Vault object types:

```
VaultObjectList Get
(VaultObjectTypes objectTypes, DataScope dataScope);
```

For example, the following C# statement retrieves the Vault folder objects:

```
VaultObjectList list = repository.Get
(VaultObjectTypes.Folder, DataScope.Properties);
```

The following C# example retrieves Vault objects of multiple types:

```
VaultObjectList list = repository.Get  
    (VaultObjectTypes.Folder | VaultObjectTypes.SignaturePolicy |  
     VaultObjectTypes.WorkflowDefinition, DataScope.Properties);
```

The `VaultObjectTypes.All` property can be used to retrieve all types, but the `Get` methods work against the entire repository. Typically, you would use `VaultObjectTypes.All` in conjunction with a context-based Vault object retrieval using the method `GetMembers`.

The repository might contain a large number of vault objects of a given type in Vault. Searching by object type can cause performance problems or out of memory exceptions.

Search for a Vault User Example

The following C# method shows how to search for a user whose user name ends with the string, *admin*.

```
private static void SearchForUser()  
{  
    // get the list of Vault users  
    VaultObjectList users = VaultWorkspace.Current.SiteRepository.Get  
        (VaultObjectType.User, DataScope.Properties,  
         RetrievalOptions.None);  
    // loop over the users and search for a username that ends with admin  
    foreach (User user in users)  
    {  
        Console.WriteLine("User name: " + user.UserName);  
        if ((user as User).UserName.ToLower().EndsWith("admin"))  
        {  
            Console.WriteLine("User found");  
        }  
    }  
}
```

Retrieve Vault Assemblies

The following C# method shows how to retrieve Vault assemblies.

```
private static void RetrieveAssemblies()  
{  
    // get the list of assemblies  
    VaultObjectList assemblies = VaultWorkspace.Current.SiteRepository.Get  
        (VaultObjectType.Assembly, DataScope.Properties);  
    // display the assembly details  
    assemblies.ForEach(delegate(VaultObject vaultObject)  
    {  
        Console.WriteLine("Assembly: {0}, Created: {1}", vaultObject.Title,  
            vaultObject.ServerCreationDate);  
    }  
}
```

Use GetMembers For Context Retrieval

For Vault objects that implement the `IMemberBasedVaultObject`, which means that they reference other Vault objects, those members can be directly accessed using the `GetMembers` method:

```
VaultObjectList GetMembers  
(IMemberBasedVaultObject memberBasedVaultObject, DataScope dataScope);
```

The `GetMembers` method returns a list of the members. The `DataScope` applies directly to the members, not the member-based object passed into the method.

The following C# example returns the contents of a folder:

```
VaultObjectList list = repository.GetMembers
(folder, DataScope.Properties);
```

The base `Repository` implementation derives from `Folder`, so the repository can be passed into the `GetMembers` method as the root folder of the repository. For example, the following C# example gets the items located in the root repository folder:

```
VaultObjectList list = repository.GetMembers
(repository, DataScope.Properties);
```

This is different from the following C# example, which gets all of the items in the repository:

```
VaultObjectList list = repository.Get
(VaultObjectTypes.All, DataScope.Properties);
```

When retrieving objects from a managed repository, the members are retrieved for the version of the object passed into the `GetMembers` call. For non-container member-based objects, the members are not version based, so this is not important. For containers, it provides a means to retrieve members for a previous version. When making the call against a non-managed repository (the User Repository), the supplied version is not considered because only one version is stored at a time.

Combine Vault object types in a `GetMembers` call

When getting the members of a Vault object, it can be useful to limit the retrieval to only certain types of members. For this reason, the following `GetMembers` method is provided that allows you to specify the Vault object type(s):

```
VaultObjectList GetMembers
(IMemberBasedVaultObject memberBasedVaultObject,
VaultObjectTypes objectTypes, DataScope dataScope);
```

The following C# example returns the subfolders of a folder:

```
VaultObjectList subFolders = repository.GetMembers
(folder, VaultObjectType.Folder, DataScope.Minimal);
```

Retrieve Vault Folders

The following C# method shows how to retrieve the Vault folders:

```
private static void RetrieveVaultFolders
(Repository repository, Workspace workspace)
{
    // get the list of Vault folders VaultObjectList folders =
    VaultWorkspace.Current.SiteRepository.Get
    (VaultObjectType.Folder, DataScope.Properties,
    RetrievalOptions.None);
    VaultId vaultId = new VaultId();
    // loop over the folders
    foreach (Folder folder in folders)
    {
        Console.WriteLine("Folder title: " + folder.CoreProperties.GetValue
        (CoreProperty.Title));
        // get the subfolders of the folder and display their details
        Folder subFolder = (Folder) repository.Get
```

```

        (folder.VaultId, DataScope.Content);
VaultObjectList subFolders = repository.GetMembers
    (subFolder, VaultObjectTypes.Folder, DataScope.All);
foreach (VaultObject folder2 in subFolders)
{
    Console.WriteLine(folder2.MembersRetrieved);
}
vaultId = folder.VaultId;
}
// display the Vault ID
Console.WriteLine("Vault ID: " + vaultId);
// when the Vault ID is known, a Vault object can be retrieved
// using that Vault ID
VaultObject vaultObject = repository.Get
    (vaultId, DataScope.Properties);
// get a specific version of a Vault object
VaultObject vaultObject2 = repository.Get
    (vaultId, new Symyx.Framework.Version(1), DataScope.Properties);
VaultObject vaultObject3 = repository.Get
    (new VaultUri(vaultId, new Symyx.Framework.Version(1)),
    DataScope.Properties);
// requests sent to a specific repository only finds Vault objects
// that are located in that repository.
// Alternatively, requests sent using VaultWorkspace.Current will
// use the Vault object location service
// to determine the appropriate repository. For example:
VaultObject vaultObject4 = VaultWorkspace.Current.Get
    (vaultId, DataScope.Properties);
// get a list of folder objects
VaultObjectList list = repository.Get
    (VaultObjectTypes.Folder, DataScope.Properties);
Console.WriteLine("list = " + list);
// get a list of folders, signature policies, and workflow
// definitions
VaultObjectList list2 = repository.Get
    (VaultObjectTypes.Folder | VaultObjectTypes.SignaturePolicy |
    VaultObjectTypes.WorkflowDefinition, DataScope.Properties);
Console.WriteLine("list2 = " + list2);
// get the members of a folder as a list
Folder myFolder = (Folder) repository.Get
    (vaultId, DataScope.Properties);
VaultObjectList list3 = repository.GetMembers
    (myFolder, DataScope.Properties);
Console.WriteLine("list3 = " + list3);
// get the list of items located in the root repository folder
VaultObjectList list4 = repository.GetMembers
    (repository, DataScope.Properties);
Console.WriteLine("list4 = " + list4);
// get the list of ALL the items located in the repository
VaultObjectList list5 = repository.Get
    (VaultObjectTypes.All, DataScope.Properties);
Console.WriteLine("list5 = " + list5);
}

```

Retrieve Hierarchy of Vault Folders Example

The following C# method shows how to retrieve the hierarchy of Vault folders, starting with the root repository folder.

```
private static void ShowRepositoryFolderHierarchy
(VaultRepository repository)
{
    // call ShowFolders to display the folder hierarchy
    ShowFolders(repository, repository, 0);
}
private static void ShowFolders
(VaultRepository repository, Folder folder, int level)
{
    // display the current folder
    Console.WriteLine
        ("{0}{1}", new string((char)9, level), folder.Title);
    VaultObjectList subFolders = repository.GetMembers
        (folder, VaultObjectType.Folder, DataScope.Minimal);
    // for each subfolder...
    foreach (Folder subFolder in subFolders)
    {
        // call ShowFolders recursively
        ShowFolders(repository, subFolder, level + 1);
    }
}
```

Use Vault Query Service to Get Vault IDs

You can use the Vault query service and provide search operators to retrieve a set of Vault IDs. The Vault IDs can then be used with the Get method to get the actual Vault objects.

To use the query service, you first build a Query object, and then use the `FindVaultIds` method to retrieve the Vault IDs of any matching Vault objects. You then use the Get method to retrieve the Vault objects.

The following C# example creates a query condition to search for an object with the title, *NotebookAssemblies*.

```
CorePropertyQueryCondition titleClause = new CorePropertyQueryCondition
(CoreProperty.Title,
    QueryComparisonOperator.QueryComparisonOperators.EqualTo,
    "NotebookAssemblies");
```

The following query condition searches for Folder objects.

```
CorePropertyQueryCondition typeClause = new CorePropertyQueryCondition
(CoreProperty.Type,
    QueryComparisonOperator.QueryComparisonOperators.EqualTo,
    VaultObjectType.Folder);
```

The following query condition searches for objects with the previous title clause and type clause.

```
Query query = new Query(titleClause & typeClause);
```

The following example runs the query and gets the results back as a list of Vault IDs:

```
List<VaultId> vaultIds =  
    vaultworkspace.Current.SiteRepository.FindVaultIds(query);
```

For details on `FindVaultIds()`, see the Framework API reference.

Also, see the `Query`, `QueryClause`, and `TitleTypeAndQuery` classes in the API (these classes are very important for finding Vault objects).

The following example gets the list of Vault objects.

```
vaultObjectList vaultObjects = repository.Get  
    (vaultIds, DataScope.Properties);
```

Retrieving a Vault object using the query service Example

The following C# method shows how to retrieve the *NotebookAssemblies* folder using the query service:

```
private static void RetrieveVaultObjectUsingQueryService  
(VaultRepository repository)  
{  
    // create a query condition to search for vault objects using a title  
    CorePropertyQueryCondition titleClause = new  
        CorePropertyQueryCondition(CoreProperty.Title,  
            QueryComparisonOperator.QueryComparisonOperators.EqualTo,  
            "NotebookAssemblies");  
    // create a query condition to search for vault objects using an  
    // object type  
    CorePropertyQueryCondition typeClause = new  
        CorePropertyQueryCondition(CoreProperty.Type,  
            QueryComparisonOperator.QueryComparisonOperators.EqualTo,  
            vaultObjectType.Folder);  
    // create a query to search for vault objects with the specified  
    // title and type  
    Query query = new Query(titleClause & typeClause);  
    // run the query and get the vault object IDs back as a list  
    List<VaultId> vaultIds =  
        vaultworkspace.Current.SiteRepository.FindVaultIds(query);  
    Console.WriteLine("vaultIds = " + vaultIds);  
    // get the list of vault objects  
    vaultObjectList vaultObjects = repository.Get  
        (vaultIds, DataScope.Properties);  
    Console.WriteLine("vaultObjects = " + vaultObjects);  
    foreach (VaultObject vaultObject in vaultObjects)  
    {  
        Console.WriteLine  
            ("Vault object title: " + vaultObject.CoreProperties.GetValue  
                (CoreProperty.Title));  
    }  
}
```

Retrieving a Vault Object Using the Query Service

The following IronPython example gets the Vault IDs where the subject is `Symyx.Notebook` and the title is `Forms.Editor`:

```
from Symyx.Framework import  
    (CoreProperty, CorePropertyQueryCondition, Query,  
     QueryComparisonOperator)
```

```

from Symyx.Framework.Vault import (Vaultworkspace, VaultId, DataScope) from
System.Windows.Forms import MessageBox
subjectClause = CorePropertyQueryCondition
    (CoreProperty.Publisher,
    QueryComparisonOperator.QueryComparisonOperators.EqualTo,
    "Symyx.Notebook")
titleClause = CorePropertyQueryCondition
    (CoreProperty.Title,
    QueryComparisonOperator.QueryComparisonOperators.EqualTo,
    "Forms.Editor")
query = Query(subjectClause & titleClause)
vaultIds = Vaultworkspace.Current.SiteRepository.FindVaultIds(query)
if vaultIds.Count > 0: MessageBox.Show(vaultIds[0].ToString())
else: MessageBox.Show('Object not found')

```

You can paste the script into the OnApplicationLoaded event in the Event Scripting area of the Experiment Editor.

1. In the Experiment Editor, choose **View > Properties**.
2. In the **Event Scripting** area, click the ellipsis button.
3. Select **OnApplicationLoaded**.
4. Click **Add Script**.
5. Paste the script into the code area, and click **OK**.

Convert Vault IDs

A Vault ID is a 128-bit positive integer. The following examples show a Document Vault ID and a User Vault ID, the 128-bit integer is shown as a hexadecimal number after the Vault object type:

```

Document.382c74c3-721d-4f34-80e5-57657b6cbc27 User.ff2d59a7-7d08-4a5b-97be-
e06df1cf0b33

```

You can change a Vault ID to a string, and convert a string that contains a Vault ID to a Vault ID.

To convert a Vault ID to a string, you use the ToString() method. For example, in C#:

```
string vaultIdString = vaultId.ToString();
```

To check that a string contains a valid Vault ID, you use the TryParse() to examine the supplied string and attempt to convert the string to a Vault ID. If the supplied string is a valid Vault ID, TryParse() returns true, otherwise the method returns false.

Converting Vault IDs to Strings Example

The following C# method shows how to convert Vault IDs to strings:

```

private static void ConvertVaultIDSToStrings()
{
    // get the list of vault folders
    VaultObjectList folders = Vaultworkspace.Current.SiteRepository.Get
        (VaultObjectType.Folder, DataScope.Properties,
        RetrievalOptions.None);

    // loop over the folders
    foreach (Folder folder in folders)
    {
        // get the vault ID
    }
}

```

```

VaultId vaultId = folder.VaultId; Console.WriteLine
    ("Vault ID: " + vaultId);

// convert the Vault ID to a string
string vaultIdString = vaultId.ToString();

// check a supplied string contains a valid Vault ID using
TryParse() bool isValidVaultId = VaultId.TryParse
    (vaultIdString, out vaultId);

// if the Vault ID is valid, retrieve the Vault object using
Get() if (isValidVaultId)
{
    Console.WriteLine("Vault ID is valid");
    VaultObject vaultObject = VaultWorkspace.Current.SiteRepository.Get
        (vaultId, DataScope.All);
    Console.WriteLine("Vault object title: " + vaultObject.Title);
}
}

```

Vault Object References and Associations

A Vault object can be referenced by either its `VaultId` or `VaultUri`. The `VaultObject.Associations` property contains a collection of Association objects that allow you to capture types of relationships between Vault objects.

An association represents a relationship between two Vault objects. The basic constituents of an association is `Type`, which is the association type listed in the `Symyx.Framework.Vault.AssociationTypes` enumeration. The following table shows examples of types of associations between two Vault objects.

Type	Relationship between two objects
Parent	Object1 is a parent of Object2. For example, a document has this relationship with each of its sections
ContainedObject	Object1 is contained in Object2. For example, a section has this relationship with the document that contains it, and a document in a search result has this relationship with the object returned by the search. Note that [obj1 "is contained in" obj2 = TRUE] implies [obj2 "is parent of" obj1 = TRUE].
CloneSource	Object 1 is a clone of Object2
Template	Object2 is a template of Object1. For example, a document or a section has this relationship with the template that was used to create that document or section.
Dependency	Object1 is dependent on Object

Source object, which is the Vault object whose Associations property contains its target associations.

Target object, which is the Vault ID of the target, or right operand of the association. It is stored in the `Association.TargetVaultId` property.

Description, which describes the association or relationship between the objects. The following C# example creates a parent association between two Vault objects:

```
Association association = new Association
    (AssociationTypes.Parent, targetObject, true, "Child object");
arentVaultObject.Associations.Add(association);
```

The `Symyx.Framework.workflow.workflowActorAssociation` provides a high-level abstraction to an association. `workflowActorAssociation` is a helper class that allows an application to associate a Vault object (for example, a user) with an actor in a workflow.

The following C# example assigns a supervisor (Mary) to a user (Jane) by adding a supervisor actor association to a user:

```
workflowActorAssociation assoc = new workflowActorAssociation
    (VaultObject<"Mary">, workflowActor<"Supervisor">);
User<"Jane">.Associations.Add(assoc);
```

Vault URIs

The objects and resources managed by the Vault are identified by a Uniform Resource Identifier (URI) whose generic syntax conforms to the definitions found in the internet Request For Comments document RFC 3986.

The generic syntax for a URI consists of a hierarchical sequence of components referred to as the scheme, authority, path, query, and fragment. For example:

```
foo://example.com:8042/dir1/retrieve?name=get_row#132
```

Where:

- `foo` is the Scheme
- `example.com:8042` is the Authority
- `dir1` is the Path
- `retrieve?name=get_row` is the Query
- `132` is the Fragment

This section describes each component of the URI syntax used by Vault.

Scheme

In the following examples, the scheme for a Vault URI is represented by the characters "vault". Although schemes are case-insensitive, the canonical form is lowercase.

Authority

The authority for a Vault URI optionally designates the Vault server endpoint that is responsible for fulfilling a resource request.

Authority Syntax

The authority component is preceded by a double slash ("//") and is terminated by the next slash ("/"), question mark ("?"), or number sign("#") character, or by the end of the URI.

When an authority is not present, the path cannot begin with two slash characters ("//"). When an authority is present, the path must begin with a slash ("/") character to imply that it is not a rootless path. For more information, see [Path](#).

The authority field consists of a host string and an optional port number. The any of the following are valid for the host field:

A registered name intended for lookup in the DNS, using the syntax defined in Section 3.5 of RFC 1034 and Section 2.1 of RFC 1123.

An IPv4 literal address is represented in dotted-decimal notation, a sequence of four decimal numbers in the range 0 to 255, separated by ".", as described in RFC 1123 by reference to RFC 0952.

A Internet Protocol literal address, version 6, as described in RFC 3513 or later, distinguished by enclosing the IP literal within square brackets "[" and "]").

Example Vault URIs:

```
vaurl://dev02.symyx.com/Repository.01234567-0123-0123-0123-0123456789AB/Document.01234567-0123-0123-0123-0123456789AB
```

```
vaurl://192.168.1.1:1234/Repository.01234567-0123-0123-0123-0123456789AB/Document.01234567-0123-0123-0123-0123456789AB
```

```
vaurl://[3ffe:1900:4545:3:200:f8ff:fe21:67cf]/Repository.01234567-0123-0123-0123-0123-0123456789AB/Document.01234567-0123-0123-0123-0123456789AB
```

Authority Usage

The authority field is primarily useful for applications that must interact directly with multiple Vault instances, for example, direct coordination of Vault servers, or server-side reparse points. For most client applications, the authority field is not necessary: the URI is simply passed to the client's active Vault server for processing. Distributed objects are located by other means, for example, the Vault object locator service and Vault federation.

Path

The path component for a Vault URI identifies the specific resource within Vault.

Rootless path

In the simplest form, the path component simply specifies the Vault ID of the resource. Responsibility for locating the resource is delegated to the Vault object locator service. In this form, the Vault ID appears directly after the scheme separator, without a proceeding slash ("/") character.

Example:

```
vaurl:Document.01234567-0123-0123-0123-0123456789AB
```

Absolute path

If the path begins with a slash ("/") character, then the first component of the path specifies the Vault ID of the repository containing the resource, and the last component of the path specifies the Vault ID of the resource. In this form, the target repository is determined from the URI without consulting the Vault object locator service. Because the resource ID is specified, no additional information is necessary to locate the resource within the scope of the repository.

Example:

```
vaurl:/Repository.01234567-0123-0123-0123-0123456789AB/Document.01234567-0123-0123-0123-0123456789AB
```

Absolute path names

If an individual path component does not match the format of a Vault ID, then it is treated as a title, and Vault attempts to locate the object by navigating the provided path. In this form, the target repository must be specified, and the Vault object locator service is not consulted. Because titles of Vault Objects

are not guaranteed to be unique, this operation does not authoritatively identify an object, and objects can be located only on a best-effort basis.

Example:

```
vault:/Repository1/Folder2/Document3
```

Version Specifiers

If any path component includes a trailing integer version number delimited by a comma (",") character, then it is treated as a reference to a specific version of the resource. If that component does not include a trailing integer delimited by a comma, then it is treated as a reference to the latest version of the resource.

Examples:

```
vault:Document.01234567-0123-0123-0123-0123456789AB,3
```

Note: This is version 3 of the Vault object.

```
vault:Document.01234567-0123-0123-0123-0123456789AB
```

Note: This is the latest version of the Vault object.

Query

The query component of a Vault URI provides a mechanism for refining the scope of an identified resource. These values correspond to the values defined in the `Symyx.Framework.Vault.DataScope` enumeration. If no scope is specified, then it is equivalent to `DataScope.All`.

Example:

```
vault:Document.01234567-0123-0123-0123-0123456789AB&scope=members
```

Fragment

The fragment component of a Vault URI allows indirect identification of a secondary resource within the scope of the primary resource that is, some resource in the scope of an identified `VaultObject`. The primary resource is responsible for the interpretation of the fragment field, and might, for example, represent a subset of the primary resource, an anchor within it, or an alternate view.

Example:

```
vault:DocumentSection.01234567-0123-0123-0123-0123456789AB#anchor
```

Vault Object Annotations

Vault object annotations provide a mechanism for attaching information to Vault objects. Annotations exist outside of the metadata and content of the object, are not subject to Vault security, and are not subject to audit trails. An annotation can be added to an object even if the user does not have permission to modify that object. Some example uses of annotations are the comment features in Workbook and workflow task features in Vault.

The following C# example creates an annotation for a hypothetical workflow:

```
Annotation annotation = new Annotation("workflow category");
annotation.Category = "workflow";
annotation.Context = "workflow task";
annotation.Title = "workflow task";
annotation.Description = "A test workflow task";
```

The following example adds the annotation to a user:

```
annotation.Recipients.Add(vaultUser.VaultId);
```

The following example adds the annotation to the Site Repository:

```
site.AddAnnotation(vaultUser, annotation);
```

The following example gets the annotations assigned to the user:

```
List<Annotation> annotations = workspace.GetAnnotationsForUser  
(vaultUser);
```

The following example removes the annotation from a user:

```
annotation.Recipients.Remove(user.VaultId);
```

The following example removes the annotation from the Site Repository:

```
site.RemoveAnnotation(vaultUser, annotation);
```

Add and Remove an Annotation

The following C# method shows how to add and remove an annotation:

```
private static void AddAnnotation(VaultWorkspace workspace)  
{  
    // get the list of Vault users  
    VaultObjectList users =VaultWorkspace.Current.SiteRepository.Get  
        (VaultObjectType.User, DataScope.Properties, RetrievalOptions.None);  
    // create an annotation  
    Annotation annotation = new Annotation("Workflow category");  
    annotation.Category = "Workflow";  
    annotation.Context = "Workflow task";  
    annotation.Title = "Workflow task";  
    annotation.Description = "A test workflow task";  
    // loop over the users  
    foreach (User user in users)  
    {  
        // add the annotation to the user  
        Console.WriteLine("Adding annotation to " + user.Title);  
        annotation.Recipients.Add(user.VaultId);  
        // add the annotation to the site repository  
        Console.WriteLine("Adding annotation to the site repository");  
        VaultRepository site = VaultWorkspace.Current.SiteRepository;  
        site.AddAnnotation(user, annotation);  
        // display the user's annotations  
        List<Annotation> annotations = workspace.GetAnnotationsForUser  
            (user);  
        for each (Annotation myAnnotation in annotations)  
        {  
            Console.WriteLine("myAnnotation title: " + myAnnotation.Title);  
        }  
        // remove the annotation  
        Console.WriteLine("Removing the new annotation");  
        annotation.Recipients.Remove(user.VaultId);  
        site.RemoveAnnotation(user, annotation);  
    }  
}
```

Create and Delete Vault Objects

The following C# example creates a folder named newFolder:

```
Folder newFolder = new Folder("newFolder");
```

The following example adds the new folder to the list of the current user's favorite folders:

```
workspace.UserRepository.Add  
    (newFolder, workspace.CurrentUser.Favorites);
```

The following example deletes the new folder:

```
workspace.UserRepository.Delete(newFolder);
```

Note: Deleting is only allowed if permitted by the repository as specified by the `workspace.RepositoryBehaviors.AllowDeletes` property. In general, deletes are only allowed in the user repository.

Adding and Deleting a Folder

The following C# method shows how to add and delete a folder:

```
private static void AddAndDeleteFolder(Vaultworkspace workspace)
{
    // create a folder
    Folder newFolder = new Folder("newFolder");

    // add the folder to the list of the current user's favorite folders
    workspace.UserRepository.Add
        (newFolder, workspace.CurrentUser.Favorites);

    // display the favorite folders
    foreach (Folder folder in workspace.CurrentUser.Favorites)
    {
        Console.WriteLine("Folder title = " + folder.Title);
    }

    // display the AllowDeletes property
    // (this is true for the User Repository)
    Console.WriteLine("workspace.UserRepository.AllowDeletes = "
        + workspace.UserRepository.AllowDeletes);

    // if the folder is not null
    if (newFolder != null)
    {
        Console.WriteLine("Deleting the new folder");

        // delete the folder
        workspace.UserRepository.Delete(newFolder);
    }
}
```

Package Vault Objects in VOZIP Files

The class `Symyx.Framework.Vault.Packaging.VaultObjectPackage` provides the ability to package Vault objects in a Vault object package file known as a VOZIP file. VOZIP files have the extension `.vozip`.

Reasons for packaging Vault objects include:

- Packaging assemblies that are used in scripts. For information about using external assemblies with Workbook scripts, see [Script From External Assemblies](#).
- Packaging an application permission for deployment to multiple Vault servers.
- Creating a property set definition and then packaging it allows access to a subset of available properties in Property Set Definitions.

To create and populate a VaultObjectPackage object:

1. Call the `VaultObjectPackage.Create` static method to create a VOZIP file for the package.
2. Write to the `VaultObjectPackage` object in the same way as writing to a `Symyx.Framework.Vault.Repository` object.

Create a Vault Object Package Example

The following C# method creates a `VaultObjectPackage` object named `package`, and then adds a `VaultDictionary` object named `customSettings` to `package`.

```
private static void CreateVaultObjectPackage()
{
    // set the VOZIP file name
    string packageFilename = "C:\\myPackage.vozip";
    // create a vault object package
    using (VaultObjectPackage package = VaultObjectPackage.Create
        (packageFilename))
    {
        // create a vault dictionary
        VaultDictionary customSettings = new VaultDictionary
        { Title = "Custom Settings" };
        // for example purposes, add a single key and value to the
        // dictionary
        customSettings.Add("myKey", "myValue");
        // add the dictionary to the vault object package
        package.AddToRoot(customSettings);
        // close the vault object package package.Close();
    }
}
```

Enclosing the last set of lines within the `using` statement ensures that the `VaultObjectPackage` object is closed properly and the package contents are written to the VOZIP file.

Note: Do not use a plus (+) character or other special characters in package or profile names. If you do so, when a user whose account uses that package or profile attempts to log on, the system displays an message that says it cannot log them on to Vault due to an error on the server.

Add an Object to a Vault Object Package

The following C# method opens an existing Vault object package and adds a `VaultDictionary` object to the package.

```
private static void AddObjectToVaultObjectPackage()
{
    // set the VOZIP file name
    string packageFilename = "C:\\myPackage.vozip";
```

```
// open the vault object package
using (VaultObjectPackage package = VaultObjectPackage.Open
    (packageFilename))
{
    // create a dictionary object
    VaultDictionary customSettings = new VaultDictionary
        { Title = "Custom Settings" };
    // add a key and value to the dictionary object
    customSettings.Add("myKey", "myValue");
    // add the dictionary object to the vault object package
    package.AddToRoot(customSettings);
    // close the package package.Close();
}
}
```

List Objects in a Vault Object Package

The following C# method shows how to list the objects in an existing Vault object package:

```
private static void ListObjectsInVaultObjectPackage()
{
    // set the VOZIP file name
    string packageFilename = "C:\\myPackage.vozip";
    using (VaultObjectPackage package = VaultObjectPackage.Open
        (packageFilename))
    {
        // get the list of vault objects
        VaultObjectList list = package.Get
            (VaultObjectType.All, DataScope.Properties);
        foreach (VaultObject obj in list)
        {
            Console.WriteLine("Vault ID: " + obj.VaultId);
            Console.WriteLine("Vault object type: " + obj.Type);
            Console.WriteLine("Vault object title: " + obj.Title);
        }
        // close the package package.Close();
    }
}
```

Publish Using a VOZIP File

To publish your custom assembly, you can write a program that includes your assembly in a Vault object package (.vozip) file which can be distributed and published to Vault. In your program:

Call the `VaultObjectPackage.Create` method to create a `VaultObjectPackage` with the specified .vozip filename. `VaultObjectPackage` is in the `Symyx.Framework.Vault.Packaging` namespace in the `Symyx.Framework.dll` assembly.

Use the `Assembly.LoadFrom` method to load the assembly. `Assembly` is in the `System.Reflection` namespace.

Create a `VaultAssembly` object and populate it with the content of your custom assembly. `VaultAssembly` is in the `Symyx.Framework.Extensibility` namespace in the `Symyx.Framework.dll` assembly.

For example, the following C# method reads from `CompanyName.ProjectName.dll` and writes the contents to a .vozip file named `myPackage.vozip`:

```
private static void ReadDLLAndWriteContentsToVaultObjectPackage()
{
    // set the path and file name for the DLL assembly to be read from
    string assemblyFile "C:\\ProjectName\\CompanyName.ProjectName.dll";
    // set the path and file name for the VOZIP file to be written
    string outputFile = "C:\\myPackage.vozip";
    // create a VaultObjectPackage object using
    (VaultObjectPackage package = VaultObjectPackage.Create(outputFile))
    {
        // load the assembly
        Assembly loadFrom = Assembly.LoadFrom(assemblyFile);

        // create a vault assembly object and populate it with the content
        // read from the assembly
        VaultAssembly vaultAssembly = new VaultAssembly(loadFrom);

        // add the vault assembly object to the package, which writes the
        // package to the VOZIP file
        package.Add(vaultAssembly, package);

        // close the package
        package.Close();
    }
}
```

For more information on packaging Vault objects, including how to list the contents of a .vozipfile, see [Package Vault Objects in VOZIP files](#).

After creating the .vozip file, use Workbook to publish the file to Vault, see "Publish Vozip Files" in the BIOVIA Workbook online help.

Delete an Object From a Vault Object Package

The following C# method shows how to search an existing Vault object package for any Vault objects with the title, *Custom Settings*, and then delete the matching Vault objects.

```
private static void DeleteObjectFromVaultObjectPackage()
{
    // set the VOZIP file name
    string packageFilename = "C:\\myPackage.vozip";
    using (VaultObjectPackage package = VaultObjectPackage.Open
        (packageFilename))
    {
        // get the list of vault objects
        VaultObjectList list = package.Get
            (VaultObjectType.All, DataScope.Properties);
        foreach (VaultObject obj in list)
        {
            // if the object title equals "Custom Settings", delete the object
            if (obj.Title.Equals("Custom Settings"))
            {
                package.Delete(obj);
            }
        }
    }
    // close the package
}
```



```

    package.Close();
}
}

```

Read and Write to a Vault Object

A program can read the contents of an assembly DLL file and write the contents to a Vault object package. For example, the following C# method reads from `Symyx.Framework.dll` and writes the contents to a VOZIP file named `myPackage2.vozip`:

```

private static void ReadDLLAndWriteContentsToVaultObjectPackage()
{
    // set the path and file name for the DLL assembly to be read from
    string assemblyFile =
        "C:\\Program Files\\Symyx\\Symyx6.6\\lib\\Symyx.Framework.dll";
    // set the path and file name for the VOZIP file to be written string
    outputFile = "C:\\myPackage2.vozip";
    // create a VaultObjectPackage object using
    (VaultObjectPackage package = VaultObjectPackage.Create(outputFile))
    {
        // load the assembly
        Assembly loadFrom = Assembly.LoadFrom(assemblyFile);
        // create a Vault assembly object and populate it with the content
        // read from the assembly
        VaultAssembly vaultAssembly = new VaultAssembly(loadFrom);
        // add the Vault assembly object to the package, which writes the
        // package to the VOZIP file
        package.Add(vaultAssembly, package);
        // close the package package.Close();
    }
}

```

By default, when you add an assembly to a Vault object package, all of the assemblies on which the first assembly is dependent on will also be added. For example, if DLL1 is dependent on DLL2 and DLL3, then all three DLLs will be written to the Vault object package. To omit dependent assemblies, create an `AssemblyPublicationOptions` object and specify the assemblies you want to omit. For example:

```

AssemblyPublicationOptions publicationOptions = new
AssemblyPublicationOptions(Iterator.FromItems("Symyx.Framework.Controls",
"Symyx.Framework.Materials", "Symyx.Framework.Reporting"));

```

You then include `publicationOptions` in the `VaultObjectPackage.Create` method call nested within the using statement:

```

using (VaultObjectPackage package = VaultObjectPackage.Create(outputFile,
publicationOptions))

```

Display DLL Assembly Dependencies

One DLL assembly can depend on another. The following C# method opens a VOZIP file, checks whether a Vault object is an assembly, and displays the DLL dependencies.

```

private static void DisplayAssemblyDependencies()
{
    // open the Vault object package VOZIP file using
    (VaultObjectPackage package = VaultObjectPackage.Open
    ("C:\\myPackage2.vozip"))

```

```

{
// display column headings
Console.WriteLine(String.Format("{0,-24} {1,-54} {2}",
    "VaultObject type", "Vault ID", "Title"));
Console.WriteLine(String.Format("{0,-24} {1,-54} {2}",
    new String('-', 24), new String('-', 54), "-----"));
Console.WriteLine();
// display the details of the Vault objects in the package
foreach (VaultObject vaultObject in package.Get
    (VaultObjectTypes.All, DataScope.All, RetrievalOptions.None))
{
// if the vault object is an assembly, display the dependencies
if (vaultObject.ObjectType == VaultObjectType.Assembly)
{
// display the vault object details
Console.WriteLine(String.Format("{0,-24} {1,-54} {2}",
    vaultObject.ObjectType, vaultObject.VaultId,
    vaultObject.Title));
// display the dependencies
Console.WriteLine("Dependencies:");
foreach (Association association in vaultObject.Associations)
{
// if the association is a dependency
if (association.AssociationType == AssociationTypes.Dependency)
{
// get the dependency
VaultObject dependency = package.Get(association.TargetVaultId,
    DataScope.All, RetrievalOptions.SuppressNotFoundExceptions);
// if the dependency is not null
if (dependency != null)
{
// display the dependency
Console.WriteLine(String.Format("{0} - {1}",
    association.TargetUri, dependency.Title));
}
}
}
}
// close the package
package.Close()
}
}
}

```

The following IronPython example creates a Vault object package VOZIP file:

```

from Symyx.Framework.Vault.Packaging import VaultObjectPackage
package = VaultObjectPackage.Create("filename.vozip")
#can also use the full path
package.AddToRoot(document)
package.Close()

```

The next example creates folder hierarchies within the Vault object package:

```
from Symyx.Framework.Vault.Packaging import VaultObjectPackage from
Symyx.Framework.Vault import Folder
package = VaultObjectPackage.Create("filename.vozip") folder = Folder
("MyFolder")
package.AddToRoot(folder) package.Add(document, folder)
package.Close()
```

Chapter 3:

Users and Security

The `Symyx.Framework.User` namespace provides classes that support user settings and preferences such as:

- Favorite links and shortcuts
- Recent documents
- Vault objects

The `Symyx.Framework.Vault.Group UserProfile` class is a dictionary of user-specific settings and preferences. The class describes a group of Vault users assembled into a single unit for which security permissions are granted.

The `Symyx.Framework.Vault.User` class describes the server account of a user. The `IsActive` property indicates if the user is allowed to log in to a Vault server.

Classes Supporting Users Settings and Preferences

The `Symyx.Framework.User` namespace provides classes that support user settings and preferences such as user-defined lists of:

- Favorite links and shortcuts (Favorites)
- Recent documents (RecentDocumentsList)
- Vault objects (UserDefinedVaultObjectList)

The `UserProfile` class is a dictionary of user-specific settings and preferences.

The `Symyx.Framework.Vault.Group` class describes a group of Vault users or groups that are assembled into a single unit granted security permissions.

The `Symyx.Framework.Vault.User` class describes the server account of a Vault user. The `IsActive` property indicates if the user is allowed to log in to a Vault server.

Access User Profiles

A user's profile is encapsulated in the `Symyx.Framework.User.UserProfile` class. To access a user's profile, use the `UserProfile.Load` method. For example, in C#:

```
try
{
    myworkspace.Current.CurrentUser.Profile.Load();
}
catch(Exception ex)
{
    Debug.WriteLine("User profile could not be retrieved:");
    Debug.WriteLine(ex.ToString());
}
```

Permissions

The `Symyx.Framework.Vault.Security` namespace contains classes that encapsulate the permissions that users and groups have on Vault objects. The `Permissions` property of the

`Symyx.Framework.Vault.VaultObject` class returns the `Symyx.Framework.Vault.Security.ObjectPermissions` for that Vault object. The `Permissions` enumeration contains the different types of permissions for low-level operations that can be granted to a Vault object.

Multiple low-level permissions are aggregated into higher level privileges that are encapsulated in the `Privilege` class.

The following C# example returns false if a Vault object does not have the specified permissions:

```
if (vaultObject.Permissions != null)
{
    // Must have the checkout permission.
    if (!vaultObject.Permissions.HasPermission(Permissions.Checkout))
    { return false; }
    // Must have the write data permission.
    if (!vaultObject.Permissions.HasPermission(Permissions.WriteData))
    { return false; }
    // Must have the write properties permission.
    if (!vaultObject.Permissions.HasPermission
        (Permissions.UpdateProperties))
    { return false; }

    // Must have the transition permission.
    if (!vaultObject.Permissions.HasPermission
        (Permissions.WorkflowTransition))
    { return false; }
}
```

Explicit Permissions

Explicit permissions are permissions that are explicitly granted or denied. The `AllowPermissions` and `DenyPermissions` properties of the `Symyx.Framework.Vault.Security.ExplicitObjectPermissions` class returns the set of permissions flags that are granted or denied.

Note: Denied permissions take precedence over allowed permissions.

To set explicit permissions:

Use the `Symyx.Framework.Vault.VaultServer.SetExplicitPermissions` method.

The following C# example sets `ReadData` permissions to `user1` on the `Favorites` collection of the current user:

```
VaultObjectList users = workspace_.SiteRepository.Get
    (VaultObjectType.User, DataScope.Minimal);
string username = @"server\user1";
VaultObject user1 = users.FindByTitle(username);
workspace_.ActiveVaultServer.SetExplicitPermissions
    (user1.VaultId, workspace_.CurrentUser.Favorites.VaultId,
    workspace_.UserRepository.VaultId, "Favorites",
    Permissions.ReadData, Permissions.NoPermission);
```

Implicit Permissions

Implicit permissions are inherited from the user or repository hierarchies. For example, although a folder may have explicit permissions when it was created, the documents inside it and its subfolders have implicit permissions. The `Permissions` property of the `Symyx.Framework.Vault.Security.ImplicitObjectPermissions` class contains the inherited permission a user has to the object. The `GranteeGuid` property specifies the VaultID of the user or group associated with the permissions, and the `VaultObjectGuid` property specifies the VaultID of the repository to which the permissions apply.

The following C# example gets the implicit permissions of a user on a Vault object:

```
ImplicitObjectPermissions implicitPermissions = null;
implicitPermissions = workspace_.ActiveVaultServer.GetImplicitPermissions
(user1.VaultId, avaultobject.VaultId);
```

Application Permissions

The `Symyx.Framework.Vault.ApplicationPermission` class supports adding privileges to a user with respect to an application. The `ApplicationPermissions` property of the `Symyx.Framework.Vault.User` class returns a collection of `ApplicationPermission`, `ApplicationPermissionCollection` class, for a Vault user.

The following C# example shows how to check whether the current user is a template editor for the Workbook application:

```
bool isTemplateEditor =
    currentUser_.ApplicationPermissions.HasExecutePermissionFor
    ("Symyx.Notebook", "Template.Editor");
```

Chapter 4:

Properties

Workbook uses properties as key/value pairs to define characteristics and attributes of objects. A property set is a collection of properties, bound by a property set definition. A property set definition contains metadata and contents (data) of an object the user can save in the Vault server. This provides an easy way to create dynamic tables or grids of data and persist them in Vault. The schema of a table-based section in Workbook is defined by one or more property set definitions, so that each column in the table corresponds to a property in a property set definition.

The property set definitions are defined in Workbook. A user with the `PropertySetEditor` permission can create custom property set definitions and add the property set definition for use in table-based Workbook sections.

Property Set Editor

Workbook includes a Property Set Editor that provides the ability to interactively create and edit property set definitions.

You can add scripts for property event handlers that provide extension points for customization. The event handlers are listed under the Scripting category in the Property Set Editor. You can open the IronPython Script Editor from the event handlers.

If you copy an IronPython script from a text editor and copy the script into the IronPython Script Editor, make sure that your code uses straight double-quote characters in the IronPython Script editor. If your script contains smart quotes, the script fails.

In the Python programming language, leading white spaces or indentations at the beginning of each logical line are interpreted by the Python parser. If you copy and paste sample scripts from this document, verify that the indentations are correct in the pasted script.

Property Set Definitions

Property Set Definitions (PSDs) are reusable field definitions that specify columns and fields in table and form sections. Use the following setting values:

Cloneable property settings

Value	Description
Allowed	Indicates that cloning the data is allowed.
NotAllowed	Indicates that data is not cloned from the source experiment
AllowedNotData	Same as Allowed.

Limitation

- Clone to latest does not support a section upgrading to a newer version and renaming the section.
- The user cannot clone an unlinked reaction scheme section into a linked reaction scheme section.
- If the parent template had linked sections such as synthetic chemistry linked to parallel chemistry, the cannot clone to latest if the link is removed. In addition, after cloning to latest, the user must relink those sections.

Allow Nulls property settings

Value	Description
AllowNulls	Indicates that null values are permitted in the property.
CannotBeNull	Indicates that null values are not permitted; if the user tries to delete data and check in the document, the original data displays in the cell when the document is checked out.
ShouldNotBeNull	The interface renders a red X until user enters a value in the field.

All Updates

Value	Description
Always	Indicates that updates are permitted.
Never	Indicates that updates are not permitted.
Once	Indicates that the user can update the value one time only if the Experiment was cloned, otherwise updates are not permitted. The Once value reference is to the original data in the source experiment prior to saving.
Until Saved	Indicates that updates to the field are permitted until the experiment is saved.
Until Managed	Indicates that updates are permitted until the experiment is checked into a managed repository.

Property Event Handlers

A property has event handlers that allow reading or updating the value when certain events are triggered.

Event Handler	Description
CalculatedValueHandler	Executes when the value is requested by the client, that is, when a property is created or when a value is calculated.
OnCreatedHandler	Executes when the property is first created. For example, a script can set the initial value of the property.
ValueChangingHandlers	Executes before the value changes. You can use the <code>ValueChangingHandlers</code> to validate a value or prevent a user from entering an invalid value.
ValueChangedHandlers	Executes after the value has changed.

Property and Vault Object Variables

Variable name	Description
property	Identifies the <code>Symyx.Framework.Properties.Property</code> object.
properties	Identifies the <code>Symyx.Framework.Properties.PropertySet</code> object in which the property resides.
host_object	Identifies the object that owns the properties.
vault_object	Identifies the <code>Symyx.Framework.Vault.VaultObject</code> object, if <code>host_object</code> is a Vault object.
<type>	Specifies the type of Vault object, whose name follows the Python naming convention for variables, for example, <code>document_section</code> , <code>table_section</code> , <code>section</code> , <code>table</code> .

Property Class and PropertySetDefinition Variables

Variable name	Description
property_class	Identifies the <code>Symyx.Framework.Properties.PropertyClass</code> object.
property_set_definition	Identifies the <code>Symyx.Framework.Properties.PropertySetDefinition</code> object in which the property class resides.

Variable Aliases

Variable name	Description
section	Identifies the <code>Symyx.Notebook.DocumentSection</code> or a section from which the section was derived.
table	Identifies the <code>Symyx.Notebook.Sections.TableSection</code> .
row	Identifies the <code>Symyx.Framework.Properties.IPropertySetHost</code> representing a row in the table.
owner	Added by generic scripting and dynamic toolbars. Represents the object that has the property on which the script is defined. Other scope variables are not present for the owner, but might exist for the subject of the event.

Validation Script Variables for Value Changing Handlers

Scripts for the `ValueChangingHandler` are executed when a user changes a property value. These scripts are used for validation. If you are creating scripts using the Property Set Editor in Workbook, the following variables are available to Validation scripts:

Variable name	Description
e	Specifies the <code>Symyx.Framework.Properties.ValueChangingEventArgs</code> object allows access to: <ul style="list-style-type: none"> ■ <code>e.NewValue</code> - The new value of the property. You can only get this value. ■ <code>e.OldValue</code> - The old value of the property. You can only get this value. ■ <code>e.NewValueIsNull</code> - Indicates whether the new value is null. ■ <code>e.OldValueIsNull</code> - Indicates whether the old value is null. ■ <code>e.AddValidationResult(ValidationResults)</code> - Adds a list of <code>Symyx.Framework.Review.ValidationResult</code> objects containing a message and severity level. ■ <code>e.AddValidationResult(string, SeverityLevel)</code> - Adds a message and <code>Symyx.Framework.Review.SeverityLevel</code>. ■ <code>e.AddValidationResult(string)</code> - Adds a message to be displayed as an error. ■ <code>e.Cancel</code> - Set to true to cancel the property being updated. If set to True, the <code>CancelMessage</code> displays and the user must press the Escape key to cancel the action. ■ <code>e.CancelMessage</code> - A message to be displayed when canceling.
sender	The <code>Symyx.Framework.Properties.Property</code> object.
property	For more information, see Property and Vault Object Variables .
properties	
host_object	
<an alias of the host_object>	
parent	Indicates if the <code>host_object</code> has a parent such as document.

CalculateValueHandler Script Variables

You can use scripts for `CalculateValueHandler` for automatic value calculations or for assigning initial values. If you are creating scripts using the Property Set Editor in the Workbook, the following variables are available to Calculated Value and Initial Value scripts:

Variable name	Description
value	This variable is set to the return value, for example: <code>value = 12</code>

The Calculated Value and Initial Value scripts can use the variables in the `ValueChangingHandler` and Validation Scripts.

ValueSelectionsProvider Script Variables

Scripts for `ValueSelectionsProvider` provide the dictionary values that are listed in a list for a property. If you are creating scripts using the Property Set Editor in Workbook, the following variables are available to `DictionaryProvider` scripts.

Variable name	Description
<code>dictionary</code>	Specifies the <code>Symyx.Framework.Properties.PropertySet</code> bound to a property set definition. All the properties are defined from existing keyed values.

Property Dictionaries

A property dictionary is a list of values that user can select from when the property is rendered. A property dictionary can contain a list of one of the following:

- Static values
- Values generated by an IronPython script that uses the following construct to add values to the dictionary:

```
dictionary.Add("key", "value")
```

- Values from a Vault vocabulary

You can assign the dictionary options to a property using the Property Set Editor in Workbook. Using the Framework API, you can assign a dictionary provider to the `ValueSelectionsProvider` property of a `PropertyClass`.

If a property containing a dictionary is an Integer or Quantity, Value, Double, Decimal, Long Integer, Measurement, and String:

- The `AllowNulls` property is set to `CannotBeNull`.
- The Initial Value is not set.

Then the default value of the property will be 0 (zero) for the numeric types and zero-length string for the String type, even if 0 and the empty string are not an allowed value in the dictionary. If this is not an acceptable default value, create an `OnCreatedHandler` script that sets a meaningful `Initial Value`. For example, assuming the desired default value for a Quantity-type property is 0 mg, the following IronPython script sets the initial value:

```
from Symyx.Framework import Quantity,
    UnitKey value = Quantity(0, None, UnitKey.MILLIGRAM)
```

Dictionary Providers Types

The `PropertyClass.ValueSelectionsProvider` property contains the dictionary for a property. This dictionary implements the `IValueSelectionsProvider` interface and can be one of the following providers:

- `StaticValueSelectionsProvider` - contains a static list of values. The following C# example creates a static dictionary using `StaticValueSelectionsProvider`:

```
StaticValueSelectionsProvider staticProvider = new
    StaticValueSelectionsProvider();
staticProvider.Dictionary.Add("key1", "key1");
staticProvider.Dictionary.Add("key2", "key2");
```

```
// Assign the static dictionary provider to a property.  
// Assume aPropertyClass is already created.  
aPropertyClass.ValueSelectionsProvider = staticProvider;
```

- **ScriptedValueSelectionsProvider** - contains a list of generated values by an IronPython script.

The following C# example executes a simple IronPython script that builds a dictionary.

```
string[] script = new string[]  
{  
    "dictionary.Add(\"1.1\", 1.1)",  
    "dictionary.Add(\"1.2\", 1.2)",  
    "dictionary.Add(\"1.3\", 1.3)",  
};  
ScriptedValueSelectionsProvider scriptedProvider = new  
    ScriptedValueSelectionsProvider(string.Join("\n", script));  
// Assign the scripted dictionary provider to a property.  
// Assume aPropertyClass is already created.  
aPropertyClass.ValueSelectionsProvider = scriptedProvider;
```

- **VaultDictionaryValueSelectionsProvider** - contains a `Symyx.Framework.Vault.VaultUri` reference to a `Symyx.Framework.Vault.VaultDictionary` that contains a set of key/value pairs.

The following C# example creates a `VaultDictionary` for the dictionary provider:

```
// Create a Vault dictionary.  
VaultWorkspace workspace = new VaultWorkspace(endpoint);  
workspace.MakeCurrentWorkspace();  
VaultUri vaultUri = new VaultUri(new VaultId  
    (VaultIdPrefixes.Unknown), new Version(1));  
VaultDictionary dictionary = new VaultDictionary();  
dictionary.Add("selection", "value");  
// Create the Vault dictionary provider  
VaultDictionaryValueSelectionsProvider vaultProvider = new  
    VaultDictionaryValueSelectionsProvider(vaultUri);  
// Assign the Vault dictionary provider to a property.  
// Assume aPropertyClass is already created.  
aPropertyClass.ValueSelectionsProvider = vaultProvider;
```

Dictionary Scripts Examples

- **Adding values to a Request Column Dictionary**

The following IronPython example is a script for the `RequestColumnDictionary` that dynamically adds to the dictionary:

```
if not e.Items.Contains("orange"): e.Items.Add("orange")
```

- **Using data from another property in the same table**

The following IronPython example is a script for the `RequestColumnDictionary` that builds a dictionary for a `Name` property based on the value of a `Count` property:

```
if e.Property is not None and e.Property.Key == "Name": e.Items.Clear()  
    for i in range(e.Property.PropertySet["Count"].Value): e.Items.Add(str(i  
        + 1))
```

■ Using data from another table

The following IronPython example is a script for the RequestColumnDictionary that builds a dictionary for a Name property using the Name property in the Materials property set in the SourceTable section:

```
if e.Property is not None and e.Property.Key == "Name": for section in
Table.Document.Sections:
if section.Title == "SourceTable": target = section
break
else:
raise RuntimeError, "SourceTable section not found" e.Items.Clear()
for row in target.GetRows():
e.Items.Add(row.PropertySets["Material"]["Name"].DisplayValue)
```

■ Using data from an external database

The following IronPython example is a property script that retrieves a list of solvent class names from a database. The retrieved values are added to a dictionary for a string property, called *SolventClass*.

```
import sys import clr
clr.AddReference("System.Data") from System import *
from System.Data import *
from System.Data.OleDb import *
# Connect to the database and
# execute the query to get all solvent class names.
connection = OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\\solventdb.mdb;Persist Security Info=False")
query = "select distinct(Class) from solventdb" adapter =
OleDbDataAdapter(query, connection) solvents = DataSet()
adapter.Fill(solvents)
rows = solvents.Tables[0].Rows.Count
# Add each class name to the dictionary.
if solvents.Tables.Count > 0 and rows > 0 :
for i in range(rows):
dr = solvents.Tables[0].Rows[i] temp = dr["Class"].ToString()
dictionary.Add(temp, temp)
else :
value = " "
```

ValueSelectionsProvider Script Variables

Scripts for ValueSelectionsProvider are used to provide dictionary values that are listed in a drop-down list for a property. If you are creating scripts using the Property Set Editor in Workbook, the following variables are available to DictionaryProvider scripts:

■ dictionary

The `Symyx.Framework.Properties.PropertySet` that has been bound to a property set definition. All the properties have been defined from existing keyed values.

Property Dictionaries

A property dictionary is a list of values the end-user selects when the property is rendered. A property dictionary can contain a list of one of the following:

- Static values
- Values generated by an IronPython script that uses the following construct to add values to the dictionary:

```
dictionary.Add("key", "value")
```

- Values from a Vault vocabulary

You can assign the dictionary options to a property using the Property Set Editor in Workbook. Use the Framework API to assign a dictionary provider to the `ValueSelectionsProvider` property of a `PropertyClass`.

If a property containing a dictionary is an `Integer` or `Quantity`, `Value`, `Double`, `Decimal`, `Long Integer`, `Measurement`, and `String`, and its `AllowNulls` property is set to `CannotBeNull`, and the `Initial Value` is not set, then the default value of the property is zero for the numeric types and zero-length string for the `String` type, even if 0 and the empty string are not an allowed value in the dictionary. If zero is not an acceptable default value, create an `OnCreatedHandler` script that sets a meaningful `Initial Value`. For example, if the desired default value for a `Quantity`-type property is 0 mg, the following IronPython script sets the initial value:

```
from Symyx.Framework import Quantity,  
    UnitKey value = Quantity(0, None, UnitKey.MILLIGRAM)
```

Chapter 5:

Materials

Material Classes and Interfaces

The `Symyx.Framework.Materials` namespace, in the `Symyx.Framework.Materials.dll`, provides classes and interfaces that support:

- Representing of chemical materials
- Composing relationships among material objects
- Creating and transforming materials
- Developing of specialized types of materials used in chemical research and production

Material Class

The `Symyx.Framework.Materials.Material` class represents a material element or a mixture in a certain physical state. It can have a density, a molecular weight, and a collection of components.

You can prepare a material using an intensive mixture definition or an extensive mixture definition. An intensive mixture is created by combining substances whose quantities are not based on actual amounts. For example, the intensive mixture definition of brine is 50% water and 50% salt.

An extensive mixture is created by combining substances whose quantities are proportional to the actual amounts in a mixture, for example, an extensive mixture definition of brine might have 15 g salt and 20 ml water. `Symyx.Framework.Materials.MixtureType` is an enumeration that lists the ways in which the quantities of substances within a mixture is specified.

Material Properties

The `Material` class contains the following properties.

The properties are available to the material script variable that represents a `Material` object. For more information, see [Material Section Scripting](#).

Property	Description
CASNumber	Specifies a string containing the Chemical Abstracts Service Number (CAS#).
Comments	Specifies a string containing comments about the material.
Components	Specifies an <code>IEnumerable</code> containing the components, <code>Symyx.Framework.Materials.Component</code> objects, of a material.
Density	Specifies a <code>Quantity</code> object containing the density of the material. See Quantity Class .
DensityCalculation	Specifies a <code>DensityCalculation</code> object for the material. See DensityCalculation Class .
FormattedMolFormula	Specifies a <code>MolecularFormula</code> object containing a formatted molecular formula, including subscript and superscript annotations.
InitialAmount	Specifies a <code>Quantity</code> object containing the initial amount of the material.

Property	Description
MF	Specifies a string containing the molecular formula of the material.
MW	Specifies a Value object containing the molecular weight of the material. See Value Class .
Preparation	Specifies a Preparation object containing instructions on how to prepare the material. See Preparation Class .
PreparationID	Specifies a VaultUri object containing the preparation ID.
PS	Specifies a <code>Symyx.Framework.Properties.PropertySet</code> containing the property set for the Material object.
Role	Specifies a string specifying the role of the material.
Structure	Specifies a <code>Symyx.Framework.Chemistry.Structure</code> object containing the molecule structure in the material. The Structure class provides both the Molfile and Chime string formats in the Molfile and Chimestring properties. For example: <pre>molfile = material.Structure.Molfile chimestring = material.Structure.Chimestring</pre>

Material as a Mixture

Material are combined to create a mixture or concentration.

To create a mixture or concentration:

1. Create a Material object for each chemical in the mixture, and set its chemical properties. The following example IronPython script creates a Material object for two chemicals, Chem1 and Chem2, and sets their molecular weight ("MW") and Density properties:

```
Chem1 = Material("Chem1") Chem1.MW = Value(100)
Chem1.Density = Quantity(1, 4, UnitKey.GRAMPERCM3)
Chem2 = Material("Chem2")
Chem2.Density = Quantity(0.5000, 4, UnitKey.GRAMPERCM3)
Chem2.MW = Value(50)
```
2. Create a Material object for the mixture, and add a Component for each material to be included. The following example IronPython script creates a Material object for a mixture Chem1_2, and adds a Component for the Chem1 and Chem2 materials:

```
Chem1_2 = new Material("Chem1_2")
Comp1 = new Component(Chem1, new Quantity(12, 4, UnitKey.MOLELITER))
Comp2 = new Component(Chem2, new Quantity(0, UnitKey.REMAINDER))
Chem1_2.Components.Add(Comp1)
Chem1_2.Components.Add(Comp2)
```

Note: This example creates an excessively concentrated molar mixture. If you get the density of the mixture, `Chem1_2.Density.Number`, the result would be 1.2, which exceeds either density of the component materials.

The `Symyx.Framework.Materials.AmountHelper` provides static methods for amount calculations for material mixture and preparations. The calculation is done for conversions among mass, volume,

and mole units. The material density and molecule weight are the key properties that determine the calculation result. The material mixture can specify its mixing model for the calculation. The result of the calculation is stored in a `Symyx.Framework.Materials.CalculationResult` object.

DensityCalculation Class

The `Material` class contains a `DensityCalculation` property, which returns a `Symyx.Framework.Materials.DensityCalculation` object. It contains specifications for calculating density such as the preferred density unit, a correction factor for non-ideal mixing, and the type (`DensityType`) of calculation that identifies a mixture's density. The `Symyx.Framework.Materials.AmountHelper` class provides a static method for calculating the ideal density of a mixture.

The following example IronPython script (CalculatedValue script for an `IdealDensity` property of type `Quantity`) creates a mixture, assigns the density type and correction factor, and gets the density and ideal density of the mixture:

```
from Symyx.Framework.Materials import Material, Component,
    AmountHelper, DensityType
from Symyx.Framework import Quantity, UnitKey, Value

# Create two chemicals
Chem1 = Material("Chem1")
Chem1.Density = Quantity(1.000, 4, UnitKey.GRAMPERCM3)
Chem1.MW = Value(100)
Chem2 = Material("Chem2")
Chem2.Density = Quantity(0.8000, 4, UnitKey.GRAMPERCM3)
Chem2.MW = Value(80)

# Create 1M mixture called Chem1and2
Chem1and2 = Material("Chem1and2")
Comp1 = Component(Chem1, Quantity(1, 4, UnitKey.MOLELITER))
Comp2 = Component(Chem2, Quantity(0, UnitKey.REMAINDER))
Chem1and2.Components.Add(Comp1)
Chem1and2.Components.Add(Comp2)

# Assign a positive corrected (5%) density to Chem1and2
Chem1and2.Density = Quantity(0, UnitKey.GRAMPERCM3)
Chem1and2.DensityCalculation.Type = DensityType.Corrected
Chem1and2.DensityCalculation.CorrectionFactor = 0.05

# Check density info for Chem1and2
properties["DensityValue"].Value = Chem1and2.Density.Value
IdealDensity = AmountHelper.GetIdealDensity
    (Chem1and2.Components, UnitKey.GRAMPERCM3)
value = IdealDensity.Quantity
```

To run this script:

1. Login the Workbook client as a user with the `PropertySetEditor` permission.
2. Using the Property Set Editor, create a property set with the following properties:
 - ID as an Integer
 - DensityValue of type Value

- IdealDensity of type Quantity

3. On the property sheet for IdealDensity, add the script to its Scripting > Calculated Value property.
4. Select **View > Grid**.

When you enter a value for ID, the DensityValue and IdealDensity properties are automatically populated.

Materials Calculations

The Symyx.Framework namespace (in Symyx.Framework.Quantity.dll) provides classes that support specifying, measuring, and converting amounts of chemical and physical materials used in experiments.

Value class

The Symyx.Framework.Value class represents a decimal number and the number of significant figures to consider when rounding that decimal according to the active rounding rule. By default, the value is rounded up if the discarded fraction is equal or greater than .5; otherwise, the value is rounded down. Symyx.Framework.RoundingRules provides a static method SetRoundingRule which allows you to set the arithmetic rounding rule with Value instances. The Symyx.Framework.RoundingRuleType enumeration lists the possible rounding techniques.

The Value.Value property contains the number, and the Value.SigFigs property contains the number of significant figures.

Value supports arithmetic operations carrying significant figures. Note that batch calculations of values are not as efficient as operations performed by the native .NET calculation objects.

You cannot convert a double number into a Value. For example, you cannot use the constructor Value(9.99) where 9.99 is of type Double.

To convert a Double to a Value object, convert the Double number to a string, and use the Value.Parse method. For example, the following IronPython script gets the property value of type Double ("DoubleProperty"), converts it to a string, parses it, then puts it into another property of type Value ("ValueProperty"):

```
from Symyx.Framework import Value
```

```
doubleString = properties["DoubleProperty"].Value.ToString()
properties["ValueProperty"].Value = Value.Parse(doubleString)
```

Note: The Value.Parse method is sensitive to the current regional settings. If you use this method with numeric literals, note that the regional settings will affect the parsed value. For example, if the regional settings are in French, the decimal point is ignored in Value.Parse("222.332"). This does not happen if you parse user input values.

Unit class

The Symyx.Framework.Unit class represents a unit of measure of chemical or physical quantity. A Unit contains a SymyxID which corresponds to a key in Symyx.Framework.UnitKey. UnitKey is an enumeration that lists the units of most common chemical and physical quantities.

The Symyx.Framework.Dimensions class describes the dimensions of a chemical or physical unit. It contains properties that return the power of various dimensions, such as ElectricCurrent, Length, Mass, Mole, ThermodynamicTemperature, Time, and others. Dimensions contains a SymyxID that corresponds to a key in Symyx.Framework.DimensionsKey. DimensionsKey is an enumeration that lists the dimensions of chemical and physical units. One dimension can have multiple units to one dimension; units are stored in the Dimensions.Units property.

The Symyx.Framework.UnitCategory class represents a category or grouping of units of chemical or physical quantity. UnitCategory contains a SymyxID that corresponds to a key in

`Symyx.Framework.UnitCategoryKey`. `UnitCategoryKey` is an enumeration that lists the categories of chemical and physical units. An object property can have multiple `UnitCategories`.

The `Symyx.Framework.UnitsHelper` class allows clients to perform operations on units including the creation of a unit with specified key or name; the conversion of amounts expressed in one unit into amounts expressed in another unit in the same category; the list of unit categories; and the dimensions of a unit. Unit keys are backward compatible with LEA applications. See “Converting amounts” below.

Quantity class

The `Symyx.Framework.Quantity` class represents a value plus a unit. It includes methods to construct a quantity from various data, compare quantities, round values, and cast the quantity into a string. The `Quantity.Number` property contains the decimal value of the quantity.

A `Quantity` stores a `Value` and a `UnitKey`. The `UnitKeys` are defined as an enum for `Symyx.Framework.Quantity`. Units are grouped by `Dimensions` and by `UnitCategory`.

Conversions

Conversions are possible among units of the same dimension wherever that is well defined. However, it is not possible to perform conversions among units of currency.

`UnitCategories` are groups of units related by use, rather than dimensional analysis. You can define new categories. An example of category is `Concentration`. This groups diverse units of molarity, density, mass, or volume fraction. There are no conversions among units of a category because they might belong to different dimensions.

`Quantity` is a value type, not a reference type. `Quantity` is indexed into two RAS fields: a `Number` and a `Unit`. These unit keys are the same as the RAS values. RAS does not support unit conversion during query.

Conversions are handled by the `UnitsHelper` object. Conversions do not modify the original value and unit.

Limitations and tips

- Calculations on `Quantity` objects are not supported, but operations on `Quantity.Number` are supported.
- A `Double` number cannot be converted into a `Quantity`. For example, you cannot use the constructor `Quantity(9.99)` where 9.99 is of type `Double`.

To convert a `Double` to a `Quantity` object, convert the `Double` number to a string, and use the `Quantity.Parse` method.

For example, the following IronPython script gets the property value of type `Double` (`"DoubleProperty"`), converts it to a string, parses it, then puts it into another property of type `Quantity` (`"QuantityProperty"`):

```
from Symyx.Framework import Quantity
doubleString = properties["DoubleProperty"].value.ToString()
properties["QuantityProperty"].value = Quantity.Parse(doubleString)
```

The `Quantity.Parse` method is sensitive to the current regional settings. If you use this method with numeric literals, the regional settings affect the parsed value. For example, if the regional settings are in French, the decimal point is ignored in `Quantity.Parse("222.332")`. This does not happen if you parse user input values.

The `Quantity` constructor is overloaded and can accept various numeric data types. See the `Quantity` constructor in the `Symyx.Framework` namespace section of the Framework API Reference.

Calculating Molecular Weight Example

The following IronPython example calculates the molecular weight (MW) based on a structure. In this example, MW is of type Quantity. The script calls Symyx Cheshire to get the weight of the structure, creates a Value using the weight and number of significant figures, and assigns it to the Quantity.Value property of MW:

```
# Calculates molweight from a structure
import Symyx.Framework.Quantity from MDL.Cheshire import Cheshire
# Create a Cheshire environment
cheshire = Cheshire()
try:
    # if structure has been entered...
    if not e.NewValueIsNull:
        # if MW has been calculated in this editor session...
        if not 'oldMW' in locals():
            # if the old structure value was set...
            if not e.OldValue == None:

cheshire.SetTarget(e.OldValue.Molfile)
cheshire.RunScript('w = weight()')
mwFromOldStruct = cheshire.UnloadVariable('w')
sfFromOldStruct = \
Symyx.Framework.Value.GetApparentSigFigs(mwFromOldStruct)

# if MW is same as previous structure,
# set oldMW var to the old structure's MW
if properties['MW'].Value == \
Symyx.Framework.Value(mwFromOldStruct, sfFromOldStruct):
    oldMW = properties['MW'].Value
else:
    oldMW = None
else:
    oldMW = None
# if MW is null or previously calculated MW is equal to current MW,
# calculate and set a new MW
if properties['MW'].IsNull or (oldMW == properties['MW'].Value):
    cheshire.SetTarget(e.NewValue.Molfile)
    cheshire.RunScript('w = weight()') mw = cheshire.UnloadVariable('w')
    sf = Symyx.Framework.Value.GetApparentSigFigs(mw) properties['MW'].Value =
    Symyx.Framework.Value(mw, sf)
    oldMW = properties['MW'].Value

finally:
    if not cheshire is None:
cheshire.Dispose()
```

To run this script:

1. Login to Workbook as a user with the PropertySetEditor permission.
2. Using the Property Set Editor, create a property set with at least the following properties:
 - Structure of type Structure
 - MW of type Quantity

3. On the property sheet for Structure, add the script to its **Scripting > Property Changed** property.
4. Select **View > Grid**. When you enter a value for Structure, the MW property is automatically populated. For more information about Cheshire scripts, see the Cheshire developer documentation.

Materials Sections C# Example

The Materials Section is derived from the Table Section. The following C# example lists the PropertySetDefinitions (PSDs) of a table section.

```
// setting up the test case
TableSection ts = new TableSection();
ICollection<PropertySetIdentifier> selectedPSDs =
    ts.TableSectionProperties.GetValue
    <ICollection<PropertySetIdentifier>>
    (TableSectionP roperty.SelectedPropertySetDefinitons);
selectedPSDs.Add(new PropertySetIdentifier("Test"));
selectedPSDs.Add(new PropertySetIdentifier("Test2"));

// listing PSDs without knowing anything about them
foreach(PropertySetIdentifier identifier in selectedPSDs)
{
    // identifier can identify a PSD by its Key, unique within Vault Server
    // or by VaultUri to the exact version of the PSD
    // this is the selected PSD Key identifier.Key PropertySetDefinition
    aSelectedPropertySetDef = PropertySetManager.GetDefinition
    (identifier.Key);
    Debug.WriteLine("The table has columns corresponding to this PSD: "
        + aSelectedPropertySetDef.Key);
    foreach(PropertyClass propertyClass in aSelectedPropertySetDef)
    {
        Debug.WriteLine("The table has a column corresponding to this
            PropertyClass: " + propertyClass.DisplayName);
    }
}
```

The following example shows how to add a row to a table, list the PSDs for a table section, clear the table, and delete a specific row.

```
public void ListRowsInATable()
{
    // setting up the test case TableSection
    ts = new TableSection();
    ICollection<PropertySetIdentifier> selectedPSDs =
        ts.TableSectionProperties.GetValue
        <ICollection<PropertySetIdentifier>>
        (TableSectionProperty.SelectedPropertySetDefinitons);
    selectedPSDs.Add(new PropertySetIdentifier("Test"));
    selectedPSDs.Add(new PropertySetIdentifier("Test2"));
    // a Row is a VaultObject, and as such it implements IPropertySetHost
    IPropertySetHost r = ts.AddRow();
    r.PropertySets["Test"]["Key"].Value = "Hello1";
    r.PropertySets["Test2"]["Key0"].Value = "Hello2";
    r.PropertySets["Test2"]["Key1"].Value = "Hello3";
    IPropertySetHost r2 = ts.AddRow();
}
```

```

r2.PropertySets["Test"]["Key"].Value = "2Hello1";
r2.PropertySets["Test2"]["Key0"].Value = "2Hello2";
r2.PropertySets["Test2"]["Key1"].Value = "2Hello3";
foreach(IPropertySetHost host in ts.Rows)
{
    foreach(PropertySetDefinition psd in host.PropertySetDefinitions)
    {
        foreach(PropertyClass pc in psd)
        {
            Debug.WriteLine(host.PropertySets[psd.Key][pc.Key].DisplayValue);
        }
    }
}
// to clear the table
ts.Clear();
// to delete a specific row by Id
ts.Delete(r.Id);
}

```

Measurement Class

The `Symyx.Framework.Measurement` class describes the measurement of a quantity, which consists of a `Value` and a `Unit`. The `Measurement.ActualAmount` property returns the `Quantity` containing the amount. The `Measurement.Detail` property is an implementation of `IMeasurementDetail` that allows additional data to be attached to the measurement, such as measurement type (`MeasurementType.Manual` or `MeasurementType.Automatic`), description, and timestamp.

The following IronPython example is an Initial Value script for a Measurement Property. The script creates an automated measurement value:

```

from Symyx.Framework import Measurement, MeasurementType, Quantity,
    UnitKey from Symyx.Framework.Properties import
    SimpleMeasurementDetail
value = Measurement(Quantity(10.0, 3, UnitKey.GRAM),\
    SimpleMeasurementDetail(MeasurementType.Automated, None,
        "Automated value"))

```

To run this script:

1. Login the Workbook as a user with the `PropertySetEditor` permission.
2. Using the Property Set Editor, create a property set with at least the following properties:
 - ID of type Integer
 - TempWeight of type Measurement
3. On the property sheet for TempWeight, add the script to its **Scripting > Initial Value** property.
4. Select **View > Grid**. When you enter a value for ID, the TempWeight property will be automatically populated.

Validate a Measurement

The Framework provides a script the capability to report validation errors to the user. When validating measurement amounts, a script writer can use the `e.AddValidationResult` script variable to display a message and indicate a severity level using `Symyx.Framework.Review.SeverityLevel`.

The following IronPython example is a validation script for a measurement property. The script evaluates the input number and displays a message.

```
from Symyx.Framework.Review import SeverityLevel
# e.NewValue is a Measurement
number = e.NewValue.ActualAmount.Number
if number < 1.5:
    # SeverityLevel not specified; default is SeverityLevel.Error.
    e.AddValidationResult("Value less than 1.5 is an error")
elif number > 7.0:
    e.AddValidationResult("Value is okay, but too high.",\
SeverityLevel.Information)
elif number > 5.5:
    e.AddValidationResult("Value is within warning range.",\
SeverityLevel.Warning)
```

To run this script:

1. Log in the Workbook as a user with the `PropertySetEditor` permission.
2. Using the Property Set Editor, create a property set with at least the following properties:
 - ID of type Integer
 - IdealValue of type Measurement
3. On the property sheet for IdealValue, add the script to its Scripting > Validations property.
4. Select **View > Grid**.
5. Enter values in **IdealValue** that might produce the validation results from the script.

If a validation requires the user to cancel an entry, you can use `e.CancelMessage`. The following example uses `e.CancelMessage` to prompt the user to cancel an entry based on validation checks:

```
if not e.NewValueIsNull:
    if e.NewValue.Number > 1:
        e.Cancel = True
        e.CancelMessage = 'The Correction Factor cannot be greater than 1'
    if e.NewValue.Number <= 0:
        e.Cancel = True
        e.CancelMessage = 'The Correction Factor must be strictly positive.'
```

To run this script:

1. Log in the Workbook as a user with the `PropertySetEditor` permission.
2. Using the Property Set Editor, create a property set with the following property:
 - CheckValue of type Value
3. On the property sheet for CheckValue, add the script to its Scripting > Validations property.
4. Select **View > Grid**.
5. Enter values in **CheckValue** that might produce the validation results from the script.

Convert Amounts

The `Symyx.Framework.UnitsHelper` class provides static, `Convert` methods that convert a double, decimal, value, or quantity from one unit to another. The source and target units must both belong to the same `UnitCategory`. The significant figures of the amount in the target unit are the same as those of the amount in the source unit, the one exception occurs with a temperature conversion, where an offset might change the significant figures.

When performing conversions and calculations, always check the input calculation values for nulls or zero.

The following IronPython example is a `CalculatedValue` script for an actual concentration quantity. The script converts a measurement (*Amount*) to milligrams, converts a quantity (*TotalVolume*) to milliliters, constructs a value with a calculated concentration number, and constructs a quantity with the calculated concentration in milligrams-per-milliliter.

```
from Symyx.Framework import Quantity, UnitKey, UnitsHelper, Value
# Before calculation, always check that input values are valid.
if not properties["Amount"].IsNull and\
not properties["TotalVolume"].IsNull and\ properties
["TotalVolume"].Value.Number > 0.0:
# Convert input Measurement.ActualAmount Quantity to milligram
actualMg = UnitsHelper.Convert(\
properties["Amount"].Value.ActualAmount, UnitKey.MILLIGRAM)
# Convert input Quantity to milliliter
volumeMl = UnitsHelper.Convert(\
properties["TotalVolume"].Value, UnitKey.MILLILITER)
# Create a Value by dividing decimal numbers of actualMg and volumeMl
valueConcentration = Value(actualMg.Number / volumeMl.Number)
# Create a Quantity using the calculated concentration
# and milligrams per milliliter unit.
value = Quantity(valueConcentration, UnitKey.MGPERML)
else:
value = None
```

To run this script:

1. Log in to the Workbook as a user with the `PropertySetEditor` permission.
2. Using the Property Set Editor, create a property set with the following properties.
 - Amount of type Measurement
 - Set the Default Unit to milligrams (mg)
 - UnitCategories to mass
 - TotalVolume of type Quantity
 - Set the Default Unit to liters (L)
 - UnitCategories to volume.
 - ActualConcentration of type Quantity
3. Leave Default Unit and UnitCategories as undefined.
4. On the property sheet for ActualConcentration, add the script to its Scripting > Calculated Value property.
5. Select **View > Grid**.

When you enter values in Amount and TotalVolume. The script should automatically populate ActualConcentration with the converted amount.

The following IronPython example is a `CalculatedValueHandler` script on a property of type Long Integer. The script converts an Integer to a Long Integer.

```
from System import Int64
MyValue = properties['MyInteger'].Value iVal = -1
if MyValue >= 0: try:
iVal = MyValue + 12345 except:
```



```

ival = -99
else:
    ival = 999
value = Int64.Parse(ival.ToString())

```

To run this script:

1. Log in to the Workbook as a user with the `PropertySetEditor` permission.
2. Using the Property Set Editor, create a property set with at least the following properties:
 - `MyInteger` of type `Integer`
 - `MyLongInteger` of type `Long Integer`
3. On the property sheet for `MyLongInteger`, add the script to its `Scripting > Calculated Value` property.
4. Select **View > Grid**.

When you enter values in *MyInteger* the script should automatically populate *MyLongInteger*.

Container Class

The `Symyx.Framework.Materials.Container` class describes the object that physically contains a material. The `Container.Material` property represents the `Material` that the `Container` holds. The `Container.Capacity` specifies a quantity indicating how much material it can hold. The `Container.Amount` property specifies a quantity of how much material it actually holds.

Preparation Class

The `Symyx.Framework.Materials.Preparation` class is used to record an actual execution of producing a material. You can record actual amounts and actual process conditions in the `Preparation` class. The `Preparation.ActionTargets` property is a collection of `Symyx.Framework.Materials.ActionTarget` objects, representing a conceptual entity that can associate with a container, a piece of equipment, or an instrument.

For example, two material charges can reference the same action target to say they are made in the same flask or vial. A material transfer can occur by taking this material from one action target to the other. The `Preparation.Procedure` property contains a `Symyx.Framework.Materials.PreparationStepGroup` that has indexed `ParameterStep` children describing each step in the preparation procedure.

The following C# example creates a preparation with two water charges. Its input parameter is a `water Material` object.

```

public static Preparation CreatePreparationWith2WaterCharges
    (Material water)
{
    Preparation target = new Preparation();
    IEnumerable<PreparationStep> enumerator =
        target.Procedure.GetChildren(null);
    foreach (PreparationStep psb in enumerator)
    {
        Assert.IsNotNull(psb);
    }
    // Create a vial ActionTarget.
    ActionTarget vial = new ActionTarget();

```

```
// Add vial to the Preparation.
target.ActionTargets.Add(vial);
// Get the PreparationStepGroup.
PreparationStepGroup topStep = target.Procedure;
// Create the first water material charge.
MaterialCharge materialCharge = new MaterialCharge();
materialCharge.ActionTarget = vial;
materialCharge.PlannedAmount = new Quantity(1M, UnitKey.MILLILITER);
materialCharge.SetMaterial(water);
// Insert the first material charge to the PreparationStepGroup
topStep.InsertChild(0, materialCharge);
// Create the second water material charge.
materialCharge = new MaterialCharge();
materialCharge.ActionTarget = vial;
materialCharge.PlannedAmount = new Quantity(200M, UnitKey.MICROLITER);
materialCharge.SetMaterial(water);
// Insert the second material charge to the PreparationStepGroup
topStep.InsertChild(1, materialCharge);
return target;
}
```

Chapter 6:

Documents

A document is an instantiation of a user-specified document (experiment) template. A user creates a new document, by selecting a document template in the Vault repository browser.

The `Symyx.Notebook.Document` class is a collection of sections (container). You can :

- Enumerate and access the `Symyx.Notebook.DocumentSection` objects that make up a document or template through the `Document.Sections` property.
- Access the document template used to create a document through the `Document.Template` property.
- Access the section template used to create a document section through the `DocumentSection.Template` property.

Document Sections

A document section is the building block of a document, a given document instance is composed of a collection of document sections.

A section template defines how each document section is created. A document template defines the specific set of section templates that are used to create document of a specific type.

Document sections represent a primary extension point for customized Workbook application behavior. Document template authors can create custom document types by referencing:

- Generic document section implementations such a section used to display images or free-form text, and or more complex section such as a data entry form.
- Domain-specific document sections such as a document created to meet the needs of analytical chemistry, a preparation section for information related to formulations.

The `Symyx.Notebook.DocumentSection` class is an abstract base class for document section implementations. Concrete document section implementations are directly or indirectly derived from `Symyx.Notebook.DocumentSection`.

`Symyx.Notebook.DocumentSection` is derived from a `VaultElement` that is derived from a `VaultObject` which enables storing and managing each `DocumentSection` instance in Vault.

A `DocumentSection` implementation can implement the interfaces that correspond to the activities it supports. For example, the default form section supports edit, report, and indexing activities, and can incorporate a custom data entry form as a section in a document.

Document Sections Types

A document can contain any of the sections described in the following table.

Section Type	Description
TextSection	Renders a text editor that allows a user to enter and format text.
FileSection	Renders files such as PDF and image files that can be inserted by a user.

Section Type	Description
FormsSection	Contains widgets such as a text box, label, button, check box, and combo boxes that a user with using the FormEditor. A form allows a user to view and enter information for a single logical unit of data.
SpreadsheetSection	Contains an Excel spreadsheet.
TableSection	<p>Renders a grid whose columns are defined by a property set definition. A table section enables viewing and entering multiple records of data in a grid format.</p> <p>The following sections are derived from the TableSection:</p> <ul style="list-style-type: none"> ■ Materials Section used for entering information about a set of materials. ■ Equipment Section used for entering information about equipments. ■ Preparation Section used for entering information about materials, their components, and their order of usage in an experiment. ■ Synthetic Chemistry Section provides rows that represent a material that participates in a reaction. ■ Formulations Section, similar to a Preparations section, but used for formulations. ■ Formulation Materials Section, similar to a Materials section, but with additional properties.
ReactionSchemeSection	Provides a workspace that can contain reaction steps that can contain a reaction.
ReferenceSection	Contains a list of reference to one or more Workbook sections.

Find a Template Example

The following IronPython example gets the DocumentSectionTemplate for the file section:

```
SectionTemplates = active_workspace.Current.SiteRepository.Get
    (VaultObjectTypes.DocumentSectionTemplate, DataScope.All)
for SectionTemplate in SectionTemplates:
    if SectionTemplate.Title == "File" : neededSection = SectionTemplate
```

Create a Document from a Template

This C# example of creates a document from a template and assigns a title:

```
using Symyx.Framework.Vault;
using Symyx.Notebook;

namespace Symyx.Notebook.Examples
{
    public class DocumentExamples
    {
        public static Document CreateDocumentFromTemplate
            (IRepository repository, VaultId templateVaultId)
        {
```

```

        Document;
    }
    var template = repository.Get(templateVaultId, DataScope.All) as
    var document = Document.Create(template); document.Title = "Example
experiment"; return document;
    }
}

```

Get a Document

A document is VaultObject that belongs to the Vault repository. To get a document, use the Get method on the Vault repository object, specifying that the data scope is all the sections in the document. The Get method returns a VaultObject that you cast as a Document.

```
myDoc = (Document)Vaultworkspace.Get(VaultURI, DataScope.All);
```

or

```
myDoc = (Document)Vaultworkspace.Get(VaultID, DataScope.All);
```

Note: There is also the `Symyx.Notebook.Document.Find` method, which is overloaded to find by Vault ID, VaultUri, Predicate, or String title.

Adding, Inserting, and Removing Document Sections

Whereas the add operation appends the section to the end of the ordered list of sections, the insert operation places a section at a specific, indexed location. The following C# example adds a section and assigns it a name.

```

using System;
using System.Collections.Generic; using System.Linq;
using System.Text;
using Symyx.Framework.Vault; using Symyx.Notebook;
using Symyx.Notebook.Sections.Text;

```

```

namespace Symyx.Notebook.Examples
{
    class DocumentExamples
    {
        public static Document CreateDocumentFromTemplate
            (IRepository repository, VaultId templateVaultId)
        {
            var template = repository.Get(templateVaultId, DataScope.All)
            as Document;
            var template = repository.Get(templateVaultId, DataScope.All)
            as var document = Document.Create(template);
            var textSection = new TextSection();
            // Assign a name
            textSection.Title = "myTextSection";
            // Add the new text section to the document
            document.Add(textSection);
            return document;
        }
    }
}

```

```
}  
}
```

The following References were added: `Symyx.Framework`, `Symyx.Notebook`, `Symyx.Notebook.Sections.Text`, `Symyx.Notebook.Sections.Text.Extensibility`, `System`, `System.Core`, `System.Data`, `System.Data.DataSetExtensions`, `System.Xml`, `System.Xml.Linq`.

Insert a Section Between Other Sections

To insert a section into a Document at a specific location, use the `Symyx.Notebook.Document.Insert` method with the following signature:

```
public void Insert(int index, DocumentSection item)
```

Remove a Section

You can remove a section from the document using the `Symyx.Notebook.Document.Remove` method and enable the removal based on the `VaultElement`, `VaultId`, `VaultObject`, or `DocumentSection`.

Text Sections Example

This C# example illustrates adding a text section to a document, and setting the `PlainText` property of the `TextDocument` in `Workbook`.

Note: If you use this example in other applications separate licensing is required for the `TXTextControl` and for `Keyoti RapidSpell.NET`.

```
using System;  
using System.Collections.Generic; using System.Diagnostics;  
using System.Linq; using System.Text;  
using Symyx.Notebook.Sections.Text;  
  
namespace Symyx.Notebook.Examples  
{  
    public static class TextDocumentExamples  
    {  
        private static Document CreateSampleDocumentContainingTextSection()  
        {  
            Document template = new Document();  
  
            // Create a document from the template Document  
            document = Document.Create(template);  
  
            // Add a new text section to the document  
            document.Add(new TextSection());  
            return document;  
        }  
  
        // Sets the RichText property of a TextDocument  
        public static TextDocument SetTextDocumentRichText()  
        {  
            Document document = CreateSampleDocumentContainingTextSection();
```

```

TextSection textSection = (TextSection) document[0];
textSection.Text.RichText =
    @"{\rtf1\ansi\ansicpg1252\deff0\deflang1033
      {\fonttbl{\f0\fswiss\fcharset0 Arial;}}"
    + "\r\n"
    + @"{\colortbl ;\red255\green0\blue0;}"
    + "\r\n"
    + @"{\*\generator Msftedit 5.41.15.1515;}
    \viewkind4\uc1\pard\b\f0\fs20 Sample\cf1\u1\b0
    text\cf0\u1none\par"
    + "\r\n"
    + @"}" + "\r\n";

Debug.Assert(textSection.Text.PlainText == "Sample text\r\n");

return textSection.Text;
}

// Sets the PlainText property of a TextDocument
public static TextDocument SetTextDocumentPlainText()
{
    Document document = CreateSampleDocumentContainingTextSection();
    TextSection textSection = (TextSection) document[0];
    textSection.Text.PlainText = "Sample text";
    Debug.Assert(textSection.Text.PlainText == "Sample text");
    return textSection.Text;
}
}
}

```

Add Text Sections and Set Plain Text

This C# example illustrates adding a text section to a document, and setting the PlainText property of the TextDocument. This illustration assumes use in Workbook. Otherwise, separate licensing is required for separate licensing is required for the TXTextControl and for Keyoti RapidSpell.NET.

```

using System;
using System.Collections.Generic; using System.Diagnostics;
using System.Linq; using System.Text;
using Symyx.Notebook.Sections.Text;
namespace Symyx.Notebook.Examples
{
    public static class TextDocumentExamples
    {
        private static Document CreateSampleDocumentContainingTextSection()
        {
            Document template = new Document();
            // Create a document from the template Document
            document = Document.Create(template);
            // Add a new text section to the document
            document.Add(new TextSection());
            return document;
        }
    }
}

```

```
// Sets the RichText property of a TextDocument
public static TextDocument SetTextDocumentRichText()
{
    Document document = CreateSampleDocumentContainingTextSection();
    TextSection textSection = (TextSection) document[0];
    textSection.Text.RichText =
        @"{\r-f1\ansi\ansicpg1252\deff0\deflang1033
        {\fonttbl{\f0\fswiss\fcharset0 Arial;}}"
        + "\r\n"
        + @"{\colortbl ;\red255\green0\blue0;}"
        + "\r\n"
        + @"{\*\generator Msftedit 5.41.15.1515;}
        \viewkind4\uc1\pard\b\f0\fs20 Sample\cf1\u1\b0
        text\cf0\u1none\par"
        + "\r\n"
        + @"}"
        + "\r\n";
    Debug.Assert(textSection.Text.PlainText == "Sample text\r\n");
    return textSection.Text;
}
// Sets the PlainText property of a TextDocument
public static TextDocument SetTextDocumentPlainText()
{
    Document document = CreateSampleDocumentContainingTextSection();
    TextSection textSection = (TextSection) document[0];
    textSection.Text.PlainText = "Sample text";
    Debug.Assert(textSection.Text.PlainText == "Sample text");
    return textSection.Text;
}
}
```

Add Data to a Text Section

The following C# snippets are from `Symyx.SDK.Samples.DataCreation.Program.cs` available in the samples folder of the SDK installation.

```
using Symyx.Notebook.Sections.Text;
...
AddNewTextSectionWithVariedRtf(document);
AddNewTextSectionWithRtfContainingImages(document);
AddNewTextSectionWithSamplePlainText(document);
AddNewTextSectionWithImage(document);
```

Additionally:

```
AddNewTextSectionWithStructure(document);
AddNewTextSectionWithTextAndStructure(document);

public static TextSection AddNewTextSection
    (Document document, string sectionTitle)
{
    var textSection = new TextSection();
    textSection.Title = sectionTitle; document.Add(textSection);
}
```



```
    return textSection;
}

public static TextSection AddNewTextSectionWithRtf
    (Document document, string sectionTitle, string sectionRtfContent)
{
    var textSection = AddNewTextSection(document, sectionTitle);
    textSection.Text.RichText = sectionRtfContent;
    return textSection;
}

public static TextSection AddNewTextSectionWithPlainText
    (Document document, string sectionTitle, string
        sectionPlainTextContent)
{
    var textSection = AddNewTextSection(document, sectionTitle);
    textSection.Text.PlainText = sectionPlainTextContent;
    return textSection;
}

public static TextSection AddNewTextSectionWithImage
    (Document document, string sectionTitle, string imageFilePath)
{
    var textSection = AddNewTextSection(document, sectionTitle);
    textSection.Text.Images.Add(imageFilePath);
    return textSection;
}

public static TextSection AddNewTextSectionWithStructure
    (Document document, string sectionTitle, string imageFilePath,
        Structure structure)
{
    var textSection = AddNewTextSection(document, sectionTitle);
    textSection.Text.Images.Add(imageFilePath, structure);
    return textSection;
}

public static TextSection AddNewTextSectionWithPlainTextAndStructure
    (Document document, string sectionTitle, string
        sectionPlainTextContent, string imageFilePath, Structure structure,
        int imagePosition)
{
    var textSection = AddNewTextSectionWithPlainText
        (document, sectionTitle, sectionPlainTextContent);
    textSection.Text.Selection.Start = imagePosition;
    textSection.Text.Images.Add(imageFilePath, structure);
    return textSection;
}
}
```

Check the Section Type

The `Symyx.Notebook.Sections.Text.TextSection` class implements the `Symyx.Notebook.Sections.Text.Extensibility.ITextSection` interface (which is found in `Symyx.Notebook.Sections.Text.Extensibility.dll`). You can use `ITextSection` to determine whether a section is a Text section. For example, in C#:

```
foreach (var section in document)
{
    if (section is ITextSection)
    {
        ((ITextSection)section).TextDocument.PlainText = "hello world";
    }
}
```

Or in Python:

```
from Symyx.Notebook.Sections.Text.Extensibility import ITextSection
for section in owner:
    if isinstance(section, ITextSection): section.TextDocument.PlainText =
        "hello world"
```

Only Text sections created in version 6.5 and later can implement `ITextSection`. You cannot check documents created in previous versions of Workbook using `ITextSection`. In versions prior to 6.5, use `section.GetType().AssemblyQualifiedName` to check the section type.

Scale an Image

The `Symyx.Notebook.Sections.Text.TextDocumentImages` class contains `TextDocumentImage` objects that represent images in a text document. The `TextDocumentImages` class, `Add` methods inserts structures, reactions, or images from in a specified filepath into the text section. Some `Add` methods can use a boolean parameter that specifies if the image is scaled to fit the text document. The `Add(string imagePath, float horizontalScalePercent, float verticalScalePercent)` signature enables scaling an image by a percentage to increase or decrease the image size.

The images added to a `TextDocument` object are returned in the `TextDocument.Images` property.

Chapter 7:

Query Service for Searching

The Symyx Framework provides a query service that allows client applications to search a Vault repository and the RAS data warehouse. The query service is built on ADO.NET. Client applications can invoke directly the ADO.NET classes to execute queries.

To connect to a data source, use `Symyx.Framework.Vault.Query.ADO.VaultDbConnection`. The current `VaultDbConnection` for the active Vault server of the current workspace can be obtained from the `VaultDbConnection` property of `Symyx.Framework.Vault.VaultServer`. The following example shows how to get the `VaultDbConnection` after logging into a workspace:

```
Vaultworkspace workspace = new Vaultworkspace(endpoint);
workspace.Login(username, password);
using (VaultDbConnection connection =
    (VaultDbConnection)workspace.ActiveVaultServer.VaultDbConnection)
{
    QueryByTitle(connection);
}
```

To search for Vault objects, use the classes in the `System.Data` namespace that is provided by the ADO.NET framework. The following example shows the ADO.NET classes `IDbCommand` and `IDbDataParameter` to search Vault objects by title. The `IDataReader` class gets the search results.

```
private static void QueryByTitle(VaultDbConnection connection)
{
    //Create a command object.
    IDbCommand command = connection.CreateCommand();
    //Define the SQL string.
    command.CommandText = "select Title, vaultId, vaultObjectType
        from VaultObject where Title like :Title";
    //Create a parameter object.
    IDbDataParameter parameter = command.CreateParameter();
    parameter.ParameterName = "Title";
    parameter.Value = "My Title%";
    //Add the parameter to the command.
    command.Parameters.Add(parameter);
    //Execute the command and get the results.
    using (IDataReader reader = command.ExecuteReader())
    {
        int rowCount = 0;
        while (reader.Read())
        {
            for (int i = 0; i < reader.FieldCount; i++)
            { Console.WriteLine("{0} = {1}", reader.GetName(i), reader[i]); }
            Console.WriteLine();
            rowCount++;
        }
        Console.WriteLine("rowCount = {0}", rowCount);
    }
}
```

For more information, see [ADO.NET classes](#).

RAS Data Schema

RAS stores information about Vault objects in the VAULTOBJECT_OBJ view in the RAS data warehouse. To query the VAULTOBJECT_OBJ view, construct the SQL query and use the `System.Data` namespace.

The following shows the data schema of the VAULTOBJECT_OBJ view in the RAS data warehouse:

Name	Null?	Type	Description
LINKID		VARCHAR2 (80)	An object reference which is created when a new object is instantiated and preserved after the object is saved.
STATUS	NOT NULL	NUMBER (11)	The status bits of a persistent object with a specified type and ID, and, possibly, the status of all its sub-objects.
CREATIONDATE		DATE	The creation date and time of the object.
CREATEDBY		VARCHAR2 (255)	The user who created the object.
LASTMODIFICATIONDATE		DATE	The date and time on which the object was last modified.
LASTMODIFIEDBY		VARCHAR2 (255)	The user who last modified the object.
ALIAS		VARCHAR2 (255)	An alternate display name for the object.
ASSOCIATIONTARGET		NUMBER (1)	True if the object can be the target of a user-defined association.
CHECKOUTUSERNAME		VARCHAR2 (255)	The user who checked out the object.
CHECKOUTSTATE		VARCHAR2 (255)	The checkout state of the object at the time it was retrieved from the server.
CLASS		VARCHAR2 (255)	The assembly-qualified typename of the .NET class that implements the containing object.
CONTENTIDENTIFIER		VARCHAR2 (255)	An ID that references the content of the object.
CONTENTSIZE		NUMBER (11)	The size of the object data.
CONTRIBUTOR		VARCHAR2 (255)	An entity responsible for making contributions to the resource.
COVERAGE		VARCHAR2 (255)	The user description of the scope of the object.
CREATOR		VARCHAR2	The ID of the user making the first Vault check-in of the

Name	Null?	Type	Description
		(255)	object.
CLIENTCREATIONDATE		DATE	The date and time that the object was originally created on the client.
DESCRIPTION		VARCHAR2 (4000)	A user description of the object, such as an abstract, a table of contents, or a free-text account of the content.
FORMAT		VARCHAR2 (255)	The media type of the object in MIME-type format.
LANGUAGE		VARCHAR2 (255)	A user description of the language used by the object.
VAULTOBJECTTYPE		VARCHAR2 (255)	The specific type of the Vault object.
PERMISSIONS		VARCHAR2 (255)	Current explicit and implicit permissions to the object.
IMPLICITPERMISSIONS		VARCHAR2 (255)	Permissions that are inherited from another object.
PUBLISHER		VARCHAR2 (255)	The name and version of the application creating the object.
RELATION		VARCHAR2 (255)	A user description of related work, for example, a literature reference.
VAULTPATH		VARCHAR2 (255)	The path to the object in the repository.
VAULTCONTAINERPATH		VARCHAR2 (255)	The path to the object container in the repository.
RIGHTS		VARCHAR2 (255)	A user description of rights held to the object.
SOURCEREPOSITORYID		VARCHAR2 (255)	The source VaultObjectId from which this VaultObject was retrieved.
SOURCE		VARCHAR2 (255)	The resource from which the described resource is derived.
SUBJECT		VARCHAR2 (1000)	A user description of the object's topic (keywords and key phrases).
TITLE		VARCHAR2 (1000)	A user title of the object, for display purposes.
TYPE		VARCHAR2 (255)	The nature of the object's content ("image", "text", etc).
VAULTID	NOT	VARCHAR2	The unique system identifier for the object.

Name	Null?	Type	Description
	NULL	(255)	
VERSION		VARCHAR2 (255)	The version number of the object which is incremented with each object change.
VERSIONCOMMENT		VARCHAR2 (4000)	Description of the current version of the object content.
VERSIONCREATIONDATE		DATE	The date and time on which the current version of the object is created.
VERSIONCREATOR		VARCHAR2 (255)	The user ID responsible for the current version of the object content.
VERSIONIDENTIFIER		VARCHAR2 (255)	The system identifier for the specific version of the object.
WORKFLOWNAME		VARCHAR2 (255)	The name of the workflow in which the object is participating.
WORKFLOWSTAGE		VARCHAR2 (255)	The stage of the workflow the object is in.
LOCKED		NUMBER (1)	True if the object is locked.
DELETED		NUMBER (1)	True if the object is deleted.
HIDDEN		NUMBER (1)	True if the object is hidden.
SYSTEM		NUMBER (1)	True if the object is a system object.
ARCHIVE		NUMBER (1)	True if the object has been updated since the last archive pass.
SYNCHRONIZE		NUMBER (1)	True if the object is included in synchronization.

Most of the columns in the VAULTOBJECT_OBJ view are Vault object core properties as enumerated in `Symyx.Framework.Properties.CoreProperty`.

Query Form Data

When a new form is created using the Symyx Framework, the Framework maps and saves the form data into a new table in the RAS data warehouse.

The view name format is as follows:

`FORMAT + sequential_number`

For example:

`FORMAT000000001`

The column names are the design-time name of the field on the form.

To query form data in the data warehouse, construct the SQL query using its data schema and use the `System.Data` namespace.

Search Results

Use the ADO.NET class `IDataReader` to retrieve the search results. The searching example also shows how to get the search results using `IDataReader`:

```
//Execute the command and get the results.
//Assume command is IDbCommand that already contains the SQL. using
(IDataReader reader = command.ExecuteReader())
{
    int rowCount = 0;
    while (reader.Read())
    {
        for (int i = 0; i < reader.FieldCount; i++)
        { Console.WriteLine("{0} = {1}",reader.GetName(i), reader[i]); }
        Console.WriteLine(); rowCount++;
    }
    DebugLogDataTable(reader.GetSchemaTable());
    Console.WriteLine("rowCount = {0}", rowCount);
}
```

The private method `DebugLogDataTable`, used in the preceding example, shows how to get the data in a table using the ADO.NET classes `DataTable`, `DataRow`, and `DataColumn`:

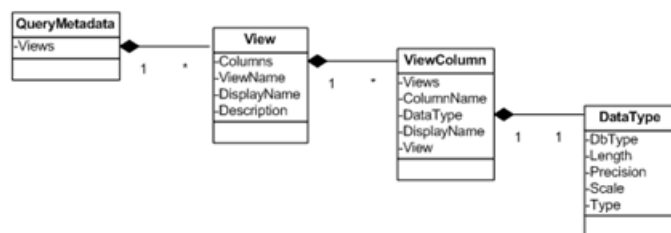
```
private static void DebugLogDataTable(DataTable table)
{
    foreach(DataRow row in table.Rows)
    {
        foreach (DataColumn column in table.Columns)
        {
            Console.WriteLine(string.Format("{0} = {1}", column.ColumnName,
                row[column]));
        }
    }
}
```

For details about the ADO.NET classes, see the [Microsoft System.Data Namespace documentation](#).

Create a Custom Query Builder Using Metadata

A custom query builder is a user interface that enables users to enter a search criteria. The query builder must have access to the metadata of the data to search. The

`Symyx.Framework.Vault.Query.Metadata` namespace contains classes that give access to the metadata. The following diagram shows the class hierarchy in the `Symyx.Framework.Vault.Query.Metadata` namespace:



The `Symyx.Framework.Vault.Query.ADO.VaultDbConnection` class contains the `QueryMetadata` property which represents the structure of the connected database. The `QueryMetadata` property returns the `Symyx.Framework.Vault.Query.Metadata.QueryMetadata` class.

The following example shows how to get the metadata:

```
private static void GetMetadata(VaultDbConnection connection)
{
    QueryMetadata metadata = connection.QueryMetadata;
    foreach (View view in metadata.Views.Values)
    {
        Console.WriteLine(view.ViewName);
        foreach (ViewColumn column in view.Columns)
        { Console.WriteLine(column.ColumnName); }
    }
}
```

Custom Vault Objects Indexing

If you create a custom Vault object, you can:

- Create a custom index for searching and retrieving the custom Vault object. See [Custom indexing](#).
- Enable the custom Vault object for full-text searching. See [Full text search indexing](#).

Full-text Search Indexing

If you want to enable full-text searching of your custom Vault object's contents, the custom Vault object class must implement the `Symyx.Framework.Vault.IIndexableText` or `Symyx.Framework.Vault.IIndexableTextContainer` interface. The Vault text indexer indexes any Vault object that implements one of these interfaces. If the object implements `IIndexableText`, it is added to the full text index in the Vault data warehouse. If the object implements `IIndexableTextContainer`, the indexer iterates through all elements returned by the `IIndexableTextContainer.TextEnumerator` property, and adds them to the index.

The `IIndexableText` interface has three properties:

- **BinaryContent**
`IIndexableText` implements this property when the custom Vault object has binary content such as a Microsoft Word file or PDF attachment.
- **TextContent**
`IIndexableText` implements this property when the custom Vault object's content to search is a text string.

■ Name

Identifies the text object.

IndexableText Implementation Example

The following is a sample implementation of an object that contains an image and text caption that is searchable:

```
public class Image : IIndexableText
{
    private string caption_;
    private string name_;
    public TextReader TextContent
    {
        get
        {
            return new StringReader(caption_.ToString());
        }
    }
    public string Name
    {
        get
        {
            return name_;
        }
    }
}
```

Custom Indexing

You can use create a custom message processor to index data for a search. The custom message processor runs using events that occur the server such as the saving of an experiment. You can use custom indexing to index custom Vault objects or default Vault objects.

For an example of a custom message processor, see the sample projects included in the samples folder of the SDK installation.

Creating a New Search Type

The Symyx SDK provides a way to extend the Vault Search system. The example is installed within the Symyx SDK:

```
\samples\Symyx.SDK.Samples.SectionIdSearchExtension\
```

The examples in the samples folder assume you have [NUnit](#), an open source unit testing framework for .NET.

You can create a new search type that will plug into the Vault Search drop down list in Notebook Explorer and provide the user with new ways to find the data stored in Vault.

For details about API calls, see the API Reference for `Symyx.Framework.Controls.IQueryBuilderGrid` interface, `Symyx.Framework.Query` class, `Symyx.Framework.QueryLogicExpression` class, and other query-related classes.

SampleIDSearchExtension

The `SectionIdSearchExtension` class is a sample SDK search extension for Workbook. It allows a user to enter the document section Id and then finds the Document that contains the corresponding section.

This sample shows how an SDK author can create a new search type by creating a class that implements the `Symyx.Framework.Controls.IQueryBuilderGrid` interface. A class that implements this interface can be configured to show up in the Vault Search dropdown list as a new search type. The new search type can be used like any other search. You can display the search results in the Notebook Explorer, save the search results as a list, and save the search as a favorite.

The `SectionIdSearchExtension` is a simple example of creating a new search type. It also demonstrates some of the best practices for creating new searches.

Build Queries

When you create a search extension, construct the query from the user input to run the query and return the results in the Notebook Explorer Vault object grid. Use the `BuildQuery` method in the `IQueryBuilderGrid` interface to return the results to Notebook Explorer.

Use the following code after the query is created:

```
query.UsePropertiesQuerySQLGeneration();  
query.ObtainParentContainers = true;
```

Simple Condition Example

The example in the [SectionID search extension](#) creates a `QueryView` to the `VaultObject_obj` table, as well as, a condition that ensures the `VaultId` field of the `VaultObject_obj` table is equal to the `VaultId` specified by the user.

```
/// <summary>  
/// Builds a query from the user input. The SDK programmer will use  
/// the Query object API found in the Symyx.Framework.Query namespace  
/// to convert the user input found in the UI to a valid Query object.  
/// </summary>  
/// <returns>A query.</returns>
```

```
public Symyx.Framework.Query BuildQuery()  
{  
    currentQuery_ = BuildQuery(GetDocumentSectionVaultId());  
    return currentQuery_;  
}
```

```
/// <summary>  
/// Builds the query object from the user input.  
/// </summary>  
/// <param name="vaultId">The vault id.</param>  
/// <returns></returns>
```

```
public Query BuildQuery(VaultId vaultId)  
{  
    // Create the query view  
    QueryView vaultObjectTableView = new QueryView  
        ("VaultObject_obj", false);  
    // Create the condition  
    QueryCondition condition = new QueryCondition
```

```

        (new QueryField(vaultObjectTableView, "VaultID"),
        QueryComparisonOperator.QueryComparisonOperators.EqualTo,
        vaultId);
    Query query = new Query(condition);
    // Set the ObtainParentContainers variable on the query.
    query.ObtainParentContainers = true;
    return query;
}

```

Review the `Symyx.Framework.Query` and `SymyxFramework.Query.Extensions` namespaces before creating a search extension.

Specific Materials and Amounts Example

The following example assumes that a user needs to search for a document that has a materials table in which 1000 grams of Benzene or 1000 grams of water are used. Pass the specific material names such as Benzene and Water, and the specific amounts and units to the `BuildQuery` method from the Search user interface.

```

return (
(
(
"P_MATERIAL_OBJ.NAME".Is
(QueryComparisonOperator.QueryComparisonOperators.EqualTo, "Benzene") |
"P_MATERIAL_OBJ.NAME".Is
(QueryComparisonOperator.QueryComparisonOperators.EqualTo, "Water")
).Grouped() & "P_PLANNEDAMOUNT_OBJ.AMOUNT_VALUE".Is
(QueryComparisonOperator.QueryComparisonOperators.EqualTo, 1000) &
"P_PLANNEDAMOUNT_OBJ.AMOUNT_UNIT".Is
(QueryComparisonOperator.QueryComparisonOperators.EqualTo,
(int)UnitKey.GRAM)
).AllOnSameRow()
).AllInSameSection().AllInSameDocument();

```

Using Synonyms Example

```

private Query Query1000gBenzeneOrWater_UsingIsOneOf()
{
return (
"P_MATERIAL_OBJ.NAME".Is(QueryComparisonOperator.QueryComparisonOperators.IsOneOf
, "Aspirin", "Synonym1", "Synonym2") &
"P_PLANNEDAMOUNT_OBJ.AMOUNT_VALUE".Is(QueryComparisonOperator.QueryComparisonOperators.EqualTo, 1000) &
"P_PLANNEDAMOUNT_OBJ.AMOUNT_UNIT".Is(QueryComparisonOperator.QueryComparisonOperators.EqualTo, (int)UnitKey.GRAM)
).AllOnSameRow().AllInSameSection().AllInSameDocument();
}

```

Specific Folder Example

```

private Query Query1000gBenzeneLocatedInSpecificFolder()
{
return (
    ("P_MATERIAL_OBJ.NAME".Is

```

```
(QueryComparisonOperator.QueryComparisonOperators.EqualTo, "Benzene") &
    "P_PLANNEDAMOUNT_OBJ.AMOUNT_VALUE".Is
(QueryComparisonOperator.QueryComparisonOperators.EqualTo, 1000) &
    "P_PLANNEDAMOUNT_OBJ.AMOUNT_UNIT".Is
(QueryComparisonOperator.QueryComparisonOperators.EqualTo,
(int)UnitKey.GRAM)
    ).AllOnSameRow()
    ).AllInSameSection() &
    "VAULTOBJECT_OBJ.VAULTPATH".Is
(QueryComparisonOperator.QueryComparisonOperators.Contains,
destinationFolder_.VaultPath)
    ).All
```

Timestamp Example

```
private Query Query1000gBenzeneCreatedByCurrentUserToday()
{
    return (
        (
            ("P_MATERIAL_OBJ.NAME".Is
(QueryComparisonOperator.QueryComparisonOperators.EqualTo"Benzene") &
            "P_PLANNEDAMOUNT_OBJ.AMOUNT_VALUE".Is
(QueryComparisonOperator.QueryComparisonOperators.EqualTo, 1000) &
            "P_PLANNEDAMOUNT_OBJ.AMOUNT_UNIT".Is
(QueryComparisonOperator.QueryComparisonOperators.EqualTo,
(int)UnitKey.GRAM)).AllOnSameRow()
            ).AllInSameSection() &
            "VAULTOBJECT_OBJ.CREATOR".Is
(QueryComparisonOperator.QueryComparisonOperators.EqualTo,
workspace_.CurrentUser.VaultId) &
            "VAULTOBJECT_OBJ.CREATIONDATE".Is
(QueryComparisonOperator.QueryComparisonOperators.DateTimeEqualTo,
DateTime.Today)
            ).AllInSameDocument();
    }
}
```

Title Example

```
private Query Query1000gBenzeneAndSpecificDocumentTitleContents()
{
    return (
        (
            ("P_MATERIAL_OBJ.NAME".Is
(QueryComparisonOperator.QueryComparisonOperators.EqualTo, "Benzene") &
            "P_PLANNEDAMOUNT_OBJ.AMOUNT_VALUE".Is
(QueryComparisonOperator.QueryComparisonOperators.EqualTo, 1000) &
            "P_PLANNEDAMOUNT_OBJ.AMOUNT_UNIT".Is
(QueryComparisonOperator.QueryComparisonOperators.EqualTo,
(int)UnitKey.GRAM)
            ).AllOnSameRow()
            ).AllInSameSection() & "VAULTOBJECT_OBJ.TITLE".Is
(QueryComparisonOperator.QueryComparisonOperators.Contains, "Benzene") &
            "VAULTOBJECT_OBJ.TITLE".Is
```

```
(QueryComparisonOperator.QueryComparisonOperators.Contains, "Water")
).AllInSameDocument();
}
```

Search Extension Development Best Practices

The lazy initialization pattern is an example of a best practice for performance. The lazy initialization pattern is found in the `IQueryBuilderGrid.UserControl` method. The `UserControl` method provides the graphical user interface for the search extension.

If the user interface has a single text box in which the user can enter the document section ID, you can create a `UserControl` that allows the user to construct a more complex search.

The following example demonstrates the lazy creation of the view class, the preferred method, because you do not control when the search extension is loaded into Workbook.

```
/// <summary>
/// Gets the UserControl view for this search extension.
/// Note: we use lazy view creation here so that we pay
/// the cost of view creation only when the view is needed.
/// </summary>
/// <value>The user control.</value> public UserControl UserControl
{
    get
    {
        if (view_ == null)
        {
            view_ = new SectionIdSearchExtensionView(this);
            if (currentQuery_ != null)
            {
                view_.SectionId = FindSectionIdInQuery(currentQuery_).ToString();
            }
            ApplyConfiguration();
        }
        return view_;
    }
}
```

Using the Model-View-Controller pattern is a best practice, which aids in improving usability. It is easier for users to launch the search with the Enter key, rather than using a mouse to click the Search button. Use the `ExecuteSearchRequest` event in the `IQueryBuilderGrid` Interface to implement the model-view-controller pattern. In the following example, use the `ExecuteSearchRequest` event to run the query after the user enters the section ID and presses the enter key.

From the `SectionIdSearchExtensionView` class:

```
/// <summary>
/// Handles the keyPress event of the documentSectionVaultId control.
/// </summary>
/// <param name="sender">The source of the event.</param>
/// <param name="e">The <ref="System.Windows.Forms.KeyPressEventArgs" />
instance containing the event data.
///</param>
private void documentSectionVaultId_KeyPress
(object sender, KeyPressEventArgs e)
```

```
{
// char 13 represents the ENTER key
if(e.KeyChar == (char)13)
{
    e.Handled = true; model_.RaiseExecuteQueryRequest();
}
}
```

From the `SectionIdSearchExtension` class:

```
internal void RaiseExecuteQueryRequest()
{
    if( GetDocumentSectionVaultId() != VaultId.Empty)
    {
        EventHandler handler = ExecuteQueryRequest; if (handler != null)
        {
            handler(this, new EventArgs());
        }
    }
}
```

Search Extension Configuration

Search extensions are configured through the `SearchExtensionService.SearchExtensions` application permission. You can view and edit the `SearchExtensionService.SearchExtensions` application permission in Workbook.

The first dictionary entry is called `SearchExtensionTypes` and is an XML-defined list of all known search extensions. Each entry contains a fully-qualified assembly class name for a class that implements the `IQueryBuilderGrid` interface and a key from the extension.

Example `SearchExtensions` XML:

```
<?xml version="1.0" encoding="utf-16"?>
<SearchExtensions>
  <SearchExtension
class="Symyx.SDK.Samples.SectionIdSearchExtension.SectionIdSearchExtension,
SectionIdSearchExtension, Culture=neutral, PublicKeyToken=null"
key="SectionIdSearchExtension" />
</SearchExtensions>
```

All subsequent entries in the dictionary are used to configure the individual search extensions. Set the name to the search extension key. The entries have a `SearchExtension` node that defines the specific extension to configure, followed by a configuration data node. You can put any valid XML inside the configuration data node, and the information is passed to the search extension using the configuration property when the extension is instantiated.

Example Search Extension configuration XML:

```
<?xml version="1.0" encoding="utf-16"?>
  <SearchExtension key="SectionIdSearchExtension"
    displayName="Document Section Id Search">
    <ConfigurationData>
      <SectionIdSearchExtensionConfiguration>
        <SectionIdSearchLabel>
          section id *from config*
        </SectionIdSearchLabel>
      </SectionIdSearchExtensionConfiguration>
    </ConfigurationData>
  </SearchExtension>
```

```

    </SectionIdSearchLabel>
  </SectionIdSearchExtensionConfiguration>
</ConfigurationData>
</SearchExtension>

```

Search Extension Publishing

While you are developing a search extension, use the Private Assembly model for distribution. After compiling and configuring the search extension, copy the compiled assembly binary (.dll) and program database (.pdb) file to the directory where the Symyx.Notebook.Application executable file (.exe) resides. Workbook finds the search extension when it dynamically loads the search extension types.

To publish the search extension so that other users can use the new search type, use the Vault Object Package (.vozip) publishing system to upload the Search Extension assembly to Vault. To learn how a Vault Object Package is created, see the SearchExtensionVoziPBuilder project in the sample code. An assembly that is published in this manner must be fully signed. Workbook does not dynamically load a search extension that is not properly signed. The SearchExtensionTypes entry must contain updated public key token information.

Example SearchExtensions XML with public key token specified:

```

<?xml version="1.0" encoding="utf-16"?>
  <SearchExtensions>
    <SearchExtension
class="Symyx.SDK.Samples.SectionIdSearchExtension.SectionIdSearchExtension,
SectionIdSearchExtension, Culture=neutral, PublicKeyToken= fb4b5791c48b7e8a"
key="SectionIdSearchExtension" />
  </SearchExtensions>

```

Chapter 8:

Scripting in BIOVIA Workbook

Scripting provides a powerful mechanism for extending capabilities of Workbook. Scripting is supported as extension points for customization by providing event handlers that execute custom IronPython scripts. IronPython scripts use the Python programming language and run within the Microsoft .NET Framework.

The scripting framework within Workbook provides script variables that represent Symyx Framework objects.

The script variables allow your scripts to use Workbook API and perform custom operations, for example, editor is one of the script variables that are available with Experiment Editor scripting. The editor variable represents a `Symyx.Notebook.ApplicationManagement.IDocumentEditor` object. Your script can call the appropriate `IDocumentEditor` API to get information about or invoke methods on the Experiment Editor.

The following script invokes the `IDocumentEditor.GetMenuItem` method, which returns a `System.Windows.Forms.ToolStripItem`, and then sets the `ToolStripItem.Enabled` property:

```
item = editor.GetMenuItem('viewToolStripMenuItem')
if item is not None:
    item.Enabled = False
```

Workbook Objects

To invoke the Workbook API from your script, import the class that you need from the appropriate namespace, and use the API calls directly within your script. For example, the following script invokes `Symyx.Notebook.Sections.Table.AddNewRowsForm` to prompt a user to add rows to a table.

```
from Symyx.Notebook.Sections.Table import AddNewRowsForm
from System.Windows.Forms import DialogResult
try:
    addNew = AddNewRowsForm()
    result = addNew.ShowDialog()
    if result == DialogResult.OK:
        i = 0
        while i < addNew.RowCount:
            table.AddRow()
            i = i + 1
        finally:
            addNew.Dispose()
```

The preceding example uses the table script variable that represents `Symyx.Notebook.Sections.Table.TableSection` and provides the `AddRow` method.

The `IronPython.dll` is placed in the global assembly cache (GAC) as part of the Workbook installation.

Python Scripting

- Use proper indentations.

In Python, leading white spaces or indentations at the beginning of each logical line are interpreted by the Python parser. If you copy and paste sample scripts from this document, verify that the indentations are correct in the pasted script.

- Use straight double-quote (") characters

If you copy an IronPython script from a text editor (such as Microsoft Word) and copy it into the IronPython Script Editor, ensure that any smart quote, (") or ("), from your text editor are entered as straight double-quote characters (") in the IronPython Script editor. If your script contains smart quotes, the script fails to run as expected.

- Use proper error-handling in your scripts perform

To catch exceptions, use try-except-finally statements. Also, initially check for non-null or valid input before performing an operation or calculation. An unhandled exception in your script might prevent subsequent scripts from executing or might cause subsequent scripts to fail.

Note: On a listbox, setting selectionmode to none is not supported; only selected items in a form can appear in reports.

Script Performance Profile

You can monitor and profile the performance of your scripts by enabling the script execution performance log file on the client. In addition to logging script execution times, performance log files are also created when a user checks in Vault objects in the check-in performance log file, and when a user checks out and opens Vault objects, in the check-out and open performance log file. These performance log files help isolate and identify performance bottlenecks on Workbook client.

Document Toolbar Scripting

Scripts that are triggered by the Experiment Editor or Form Editor events automatically incur a performance penalty during user activities such as opening, editing, and saving documents.

Use document-level dynamic toolbar buttons to execute the scripts instead of executing scripts during events such as `OnApplicationLoaded`, `OnSaving`, `OnSaved`, and `OnSectionActivated` in the Experiment Editor, or during events such as `OnEdit` and `OnClick` events in the Form Editor. When using a custom toolbar button, a script only executes when the user clicks the button.

You can add menu items to customize Workbook. In Workbook you can add custom menu items at run time using document event scripts. Use document-level dynamic toolbar buttons instead of using custom menu items to provide custom functionality. The scripts do not need to execute scripts to add menu items during user activities. Toolbar button gives the user the option to execute what is needed only at the appropriate time without a performance penalty.

For more information about toolbar scripting, see [Custom Toolbar Scripting](#).

Optimize Scripts

If you need to perform custom logic during start-up or during common document events, there are potential performance costs. Avoid unnecessary or repetitive calls, and avoid possibly long-running server code. For example, if a script executes server calls to populate a list in a combo box, avoid executing it during the Experiment Editor's `OnApplicationLoaded` event. The following are some alternatives:

- Populate a hidden listbox with the vocabulary only when the document is first created. This works if the vocabulary doesn't change often.

- Create an add-in that runs at login-time to download the vocabulary as text snippets to the local disk. This makes login a few seconds slower but everything else faster. This works if it is acceptable for users to wait until the next login to update the vocabularies.
- Use the combo box list event to execute the server call to populate the list only when the user drops down the combo box.
- Use a check box to enable the controls containing the combo box, and get the vocabulary only if the user clicks the check box. Alternatively, use a button to allow a user to click the button to get the vocabulary.

To help you optimize your script, use the diagnostic logging that is available with Workbook. The diagnostic log helps in monitoring the performance of your script execution. For example, if your script is performing server or external calls, it is useful to profile its performance to determine whether it should be executed from an *automatic* event such as `OnSectionActivated` and `OnSectionDeactivated` event or from an event that requires user action such as the button `OnClick` event.

Form Editor Scripting

Using the Form Editor in Workbook, you can specify IronPython scripts that execute when events on the form, `BaseForm`, or form widgets are triggered. Scripts can perform custom processing such as data validation, automatic updates, or display data in the form. Specify your scripts in the property sheet for a widget or a base form in the Form Editor.

FormSection Events

The following table contains the form or widget events that can execute IronPython scripts.

Event	Description	Type	Script variables
OnClick	Occurs when the button is clicked.	Button	active_ form
OnEdit	Occurs when the form is loaded for editing.	Form	active_ form
OnValidate	Occurs when the widget on the form changes See OnValidate Script for an example.	CheckBox, ComboBox, ListBox, TextBox	active_ widget active_ form
OnReview	Occurs when the widget is reviewed. Use an <code>OnReview</code> script to manipulate review messages in the <code>active_widget.ReviewResults</code> that is a <code>Symyx.Framework.Review.ReviewResultCollection</code> object. See OnReview Script for an example. Displaying a Message Box or any UI component from an <code>OnReview</code> scripts is not supported. Review scripts are also run on the server, invoking a UI component could cause problems.	CheckBox, ComboBox, ListBox, TextBox, PictureBox	active_ widget active_ form

Event	Description	Type	Script variables
OnValueChanged	Occurs when the value of a widget on the form changes. The OnValueChanged event occurs after the OnValidate event occurs, and is not fired when the widget validation fails.	Form	active_form

The events execute scripts on a Form document and its widgets. You can assign scripts to these events using the Form Designer.

Form section scripts do not reference ELN assemblies. To create Workbookobjects, you need to reference the form section in your scripts using a Python script as follows:

```
import clr
clr.AddReference("Symyx.Framework")
```

FormSection Script Variables

Within a script, you can access and update the properties of the current form and its widgets by using the following variables:

- **active_form** - Represents the current form, `Symyx.Notebook.Forms.BaseForm`. The following example sets the back color of the active form.
- `active_form.BackColor = System.Drawing.Colors.Blue`
- `active_form.Controls["widgetName"]` represents a widget (whose name is specified by `widgetName`) on the current form.
 - The following example sets (toggles) the `ReadOnly` property of a `TextBox` named `Comments`:
`active_form.Controls["Comments"].ReadOnly = not active_form.Controls["Comments"].ReadOnly`
 - `active_form.Controls["groupBoxName"].Controls["widgetName"]` represents a widget (whose name is specified by `widgetName`) in a `GroupBox` (whose name is specified by `groupBoxName`) on the current form.
 - The following example increments the `SelectedIndex` property of a `ComboBox` in a `GroupBox`:
`newIdx = active_form.Controls["GroupBox1"].Controls["lbColors"].SelectedIndex + 1`
- **active_widget** - Represents the current widget on the form (an object that implements `Symyx.Notebook.Forms.IModifiableWidget` such as a `CheckBox`, `ComboBox`, `ListBox`, `PictureBox`, and `TextBox`).
 - The following example sets text on a `TextBox` control:
`active_widget.Text = "hello world"`
 This is equivalent to:
`active_form.Controls["TextBox1"].Text = "hello world"`
 - `active_widget.ReviewResults` returns a `Symyx.Framework.Review.ReviewResultCollection` object containing review results for the widget.
- **active_section** - Represents the form section containing the `BaseForm` (`Symyx.Notebook.DocumentSection`).

Note: `active_section` is only available in a `FormSection` within an experiment; `active_section` is null in the Form Designer.

- `active_document` - Represents the document containing the form section (`Symyx.Notebook.Document`).

Note: `active_document` is only available in a `FormSection` within an experiment; `active_document` is null in the Form Designer.

- `active_workspace` - Represents the current Vault workspace (`Symyx.Framework.Vault.Vaultworkspace`).

Note: `active_workspace` is only available in a `FormSection` within an experiment; `active_workspace` is null in the Form Designer.

FormSection Events Script Variables

The following tables show the script variables that are available to the events on a Form. You can add scripts for the events using the Form Designer.

OnClick Event

Applies to Button

Script variable	Represents
<code>active_form</code>	<code>Symyx.Notebook.Forms.BaseForm</code>
<code>active_section</code>	<code>Symyx.Notebook.Sections.Forms.FormsSection</code> Use <code>active_section</code> if the Form is used in a <code>FormSection</code> within an experiment. Note: <code>active_section</code> returns null when used within the Form Designer.
<code>active_document</code>	<code>Symyx.Notebook.Document</code> that owns the <code>FormsSection</code> Use <code>active_document</code> if the Form is used in a <code>FormSection</code> within an experiment. Note: <code>active_document</code> returns null when used within the Form Designer.
<code>active_workspace</code>	<code>Symyx.Framework.Vault.Vaultworkspace</code> Use <code>active_workspace</code> if the Form is used in a <code>FormSection</code> within an experiment. Note: <code>active_workspace</code> returns null when used within the Form Designer.

OnEdit Event

Applies to Form

Script variable	Represents
<code>active_form</code>	<code>Symyx.Notebook.Forms.BaseForm</code>
<code>active_section</code>	<code>Symyx.Notebook.Sections.Forms.FormsSection</code> Use <code>active_section</code> if the Form is used in a <code>FormSection</code> within an experiment. Note: <code>active_section</code> returns null when used within the Form Designer.

Script variable	Represents
active_document	<p><code>Symyx.Notebook.Document</code> that owns the <code>FormsSection</code> Use <code>active_document</code> if the Form is used in a <code>FormSection</code> within an experiment.</p> <p>Note: <code>active_document</code> returns null when used within the Form Designer.</p>
active_workspace	<p><code>Symyx.Framework.Vault.Vaultworkspace</code> Use <code>active_workspace</code> if the Form is used in a <code>FormSection</code> within an experiment.</p> <p>Note: <code>active_workspace</code> returns null when used within the Form Designer.</p>

OnValidate Event

Applies to CheckBox, ComboBox, ListBox, TextBox

Script variable	Represents
active_form	<code>Symyx.Notebook.Forms.BaseForm</code>
active_widget	<code>Symyx.Notebook.Forms.IModifiablewidget</code>
active_section	<p><code>Symyx.Notebook.Sections.Forms.FormsSection</code> Use <code>active_section</code> if the Form is used in a <code>FormSection</code> within an experiment.</p> <p>Note: <code>active_section</code> returns null when used within the Form Designer.</p>
active_document	<p><code>Symyx.Notebook.Document</code> that owns the <code>FormsSection</code> Use <code>active_document</code> if the Form is used in a <code>FormSection</code> within an experiment.</p> <p>Note: <code>active_document</code> returns null when used within the Form Designer.</p>
active_workspace	<p><code>Symyx.Framework.Vault.Vaultworkspace</code> Use <code>active_workspace</code> if the Form is used in a <code>FormSection</code> within an experiment.</p> <p>Note: <code>active_workspace</code> returns null when used within the Form Designer.</p>

OnReview Event

Applies to CheckBox, ComboBox, ListBox, TextBox, PictureBox

Script variable	Represents
active_form	<code>Symyx.Notebook.Forms.BaseForm</code>
active_widget	<code>Symyx.Notebook.Forms.IModifiablewidget</code>
active_section	<p><code>Symyx.Notebook.Sections.Forms.FormsSection</code> Use <code>active_section</code> if the Form is used in a <code>FormSection</code> within an experiment.</p> <p>Note: <code>active_section</code> returns null when used within the Form Designer.</p>
active_document	<p><code>Symyx.Notebook.Document</code> that owns the <code>FormsSection</code> Use <code>active_document</code> if the Form is used in a <code>FormSection</code> within an experiment.</p>

Script variable	Represents
	Note: active_document returns null when used within the Form Designer.
active_workspace	Symyx.Framework.Vault.Vaultworkspace Use active_workspace if the Form is used in a FormSection within an experiment. Note: active_workspace returns null when used within the Form Designer.

OnValueChanged Event

Applies to Form

Script variable	Represents
active_form	Symyx.Notebook.Forms.BaseForm
active_section	Symyx.Notebook.Sections.Forms.FormsSection Use active_section if the Form is used in a FormSection within an experiment. Note: active_section returns null when used within the Form Designer.
active_document	Symyx.Notebook.Document that owns the FormsSection Use active_document if the Form is used in a FormSection within an experiment. Note: active_document returns null when used within the Form Designer.
active_workspace	Symyx.Framework.Vault.Vaultworkspace Use active_workspace if the Form is used in a FormSection within an experiment. Note: active_workspace returns null when used within the Form Designer.

Access Widgets

A FormSection can use the following widgets or controls:

- Button
- CheckBox
- ComboBox
- GroupBox
- Label
- ListBox
- PictureBox
- TextBox

Each of the widgets represents an object that has properties, methods, and events that you can access or invoke in your script.

To view the properties, methods, and events of these widgets, see their API member listings in the product's API documentation. The widgets inherit from corresponding base controls in `System.Windows.Form`, and also inherit their API. For more information, see [Microsoft System.Windows.Form documentation](#).

To access a widget, specify:

```
active_form.Controls["widgetName"]
```

The widgetName is the name of the widget on the current form. In the following example, the Controls["TextBox1"] specifies the name of a TextBox control.

To set the Text property use:

```
active_form.Controls["TextBox1"].Text = "hello world"
```

To access a widget in a GroupBox, specify:

```
active_form.Controls["groupBoxName"].Controls["widgetName"]
```

The groupBoxName is the name of the GroupBox on the current form. WidgetName is the name of the widget in the specified GroupBox. In the following example, Controls["txtTitle"] and Controls["lblTitle"] specify TextBox and Label controls in a GroupBox control named, GroupBox1. In this example, the copyTitle function is registered to the TextBox.TextChanged event:

```
def copyTitle(sender,e):
    active_form.Controls["GroupBox1"].Controls["lblTitle"].Text =\
        active_form.Controls["GroupBox1"].Controls["txtTitle"].Text
    active_form.Controls["GroupBox1"].Controls["txtTitle"].TextChanged
+= copyTitle
```

To run this script:

1. In Workbook, create a new form.
2. In the Form Designer, add a GroupBox named GroupBox1.
3. In GroupBox1, add the following widgets:
 - A TextBox named txtTitle
 - A Label named lblTitle
4. On the form, select the txtTitle TextBox widget.
5. On the Properties pane for txtTitle, select **Validation > OnValidate**. Add the script.
6. Select **View > Preview** to see the form.
7. Enter text in txtTitle, and press the Tab key. Change the text on txtTitle. The changed text in txtTitle is automatically copied into lblTitle.

OnReview Script Example

The following is a sample script for a ComboBox widget's OnReview event. It checks if the user selected an item from the ComboBox, and adds messages to the ReviewResultCollection.

```
clr.AddReference("Symyx.Framework")
from Symyx.Framework.Review import *
# Get the current ReviewResults of the ComboBox
reviewResultCollection = active_widget.ReviewResults
# Get the selected value from the ComboBox.
s = active_widget.SelectedItem
# If no selected value, add an error to ReviewResults;
# else add an informational message.
if s == "" or s == None:
    err = "From ReviewScript(ColorComboBox) - Please select a value."
    reviewResultCollection.Add(ReviewResult(SeverityLevel.Error, err))
else:
    info = "From ReviewScript(ColorComboBox) - You selected " + s
```

```
reviewResultCollection.Add(ReviewResult(SeverityLevel.Information, info))
# Update the ReviewResults of the ComboBox
active_widget.ReviewResults = reviewResultCollection
```

To run this script:

1. In Workbook, create a new form.
2. In the Form Designer, import the `form_OnReview.snform` located in the `NotebookDocExamples` directory of the SDK documentation.
3. On the form, select the `cbColor` ComboBox widget.
4. On the **Properties** pane for `txtSolvent`, open the `OnReview` script.
You do not have to change anything in the script.
5. Save and check-in the form.

To see the review results, create an experiment with a Form Section using the form you just checked in:

- In the **Experiment Editor**, select **View > Review Results** to see the results of the script.

OnValidate Script Example

The following is a sample script for a TextBox widget's `OnValidate` event. It retrieves values from a database based on the text entered, and displays the values in a ComboBox.

```
import sys
import clr
clr.AddReference("System.Data")

from System import *
from System.Data import *
from System.Data.OleDb import *
# Reset the combobox
active_form.Controls["cbSolventClasses"].Items.Clear()
# Perform a lookup in a DB if there is a name to look up
solventName = active_form.Controls["txtSolvent"].Text
if not String.IsNullOrEmpty(solventName) :
    connection = OleDbConnection
        ("Provider=Microsoft.Jet.OLEDB.4.0;
        DataSource=C:\\solventdb.mdb;
        Persist Security Info=False")
    query = String.Format
        ("select * from solventdb where name like '%{0}%', solventName)
    adapter = OleDbDataAdapter(query, connection)
    solvents = DataSet()
    adapter.Fill(solvents)
    rows = solvents.Tables[0].Rows.Count
    # If found, get the values and add them to the combobox
    if solvents.Tables.Count > 0 and rows > 0 :
        active_form.Controls["cbSolventClasses"].BeginUpdate()
        for i in range(rows):
            dr = solvents.Tables[0].Rows[i]
            temp = dr["Name"].ToString()
            active_form.Controls["cbSolventClasses"].Items.Add(temp)
        active_form.Controls["cbSolventClasses"].EndUpdate()
```

To run this script:

1. In the Workbook, create a new form.
2. In the Form Designer, import the `form_OnValidate.snform` located in the `NotebookDocExamples` directory of the BIOVIA Workbook SDK documentation.
3. On the form, select the `txtSolvent` `TextBox` widget.
4. On the Properties pane for `txtSolvent`, open the `OnValidate` script. Search for `Source=C:_temp\\solventdb.mdb` in the script, and edit its appropriate location.
The `solventdb.mdb` file is a Microsoft Access database file located in the `NotebookDocExamples` directory of the BIOVIA Workbook SDK documentation. After you edit the script, close the Script Editor.
5. Select **View > Preview** to see the form.

To use the lookup service:

- Type a word to look up, and press the **Tab** key.
The `cbSolventClasses` list displays a list of solvent classes that were retrieved from the database.

Add Scripts to an Experiment Template

In the Experiment Editor, you can create custom experiments and experiment templates. You can add toolbar buttons to experiments created from an experiment template. You can write Python scripts that respond events that occur in the experiment. Event scripting is available using the `Scripts` property in the Experiment Editor's `Experiment` and `Section` properties. For more information about scripting, see "Scripting in BIOVIA Workbook" in the *BIOVIA Workbook SDK*.

The following sections of an experiment are for internal use only, do not use these sections in scripts:

- Grouped Materials
- Plate Layout
- Reaction List
- Spreadsheet

IMPORTANT! You must have administrator or Template Editor permission to add scripts to the Experiment Editor.

To add a script using the Experiment Editor:

1. In an open experiment template, select **View > Properties**.
2. In the **Properties** pane, select the **Experiment** or **Section** tab.
3. Navigate to **Event Scripting**, and click the ellipsis button.
4. In the **Event Scripting** dialog, select the event to access.
5. Click **Add Script**.
6. Using Python write a script to handle the event.
7. Click **OK**, and test the script by initiating the event.
8. Implement changes as needed.

Experiment Editor Events

The following table contains the Experiment Editor events that can execute IronPython scripts. The event descriptions include the `EventArgs` that are available to the event script.

Note: In Workbook, if an `OnLockingSection` script encounters an exception, the Experiment Editor user interface is not refreshed and subsequent sections are not property locked or unlocked. To avoid unhandled exceptions from your script, perform proper error handling within your script.

Event	Description
<code>OnApplicationClosing</code>	Occurs when the Experiment Editor is about to close. Scripts executed during this event can choose to cancel the close. The <code>EventArgs</code> type is <code>Symyx.Framework.ApplicationManagement.ApplicationClosingEventArgs</code> , whose properties are: <ul style="list-style-type: none"> ■ <code>e.Cancel</code> - indicates whether the event is canceled. ■ <code>e.ReasonForCanceling</code> - specifies the reason for canceling the event.
<code>OnApplicationLoaded</code>	Occurs when a document is fully loaded in the Experiment Editor. The <code>EventArgs</code> type is <code>Symyx.Framework.ApplicationManagement.ApplicationLoadedEventArgs</code> .
<code>OnInsertingSection</code>	Occurs before the Experiment Editor inserts a section. Scripts executed during this event can choose to cancel the insertion. The <code>EventArgs</code> type is <code>Symyx.Notebook.ApplicationManagement.InsertingSectionEventArgs</code> , whose properties are: <ul style="list-style-type: none"> ■ <code>e.Section</code> - specifies the section to insert. ■ <code>e.Cancel</code> - indicates whether the event is canceled. ■ <code>e.CancelReason</code> - specifies the reason for canceling the event.
<code>OnLockingSection</code>	Occurs before the Experiment Editor locks a section. Script executed during this event can choose to cancel the lock. The <code>EventArgs</code> type is <code>Symyx.Notebook.ApplicationManagement.LockingSectionEventArgs</code> , whose properties are: <ul style="list-style-type: none"> ■ <code>e.Section</code> - specifies the section to insert. ■ <code>e.Cancel</code> - indicates whether the event is canceled. ■ <code>e.CancelReason</code> - specifies the reason for canceling the event.

Event	Description
OnMenuItemEnabledStatesUpdated	<p>Occurs after the Experiment Editor updates the <i>Enabled</i> state of its menu items. If the event occurred due to a user menu selection, that menu is included in the event arguments. The EventArgs type is <code>Symyx.Notebook.ApplicationManagement.MenuItemEnabledStatesUpdatedEventArgs</code>, whose properties are:</p> <ul style="list-style-type: none"> ■ <code>e.MenuOpening</code> - indicates if the menu is opened or selected. ■ <code>e.Menu</code> - specifies the menu to open.
OnRemovingSection	<p>Occurs before the Experiment Editor removes a section. Scripts executed during this event can choose to cancel the removal. The EventArgs type is <code>Symyx.Notebook.ApplicationManagement.RemovingSectionEventArgs</code>, whose properties are:</p> <ul style="list-style-type: none"> ■ <code>e.Section</code> - specifies the section to remove. ■ <code>e.Cancel</code> - indicates whether the event is canceled. ■ <code>e.CancelReason</code> - specifies the reason for canceling the event.
OnSaving	<p>Occurs before the experiment is saved. Scripts executed during this event can choose to cancel the save. The EventArgs type is <code>Symyx.Notebook.ApplicationManagement.SavingEventArgs</code>, whose properties are:</p> <ul style="list-style-type: none"> ■ <code>e.Cancel</code> - indicates if the event is canceled. ■ <code>e.CancelReason</code> - specifies reason for canceling the save. ■ <code>e.IsAutoSave</code> - Indicates an automatic save. <p>An OnSaving script executes before a user responds to a prompt to save changes, if the user attempts to exit the Experiment Editor without saving any changes.</p>
OnSaved	<p>Occurs after the experiment is saved. The EventArgs type is <code>Symyx.Notebook.ApplicationManagement.SavedEventArgs</code>.</p>
OnSectionInserted	<p>Occurs after a section is inserted. The EventArgs type is <code>Symyx.Notebook.ApplicationManagement.SectionInsertedEventArgs</code>, whose property is:</p> <ul style="list-style-type: none"> ■ <code>e.Section</code> - specifies the section to insert.
OnSectionLocked	<p>Occurs after a section is locked. The EventArgs type is <code>Symyx.Notebook.ApplicationManagement.SectionLockedEventArgs</code>, whose property is:</p>

Event	Description
	<ul style="list-style-type: none"> ■ <code>e.Section</code> - specifies the section to lock.
OnSectionRemoved	<p>Occurs after a section is removed. The EventArgs type is <code>Symyx.Notebook.ApplicationManagement.SectionRemovedEventArgs</code>, whose property is:</p> <ul style="list-style-type: none"> ■ <code>e.Section</code> - specifies the section to remove.
OnSectionUnlocked	<p>Occurs after a section is unlocked.</p> <p>The EventArgs type is <code>Symyx.Notebook.ApplicationManagement.SectionUnlockedEventArgs</code>, whose property is:</p> <ul style="list-style-type: none"> ■ <code>e.Section</code> - specifies the section that was unlocked.
OnToolBarButtonEnabledStatesUpdated	<p>Occurs after the Experiment Editor updates the Enabled state of its toolbar items. The EventArgs type is <code>Symyx.Notebook.ApplicationManagement.ToolBarButtonEnabledStatesUpdatedEventArgs</code>.</p>
OnUnlockingSection	<p>Occurs before the Experiment Editor unlocks a section. Scripts executed during this event can choose to cancel the unlock.</p> <p>The EventArgs type is <code>Symyx.Notebook.ApplicationManagement.UnlockingSectionEventArgs</code>, whose properties are:</p> <ul style="list-style-type: none"> ■ <code>e.Section</code> - specifies the section to unlock. ■ <code>e.Cancel</code> - indicates whether the event is canceled. ■ <code>e.CancelReason</code> - specifies the reason for canceling the event.

Experiment Editor Event Scripts

• Events on the Experiment Editor, not the active document: The events listed in “Experiment Editor events” above are events on the Experiment Editor, not on the active document. If your script performs an action on the document such as removing or adding a section, none of the Experiment Editor events will occur because these events are on the Experiment Editor, not the document.

- One Python script interpreter instance per session

Only one instance of the Python script interpreter is instantiated per Experiment Editor session. As a result, when import statements are run from any script, they affect all subsequent scripts until the Experiment Editor is closed.

- A script change on a document does not affect its dirty bit

An Experiment Editor event script that makes a change to the active document does not affect the dirty bit on the properties and content of the document. However if the script needs to set the dirty bit on the document, set the `editor.Document.IsDirty` property to true:

```
editor.Document.IsDirty = True
```

- Displaying the signature dialog when locking a section

If the Lock Section Signature property of a section is set to require a signature, locking the section from a script does not display the signature dialog that requires the user to sign. For example, the following script does not display the signature dialog:

```
editor.ActiveDocumentSection.IsLocked=True
```

To display the signature dialog, use the following script:

```
editor.LockAndUnlockSections()
```

- Check the existence of a dynamic ToolBar item before setting its Enabled or Visible property

The `OnToolBarButtonEnabledStatesUpdated` event fires when dynamic toolbars are not initialized. This causes exceptions to be thrown in scripts designed to set the Enabled or Visible properties of dynamic toolbar buttons, unless the scripts first check whether the toolbar exists. The following example first checks if the toolbar button `IsTemplate` exists before setting the Visible property of the button:

```
if sender.GetToolBarItem("mode", "IsTemplate") is not None:
    sender.GetToolBarItem("mode", "IsTemplate").Visible =
    sender.Document.IsTemplate
```

Experiment Editor Events Script Variables

The following variables are available to Experiment Editor event scripting. These script variables represent objects in the Workbook environment. Using these script variables, you can invoke the appropriate API on the objects they represent. For details about the API that can be used with these script variables, see the API Reference. For examples, see [Add Scripts to an Experiment Template](#).

Variable name	Description
editor	Implements the <code>Symyx.Notebook.ApplicationManagement.IDocumentEditor</code> interface in the <code>Symyx.Notebook.dll</code> . The Experiment editor object implements other interfaces, but only the properties, events and methods in the <code>IDocumentEditor</code> interface are supported for scripting purposes.
sender	Same as editor.
owner	The document or document section on which the script is attached.
e	Container for event-specific objects. See the "<Event Name>EventArgs" classes in the <code>Symyx.Notebook.ApplicationManagement</code> namespace in <code>Symyx.Notebook.dll</code> . For example, in <code>Symyx.Notebook.ApplicationManagement.InsertingSectionEventArgs</code> the <code>e</code> contains <code>e.Section</code> , the section to insert, <code>e.Cancel</code> , a boolean value indicating if the script cancels the insertion, and <code>e.CancelReason</code> , a string value the script can set to explain why it canceled the insertion.
active_workspace	The <code>Symyx.Framework.Vault.VaultWorkspace</code> object. It is the same as the active workspace inserted into forms section script environment. Instead of using <code>active_workspace</code> , you can use <code>Symyx.Framework.Vault.VaultWorkspace</code> in your script. For example: <pre>from Symyx.Framework.Vault import VaultWorkspace isOnline = VaultWorkspace.Current.IsOnline</pre>

Get the Active Section

If an experiment has multiple sections, and your script needs to perform operations on the active section, use the `editor.ActiveDocumentSection` property. For example:

```
section = editor.ActiveDocumentSection
```

Access Menu Items

A script has access to the menus and menu items in the Workbook Experiment Editor, Text Section, File Section, and Form Section. To access the menu items, use the `editor.GetMenuItem('menuItemName')` method. Use the name of the menu item as the value in `menuItemName` parameter.

For example, the following script disables the View menu in the Experiment Editor:

```
item = editor.GetMenuItem('viewToolStripMenuItem')
if item is not None:
    item.Enabled = False
```

If you change a menu item property, the modification is not saved when the menu is opened because menu items are recreated each time the menu is opened. If you want to set menu item properties, use the `OnMenuItemEnabledStatesUpdated` event to make the changes. If you need to set a menu item property based on a calculation in a different event such as `OnSaved`, you can persist the result in a file or in a Document property. For example, the following script for the `OnSaved` event gets the `openToolStripMenuItem` and sets the `editor.Document.Description` property:

```
item = editor.GetMenuItem("openToolStripMenuItem")
if item is not None:
    editor.Document.Description = "Disable openToolStripMenuItem"
```

A script used with the `OnMenuItemEnabledStatesUpdated` event can inspect the `editor.Document.Description` property to determine whether or not it disables the `openToolStripMenuItem`, for example:

```
item = editor.GetMenuItem("openToolStripMenuItem")
if editor.Document.Description == "Disable openToolStripMenuItem":
    item.Enabled = False
else:
    item.Enabled = True
```

Menu Item Property Changes

If you change a property of a menu item, the modification is not saved when the menu is opened because the menu items are recreated every time the menu is opened. If you want to set the properties of a menu item, set them in the `OnMenuItemEnabledStatesUpdated` event. If you need to set a property of a menu item based on a calculation in a different event such as `OnSaved`, you can persist the result in a file or in a Document property. For example, the following script for the `OnSaved` event gets the `openToolStripMenuItem` and sets the `editor.Document.Description` property:

```
item = editor.GetMenuItem("openToolStripMenuItem")
if item is not None:
    editor.Document.Description = "Disable openToolStripMenuItem"
```

Then a script for the `OnMenuItemEnabledStatesUpdated` event can inspect the `editor.Document.Description` property to determine whether or not it disables the `openToolStripMenuItem` item = editor.GetMenuItem("openToolStripMenuItem").

```
if editor.Document.Description == "Disable openToolStripMenuItem":  
    item.Enabled = False  
else:  
    item.Enabled = True
```

Menu Item Names

The following lists show the names of the menu items that are accessible in the Experiment Editor, Text Section, File Section, and Form Section. Select a name from the list when using the `editor.GetMenuItem` method:

Menu item names for the ExperimentEditor

- `fileToolStripMenuItem`
- `openToolStripMenuItem`
- `exitToolStripMenuItem`
- `undoToolStripButton`
- `redoToolStripButton`
- `viewToolStripMenuItem`
- `toolsToolStripMenuItem`
- `closeToolStripMenuItem`
- `notebookExplorerToolStripMenuItem`
- `preferencesToolStripMenuItem`
- `editToolStripMenuItem`
- `undoToolStripMenuItem`
- `redoToolStripMenuItem`
- `cutToolStripMenuItem`
- `copyToolStripMenuItem`
- `pasteToolStripMenuItem`
- `deleteToolStripMenuItem`
- `selectAllToolStripMenuItem`
- `helpToolStripMenuItem`
- `contentsToolStripMenuItem`
- `aboutToolStripMenuItem`
- `toggleOnlineToolStripMenuItem`
- `newToolStripMenuItem`
- `windowToolStripMenuItem`
- `onlineStripMenuItem`
- `offlineStripMenuItem`
- `saveToolStripMenuItem`
- `checkInToolStripMenuItem`
- `checkOutToolStripMenuItem`
- `undoCheckOutToolStripMenuItem`

- historyToolStripMenuItem
- pageSetupToolStripMenuItem
- printPreviewToolStripMenuItem
- printToolStripMenuItem
- cloneToolStripMenuItem
- arrangeAllToolStripMenuItem
- compareSideBySideToolStripMenuItem
- transitionToolStripMenuItem
- editPropertySetsMenuItem

Menu item names for the Text Section

- editToolStripMenuItem
- selectAllToolStripMenuItem
- findToolStripMenuItem
- replaceToolStripMenuItem
- viewToolStripMenuItem
- normalViewToolStripMenuItem
- pageLayoutViewToolStripMenuItem
- showStatusBarToolStripMenuItem
- showHorizontalRulerToolStripMenuItem
- showVerticalRulerToolStripMenuItem
- zoomToolStripMenuItem
- zoom25ToolStripMenuItem
- zoom50ToolStripMenuItem
- zoom75ToolStripMenuItem
- zoom100ToolStripMenuItem
- zoom150ToolStripMenuItem
- zoom200ToolStripMenuItem
- zoom300ToolStripMenuItem
- insertToolStripMenuItem
- insertImageToolStripMenuItem
- insertPageBreakToolStripMenuItem
- formatToolStripMenuItem
- fontDialogToolStripMenuItem
- formatParagraphToolStripMenuItem
- formatTabsToolStripMenuItem
- formatBulletsAndNumberingToolStripMenuItem
- formatBulletsAndNumberingAttributesToolStripMenuItem
- increaseNumberingLevelToolStripMenuItem
- decreaseNumberingLevelToolStripMenuItem

- `formatBulletsAndNumberingAsArabicNumbersToolStripMenuItem`
- `formatBulletsAndNumberingAsCapitalLettersToolStripMenuItem`
- `formatBulletsAndNumberingAsLowercaseLettersToolStripMenuItem`
- `formatBulletsAndNumberingAsRomanNumeralsToolStripMenuItem`
- `formatBulletsAndNumberingAsLowercaseRomanNumeralsToolStripMenuItem`
- `bulletsToolStripMenuItem`
- `formatImageToolStripMenuItem`
- `formatTextColorToolStripMenuItem`
- `formatTextBackgroundColorToolStripMenuItem`
- `formatDocumentBackgroundColorToolStripMenuItem`
- `tableToolStripMenuItem`
- `tableInsertToolStripMenuItem`
- `insertTableToolStripMenuItem`
- `insertTableColumnLeftToolStripMenuItem`
- `insertTableColumnRightToolStripMenuItem`
- `insertTableRowAboveToolStripMenuItem`
- `insertTableRowBelowToolStripMenuItem`
- `tableDeleteToolStripMenuItem`
- `deleteTableToolStripMenuItem`
- `deleteTableColumnToolStripMenuItem`
- `deleteTableRowsToolStripMenuItem`
- `splitTableToolStripMenuItem`
- `splitTableAboveToolStripMenuItem`
- `splitTableBelowToolStripMenuItem`
- `tableSelectToolStripMenuItem`
- `selectTableToolStripMenuItem`
- `selectTableRowToolStripMenuItem`
- `selectTableCellToolStripMenuItem`
- `showTableGridLinesToolStripMenuItem`
- `tablePropertiesToolStripMenuItem`

Menu item names for the Form Section

- `insertToolStripMenuItem`
- `insertFormToolStripMenuItem`
- `toolsToolStripMenuItem`
- `manageFormsToolStripMenuItem`

Menu item names for the File Section

- `menuEditDeleteSection_`
- `menuEditRenameSection_`

- menuEditReplaceSectionWith_
- menuInsertInsertFile_
- menuViewDisplayAnnotations_
- menuFileSaveSectionAs_
- menuEditEditExternalFile_
- menuViewRefresh_
- menuEditSelectWorksheets_

Access Workbook Toolbar Items

A script has access to the tool bars and tool bar items on the Experiment Editor, Text Section, File Section, and Form Section. To access the toolbar items that are available, use the

```
editor.GetToolBarItem('toolbarName', 'toolbarItemName')
```

In the method, `toolbarName` is the name of the toolbar, and `toolbarItemName` is the name of the toolbar item.

For example, the following script disables the Delete toolbar button on the Experiment Editor. The `standardToolStrip` is the name of the toolbar and `deleteToolStripButton` is the name of the toolbar item:

```
item = editor.GetToolBarItem('standardToolStrip', 'deleteToolStripButton')
if item is not None:
    item.Enabled = False
```

Section Toolbars and Toolbar Items

The following lists show the names of the toolbar and toolbar items that are accessible in the Experiment Editor, Text Section, File Section, and Form Section. Use the toolbar name and the toolbar item name from the appropriate list when using the `editor.GetToolBarItem` method.

Section Name	Toolbar Name	Toolbar Item
Experiment Editor	standardToolStrip	newToolStripButton openToolStripButton saveToolStripButton printToolStripButton cutToolStripButton copyToolStripButton pasteToolStripButton deleteToolStripButton
Text Section	dateTimeToolbar	DateToolStripButton TimeToolStripButton DateTimeToolStripButton
	standardToolStrip	findToolStripButton undoToolStripButton redoToolStripButton spellingToolStripButton

Section Name	Toolbar Name	Toolbar Item
	formattingToolStrip	fontToolStripComboBox fontSizeToolStripComboBox boldToolStripButton italicsToolStripButton underlineToolStripButton subscriptToolStripButto superscriptToolStripButton alignLeftToolStripButton alignCenterToolStripButton alignRightToolStripButton justifyToolStripButton numberingToolStripButton bulletsToolStripButton decreaseIndentToolStripButton increaseIndentToolStripButton zoomToolStripComboBox showMarkupToolStripButton
Form Section	sectionToolStrip	insertFormBrowseToolStripButton
File Section	standardToolStrip	fonts_ sizes_ cutToolStripButton copyToolStripButton pasteToolStripButton deleteToolStripButton undoToolStripButton redoToolStripButton cmdBold_ cmdItalic_ cmdUnderline_ cmdAlignLeft_ cmdAlignCenter_ cmdAlignRight_ cmdIncreaseIndent_ cmdDecreaseIndent_ cmdBullets_ cmdChangeBackColor_ cmdChangeForeColor_

Section Name	Toolbar Name	Toolbar Item
File Section	standardToolStrip	sizes_ cutToolStripButton copyToolStripButton pasteToolStripButton deleteToolStripButton undoToolStripButton redoToolStripButton cmdBold_ cmdItalic_ cmdUnderline_ cmdAlignLeft_ cmdAlignCenter_ cmdAlignRight_ cmdIncreaseIndent_ cmdDecreaseIndent_ cmdBullets_ cmdChangeBackColor_ cmdChangeForeColor_
	PdfToolBar_	cmdFirstPage_ cmdPreviousPage_ cmdNextPage_ cmdLastPage_ txtCurrentPage_ labelDivisor_ labelPagesCount_ cmdSinglePage_ cmdSingleContinuous_ cmdFacingContinuous_ cmdFacingPages_ ccmdZoomInTool_ cmdFitPage_ cmdFitWidth_ cmdRotateRight_ labelSearch_ txtSearchCriteria_ cmdFindPrevious_ cmdFindNext_

Section Name	Toolbar Name	Toolbar Item
	PropertiesToolBar	createLine_ createArrow_ createFreehand_ createRectangle_ createEllipse_ createImage_ createText_ drawColor_ fillColor_ lineWidth_ lineStyle_ arrowStyle_ opacity_
File Section	ZoomToolBar_	cmdZoomOut_ txtCurrentZoom_ abelPrecent_ cmdZoomIn_

Access Workbook Toolstrips

A script has access to the toolstrips on the Experiment Editor. To access the toolstrips that are available, use the `editor.GetToolStrip('toolStripName')` method where `toolStripName` is the name of the Workbook toolstrip. The toolstrip names are:

In the `editor.GetToolStrip` method, the `standardToolStrip` contains the New, Open, Save, Print, Cut, Copy, Paste, and Delete button.

The `SectionReferencesStrip` contains the button for Sample IDs referenced by a section.

Check User Permissions and Disable a Section Example

The following example disables a Text Section if the current user does not have permission.

```
hasPermission = active_
workspace.CurrentUser.ApplicationPermissions.HasExecutePermissionFor("Symyx.Notebook", "SectionTemplate.Editor")
for section in editor.Document.Sections:
    if section.Title == 'Text':
        section.ReadOnly = not hasPermission
editor.RefreshView()
```

To run this script:

1. Login to Workbook as a user with the `SectionTemplate.Editor` permission.
2. Create a new experiment template in Experiment Editor. Add a Text Section.
3. Add the script to the `OnApplicationLoaded` event of the Text Section.
4. Save and check in the template.
5. Log off the Workbook client.
6. Log in as a user without `SectionTemplate.Editor` permission.

7. Create an experiment using the template created in the preceding steps.

You should see a disabled Delete Section option in the Text section.

Remove the Active Section

The following example checks the active section can be removed, then removes the active section from the document if the section can be deleted.

```
import clr
clr.AddReference('System.Windows.Forms')
from System.Windows.Forms import MessageBox MB
section = editor.ActiveDocumentSection
if section.AllowSectionDelete:
    editor.Document.Remove(section)
    editor.Document.IsDirty = True
    editor.RefreshView()
else:
    MB.Show('Section cannot be deleted.')
```

To run this script:

1. Login to Workbook as a user with the SectionTemplate.Editor permission.
2. Create a new experiment template in Experiment Editor. Add any type of section.
3. On the Properties pane for that section, set Allow Section Delete to false.
4. Add the script to a Dynamic Toolbar item.
5. Save the template.
6. Click the toolbar item you just created. You should see the MessageBox indicating that the section cannot be deleted.

The `Symyx.Notebook.DocumentSectionProperty.AllowSectionDelete` field is read-only in the API. The end-user can set `AllowSectionDelete`.

Rename the Active Section

The following example checks if the active section can be renamed, then renames the active section:

```
from System.Windows.Forms import MessageBox as MB
section = editor.ActiveDocumentSection
if section.AllowSectionRename:
    section.Title = 'New Title'
    editor.Document.IsDirty = True
    editor.RefreshView()
else:
    MB.Show('Section cannot be renamed')
```

To run this script:

1. Log in to the as a user with the SectionTemplate.Editor permission.
2. Create a new experiment template in Experiment Editor. Add any type of section.
3. On the Properties pane for that section, set Allow Section Rename to false.
4. Add the script to a Dynamic Toolbar item.
5. Save the template.
6. Click the toolbar item you just created. You should see the MessageBox indicating that the section cannot be renamed.

Add a Button to a Toolstrip

The following example adds a button to the `SectionReferenceStrip` toolstrip that displays a message box. The

`SectionReferenceStrip` is the toolstrip with the button for Sample IDs referenced by a section.

```
import System
from System.Windows.Forms import MessageBox
from System.Windows.Forms import MessageBoxButtons
from System.Windows.Forms import MessageBoxIcon
from System.Windows.Forms import ToolStripButton
from System.Windows.Forms import ToolStripSeparator

def ShowMessage(sender, e):
    MessageBox.Show('Hello world', editor.Title, MessageBoxButtons.OK,
        MessageBoxIcon.Information)
    tool_strip = editor.GetToolStrip('SectionReferencesStrip')
    tool_strip_item = ToolStripButton('Show Message', None, ShowMessage,
        'myToolStripButton')
    tool_strip.Items.Add(ToolStripSeparator())
    tool_strip.Items.Add(tool_strip_item)
```

Use `document.Title` to refer to the title of a document. Use `editor.Title` to refer to the titlebar of a window.

To run this script:

1. Login to the Workbook as a user with the `SectionTemplate.Editor` permission.
2. Create a new experiment template in Experiment Editor. Add any type of section.
3. Add the script to any of the events of the experiment section, for example, to the `OnSaving` event.
4. Save the template.

If you added the script to the `OnSaving` event, you should see a `ShowMessage` toolbar item next to the Sample ID toolbar item, as the following view:



Custom Toolbar Scripting

You can create custom toolbars for a document section, or for the entire experiment, and add buttons that execute IronPython scripts in the Template editor. The Dynamic Toolbar property of a document section in Workbook opens the Dynamic Toolbar Editor. Use the Dynamic Toolbar Editor to create or update custom toolbars with custom buttons.

Custom ToolStripButton Example

The following example is a script for a custom `ToolStripButton` that displays information about the custom section.

```
import clr
clr.AddReference("System.Windows.Forms")
from System.Windows.Forms import MessageBox
from System.Text import StringBuilder
sb = StringBuilder()
sb.Append("About this section:\n\n")
sb.Append("Title: " + editor.ActiveDocumentSection.Title + "\n")
sb.Append("Class: " + editor.ActiveDocumentSection.Class + "\n")
sb.Append("Type: " + editor.ActiveDocumentSection.Type + "\n")
sb.Append("Version: " + editor.ActiveDocumentSection.Version.ToString)
```

```
() + "\n")
MessageBox.Show(sb.ToString(), "Custom Toolbar Example")
```

Assign a Script to a Toolbar Button

To assign a script to a custom toolbar button for the entire document or experiment follow the instructions below but use Dynamic Toolbars on the Experiment property sheet instead of Dynamic Toolbars on the Section property sheet.

To assign a script to a custom toolbar button for a Workbook section:

1. View **Properties > Section** to see the property sheet for that Workbook section. Click the ... button associated with Dynamic Toolbars to launch the Dynamic Toolbar Editor.
2. On the Dynamic Toolbar Editor, click **Add Toolbar** to create a new toolbar for the Workbook section, then click **Add Toolbar Item** to add a button to that new toolbar.
3. On the Dynamic Toolbar Item property sheet, change the property values accordingly. To add a script that the button will execute, click the ... button for the Script property to launch the Iron Python Script Editor. Enter your script in the Iron Python Script Editor:

Variables for Custom Toolbar Scripts

The following variables are available to scripts used with custom toolbar items. These script variables represent objects in the Workbook environment. Using these script variables, you can invoke the appropriate Symyx Framework API on the objects they represent. For details about the API that can be used with these script variables, see the API Reference.

Variable name	Description	Example
editor	<p>Implements the <code>Symyx.Notebook.ApplicationManagement.IDocumentEditor</code> interface in <code>Symyx.Notebook.dll</code>.</p> <div> <p>Note: The editor object implements other interfaces, but only the properties, events and methods in the <code>IDocumentEditor</code> interface are supported for scripting purposes.</p> <p>Interfaces are subject to change in subsequent releases. Scripts written against the changes interfaces other than the <code>IDocumentEditor</code> might break in future releases.</p> </div>	To get the active document section from the editor object: <code>editor.ActiveDocumentSection</code>
sender	The <code>System.Windows.Forms.ToolStripButton</code> that was clicked.	To get the <code>ToolStrip</code> of the current <code>ToolStripButton</code> : <code>toolStrip = sender.GetCurrentParent()</code>
e	The <code>System.EventArgs</code> for the <code>ToolStripButton</code> Click event.	None
active_workspace	The <code>Symyx.Notebook.Vault.NotebookWorkspace</code> object in <code>Symyx.Notebook.dll</code> .	<code>isOnline = active_workspace.IsOnline.ToString()</code>

Variable name	Description	Example
	It is the same as the active workspace inserted into forms section script environment.	
owner	For Table Section scripts that are executed by a custom toolbar item, owner is the only item such as TableSection on which the dynamic toolbar is defined.	

Custom toolbar scripts can also access the standard menu and toolbar items that are available in Workbook. For more information, see [Access Menu Items](#) and [Access Workbook Toolbar Items](#).

Interaction Between Scripts

The following example shows how scripts for multiple ToolStripButtons can interact with each other. For this example, two ToolStripButtons are defined on the toolbar: toolbarItem1 and toolbarItem2.

The script for toolbarItem1 toggles the Enabled property of toolbarItem2:

```
toolStrip = sender.GetCurrentParent()
otherButtons = toolStrip.Items.Find("toolbarItem2", True)
otherButton = otherButtons[0]
otherButton.Enabled = not otherButton.Enabled
```

Similarly, toolbarItem2 toggles the Enabled property of toolbarItem1:

```
toolStrip = sender.GetCurrentParent()
otherButtons = toolStrip.Items.Find("toolbarItem1", True)
otherButton = otherButtons[0]
otherButton.Enabled = not otherButton.Enabled
```

To run these scripts:

1. Login to Workbook as a user with the SectionTemplate.Editor permission.
2. Create a new experiment template in Experiment Editor. Add any type of section.
3. On the Properties pane of the section, configure (click) the Dynamic Toolbars property.
4. Add a toolbar, and create two toolbar items: toolbarItem1 and toolbarItem2.
5. Add the first script to toolbarItem1; add the second script to toolbarItem2.
6. Save the template.
When you click toolbarItem1, toolbarItem2 is disabled. When you click toolbarItem1 again, toolbarItem2 is enabled.

Insert an Excel File

The following example inserts an Excel file into a File Section.

```
import clr clr.AddReference('System.Windows.Forms')
from System.Windows.Forms import OpenFileDialog from System.Windows.Forms
import DialogResult clr.AddReference('System')
from System import Array
fd = OpenFileDialog()
fd.Filter = 'Excel Files|*.xls;*.xlsx' if fd.ShowDialog() ==
DialogResult.OK:
```

```
filePaths = Array.CreateInstance(object, 1) filePaths[0] = fd.FileName
fileSection = editor.ActiveDocumentSection filePackages =
fileSection.BuildersManager.BuildExternalFilesPackages(filePaths, False)
fileSection.Files.InsertRange(0, filePackages, True)
```

To run this script:

1. Login to Workbook as a user with the `SectionTemplate.Editor` permission.
2. Create a new experiment template in Experiment Editor. Add a File Section.
3. On the Properties pane of the section, configure (click) the Dynamic Toolbars property.
4. Add a toolbar, and create a toolbar item, for example, *Insert Excel file*.
5. Add the script to the toolbar item.
6. Save the template.

When you click the toolbar item, the Open File dialog prompts you to select an Excel file. The selected file is displayed in the File Section.

Add a Section to an Experiment

The following example creates a File Section and adds it to the current experiment.

```
from Symyx.Framework.Vault import VaultWorkspace, VaultObjectType, DataScope
from Symyx.Notebook import Document, DocumentSection
# Get the FileSection template def getSectionTemplate(title):
sectionList = VaultWorkspace.Current.Get
(VaultObjectType.DocumentSectionTemplate, DataScope.Minimal)
sectionTemplate = sectionList.FindByTitle(title) if sectionTemplate is not
None:
sectionTemplate = VaultWorkspace.Current.Get(sectionTemplate.VaultId,
DataScope.All)
return sectionTemplate
# Create a FileSection
fileSectionTemplate = getSectionTemplate("File") newSection =
DocumentSection.Create(fileSectionTemplate) newSection.Title = "Attachments"
# Add the FileSection to the current experiment owner.Document.Add
(newSection)
```

To run this script:

1. Log in to Workbook as a user with the `SectionTemplate.Editor` permission.
2. In the **Experiment Editor** create a new experiment template.
3. Add a section.
4. On the Properties pane of the section, select the Dynamic Toolbars property to configure the change.
5. Add a toolbar, and create a toolbar item, for example, *Add Attachment Section*.
6. Add the script to the toolbar item.
7. Save the template.

When you click the toolbar item, the script will add a new File Section called *Attachments*.

8. Save and reopen the experiment to see the new section.

Error Handling in Scripts

Ensure that your scripts perform proper error-handling. To catch exceptions, use try-except-finally statements. Also, initially check for non-null or valid input before performing an operation or calculation. An unhandled exception in your script might prevent subsequent scripts from executing or might cause subsequent scripts to fail.

To catch all exceptions, place your code in a try block, display an error message in the except block, and place cleanup code in the finally block to continue with all event processing. For example:

```
import clr clr.AddReference('System.Windows.Forms')
from System.Windows.Forms import *
try:
x=1/0 except:
    MessageBox.Show('An exception occurred');
finally:
    \ This code runs whether there is an exception or not.
    MessageBox.Show('Put any clean up code here.')
```

You could display exceptions in the except block:

```
import clr clr.AddReference('System.Windows.Forms')
from System.Windows.Forms import *
x=1/0 except Exception, inst:
    MessageBox.Show(inst.ToString());

\ This code runs whether there is an exception or not.
    MessageBox.Show('Put any clean up code here.')
```

If you want to catch a specific exception, display an error message for that exception in the except block:

```
import clr
clr.AddReference('System.Windows.Forms')
from System.Windows.Forms import *
try:
x=1/0 except ZeroDivisionError:
    MessageBox.Show('A divide by zero exception occurred');
finally:
    \ This code runs whether there is an exception or not.
    MessageBox.Show('Put any clean up code here.')
```

Cancel an Action

To trap an error in a defined event such as OnLockingSection event, cancel the action, for example, cancel locking the section by setting `e.Cancel` to True and letting the Experiment Editor display the error to the user by setting `e.CancelReason` in the except block.

For example:

```
import clr
clr.AddReference('System.Windows.Forms')
from System.Windows.Forms import *
try:
x=1/0
except Exception, inst:
    e.Cancel = True
    e.CancelReason = inst.ToString()
finally:
```

```
\ This code runs whether there is an exception or not.  
MessageBox.Show('Put any clean up code here.')
```

The section is not locked. Any other scripts associated with the event continues to run.

Raising an Exception

To catch and raise an exception from a script, invoke `raise` in the `except` block.

For example:

```
import clr  
clr.AddReference('System.Windows.Forms')  
from System.Windows.Forms import *  
try:  
    x=1/0  
except Exception, inst:  
    raise inst  
finally:  
    \ This code runs whether there is an exception or not.  
    MessageBox.Show('Put any clean up code here.')
```

This halts all script execution and event processing. For example, the section is not locked.

sys.exit

Any exit code returned from IronPython using the `sys.exit()` call is ignored. It is assumed that the script author intended to exit early. If the script author intends to abnormally end from a script, the script should throw an exception.

Generate Unique IDs

You can grant Workbook client users the ability to generate unique IDs for items associated with a Workbook document such as sample, batch, material, equipment, experiment.

The `Symyx.Notebook.Vault.NotebookWorkspace` class provides two methods that generate unique IDs:

- Use `GenerateUniqueIds`, recommended for performance reasons, if you need multiple unique IDs. The method reduces the number of iterations needed to retrieve and save to the database.
Use the `#generate` method to generate a specified number of IDs;
`active_workspace.GenerateUniqueIds('SEQUENCE_NAME', <this_many>)`
- Use the `GenerateUniqueId` method when a single ID is needed.
Use the `#generate` a single ID, and requires a roundtrip to the database for one ID;
`active_workspace.GenerateUniqueId('SEQUENCE_NAME')`

Sequence Name for Unique IDs

Material sections include a script that automatically generates *SAMPLE* ids.

Note: The sequence increases for each generated ID, however values are not contiguous. The server handles requests on a first-come, first-served basis, as a result, multiple users might see IDs values increase quickly.

Your organization can associate a sequence name with a set of unique IDs. Your organization also determines how many sequences there are as well as their names.

SEQUENCE_NAME is a valid Oracle name with up to 26 ASCII characters. Underscores (_) are allowed in the sequence name, and spaces are not allowed. The sequence is automatically created the first time the call is made with this argument value without database administrator (DBA) intervention.

Note: The SEQUENCE_NAME maps to an Oracle sequence, VID_SEQUENCE_NAME.

ID Formatting

You can format the IDs any way you want. The formatting is done in the script just before it is associated with the object.

For example, to generate a sequence with the following formatting pattern:

```
"S-0000001", "S-0000002", "S-0000003"
```

Use the following Iron Python code:

```
sampleIdProperty.Value = "S-" + sampleId.ToString().PadLeft(7, '0')
```

Generate SampleID Example

The following example sets the SampleID for selected rows in a Table Section. The method is used to generate the SampleID values for the selected rows. If the Document.Autname property is assigned, it is used as a prefix.

```
from System import String
from Symyx.Framework.Vault import VaultWorkspace from
Symyx.Framework.WinForms import MessageBox
section = editor.ActiveDocumentSection
document = editor.Document
# the function that sets the ID to a given row in the table section
def SetSampleId(row, sampleId):
    sampleIdProperty = row.Properties['SampleIdentification']['SampleId']
    autname = document.Autname
    if String.IsNullOrEmpty(sampleIdProperty.Value):
        if not String.IsNullOrEmpty(autname):
            sampleIdProperty.Value = String.Format
                ('{0}-{1}', autname, sampleId)
        else:
            sampleIdProperty.Value = sampleId;
# the main body of the script that
#1) obtains IDs for all rows selected in the table in a single call
#2) iterates over each selected row and calls the method above
#3) reports any errors to the user
#
    if active_workspace.IsOnline:
        if not section.IsLocked:
            if not editor.IsReadOnly:
                rows = section.GetSelectedRows()
                if rows.Length > 0:
                    sampleId = active_workspace.GenerateUniqueIds
                        ('SAMPLE', rows.Length)
                    rowIndex = 0
                    for row in rows:
                        SetSampleId(row, sampleId[rowIndex])
                        rowIndex = rowIndex + 1
```

```
        section.View.RefreshData()
    else:
        MessageBox.Show
            ('Not available when document is opened ReadOnly')
    else:
        MessageBox.Show('Not available when section is locked')
    else:
        MessageBox.Show('Not available when working offline')
```

To run this script, add the `SampleIdentification` property set to a Table Section, and add the script to the `OnSaving` event of the section. Enter some data in the Table Section, select a few rows, and save the experiment. You should see generated IDs in the Sample ID fields.

List Variables in Scope

To learn which variables are in scope, use the following script to display a window that lists the variables in scope and excludes the built-in variables.

```
from System.Windows.Forms import MessageBox as MB
s = ""
s = str(globals().keys())
s.strip('[]')
pos = s.IndexOf(' builtins__')
s = s.replace(',', '\n')
MB.Show(s.Substring(1, pos-2))
```

Add a Dictionary to a Recipe Section

The following Python example script associates the "Weigh" vocabulary with the selection of terms associated with this operation. The script adds a `PropertyDictionary` property in the Recipe section. The script inserts data from the `EquipmentID` column of the Equipment section into a drop-down list box (the dictionary) for the fields for Top-Loading Balance and Analytical Balance.

```
if operation.Name == "Weigh" and (property.Key == "Top-Loading Balance" or
property.Key == "Analytical Balance"):
# sender is the current Recipe section and ParentContainer is the
# document for section in
sender.ParentContainer.Sections:
if section.Title == "Equipment": target = section
break else:
    raise RuntimeError, "SourceTable section not found"
rowid = 100
for row in target.GetRows(): dic_id = str(rowid)
dictionary[dic_id] = row.PropertySets
    ["Equipment"]["EquipmentId"].DisplayValue
rowid = rowid + 1
```

Content History for a Control

Workbook does not create a history for changes to the content in a form when the content was updated by a script associated with a form control. Workbook History does not record changes to content (text, data) in the following `Symyx.Notebook.Forms.IModifiableWidget` controls (CheckBox, ComboBox, ListBox, PictureBox) if that change was done by a script.

Changes to the content of a TextBox are recorded.

You can associate an IronPython script with a control event such as a button click event to record the changes to the content of a ComboBox. The following IronPython script shows an example.

```
import clr
clr.AddReference("Symyx.Framework")
from Symyx.Framework.History
import ContentHistoryEntry from System.Globalization
import CultureInfo from System import String

def getContext(baseFormName, widgetName):
    # context name looks like: baseForm1.widgetName
    context = String.Format
        (CultureInfo.CurrentCulture, "{0}.{1}", baseFormName, widgetName)
    return context

def formatDescription(control, lastText):
    currentText = control.Text
    if lastText is None or lastText == "":
        lastText = "<empty>"
    if currentText is None or currentText == "":
        currentText = "<empty>"
    desc = String.Format(CultureInfo.CurrentCulture,
        "{0} was changed from {1} to {2}", control.DisplayName,
        lastText, currentText)
    return desc
# description looks like: widgetName was changed from Delhi to Mumbai

def addContentHistory(lastText, control):
    if ActiveForm.ParentSection is not None and
        ActiveForm.ParentSection.PendingContentHistory is not None:
        context = getContext(ActiveForm.DisplayName, control.DisplayName)
        desc = formatDescription(control, lastText)
        contentHistory = ContentHistoryEntry(context, desc)
        ActiveForm.ParentSection.PendingContentHistory.Add(contentHistory)
        tbCity = ActiveForm.Controls["tbCity"]
        cmbCities = ActiveForm.Controls["cmbCities"]

        if cmbCities.Items.Contains(str(tbCity.Text)):
            lastText = cmbCities.Text
            # capture the original text or content
            cmbCities.SelectedIndex = cmbCities.Items.IndexOf
                (str(tbCity.Text))
            addContentHistory(lastText, cmbCities)
```

Form Control Content History

Workbook History does not record a change to content within the following `Symyx.Notebook.Forms.IModifiableWidget` controls if the change is done by a script: `CheckBox`, `ComboBox`, `ListBox`, `PictureBox`.

Note: Changes to the content of a `TextBox` are recorded by default.

The following IronPython script can be associated with, say, a button click event, to record changes to the content of a `ComboBox`.

```
import clr
clr.AddReference("Symyx.Framework")

from Symyx.Framework.History
import ContentHistoryEntry from System.Globalization
import CultureInfo from System import String

def getContext(baseFormName, widgetName):
# context name looks like: baseForm1.widgetName
    context = String.Format(CultureInfo.CurrentCulture,
        "{0}.{1}",baseFormName, widgetName)
    return context

def formatDescription(control, lastText):
    currentText = control.Text
    if lastText is None or lastText == "":
        lastText = "<empty>"
    if currentText is None or currentText == "":
        currentText = "<empty>"
    desc = String.Format(CultureInfo.CurrentCulture,
        "{0} was changed from {1} to {2}", control.DisplayName,
        lastText, currentText)
    return desc

# description looks like: widgetName was changed from Delhi to Mumbai
def addContentHistory(lastText, control):
    if ActiveForm.ParentSection is not None and
        ActiveForm.ParentSection.PendingContentHistory is not None:
        context = getContext(ActiveForm.DisplayName, control.DisplayName)
        desc = formatDescription(control, lastText)
        contentHistory = ContentHistoryEntry(context, desc)
        ActiveForm.ParentSection.PendingContentHistory.Add(contentHistory)

tbCity = ActiveForm.Controls["tbCity"]
cmbCities = ActiveForm.Controls["cmbCities"]

if cmbCities.Items.Contains(str(tbCity.Text)):
    lastText = cmbCities.Text
    # capture the original text or content
    cmbCities.SelectedIndex = cmbCities.Items.IndexOf(str(tbCity.Text))
    addContentHistory(lastText, cmbCities)
```

ELN Assembly Cache

The ELN Assembly Cache (EAC) is a core service in the Symyx Framework that provides dynamic access to custom .NET assemblies and the types within them. In concept, it is similar to the Global Assembly Cache (GAC) in the .NET Framework, except that the ELN Assembly Cache can also resolve versioned types by downloading custom assemblies that implement them from BIOVIA Vault Server. Its primary service is to instantiate an object that corresponds to a .NET-style assembly-qualified class name, such as the following:

```
"Symyx.Notebook.Sections.Background, Symyx.Notebook, Version=6.1.0.580,
Culture=neutral, PublicKeyToken=b77a5c561934e089"
```


When an assembly-qualified object is requested by an application, the Assembly Cache:

- First checks if that specific version of the implementing assembly has been loaded into the current AppDomain.
- If not currently loaded, the EAC checks in a local store for the specific version of the assembly that implements that type.
- If the EAC does not find the assembly and the application is online, for example, the Vault server is network-accessible, the EAC requests the assembly from Vault and saves the assembly in the local store.

Whether the assembly was found locally or downloaded from Vault, the assembly is dynamically loaded, and an object of the requested type is instantiated and returned.

A request for the assembly-qualified class `Symyx.Notebook.Sections.Background` while online, causes the Assembly Cache to:

- Determine if version 6.1.0.580 of `Symyx.Notebook.dll` has been loaded.
- If not, determine if version 6.1.0.580 of `Symyx.Notebook.dll` can be found in the local store.
- If not, request version 6.1.0.580 of `Symyx.Notebook.dll` from Vault and save it in the local store.
- Dynamically load version 6.1.0.580 of `Symyx.Notebook.dll` using the provided public key token.
 - Instantiate and return an object of type `Symyx.Notebook.Sections.Background`.

When an application is not online, the application is only allowed to request types in assemblies found in the local store. Attempting to retrieve an instance that is not in the local store, while the server is offline, throws an `AssemblyNotFoundException`.

Assemblies are stored in a private location on the local disk, using a hierarchical version naming scheme similar to that used by the Global Assembly Cache.

If you published an assembly to Vault, you can use the Assembly Cache to retrieve that assembly and use it in your implementation. This is useful when you want to invoke a third-party assembly from an IronPython script that you use with Workbook. The `GetAssembly` method of `Symyx.Framework.Extensibility.AssemblyCache` gets the specified assembly from the Symyx Assembly Cache. If necessary, the `GetAssembly` method downloads the assembly from Vault.

The following IronPython example gets an assembly from the Assembly Cache, and adds a reference to the cache so that a script can use the script's namespace:

```
import clr
clr.AddReference("mycompany.product.UI")
from Symyx.Framework.Extensibility
import AssemblyCache
assemblyName = "mycompany.product.UI, Version=1.0, Culture=neutral,
PublicKeyToken=5779810541ea1fbe"
assembly = AssemblyCache.GetAssembly(assemblyName)
clr.AddReference(assembly)
from mycompany.product.UI import *
```

The `AssemblyCache` also provides `CreateInstance` methods for creating an instance of specified types. The following C# example shows how to create an instance of the sample assembly used in the previous example:

```
string typeName = @"mycompany.product.UI, Version=1.0, Culture=neutral,
PublicKeyToken=5779810541ea1fbe";
object instance = AssemblyCache.CreateInstance<object>(typeName);
```

Release Memory

If you subclass `VaultObject`, `VaultElement`, `VaultContainer`, `Document`, `DocumentSection`, or any other Framework or Workbook object subclass, then to ensure unused memory is properly released:

Implement the `IDisposable` interface as indicated by the Microsoft guidelines.

Call the `Symyx.Framework.Vault.VaultObject.DeleteAllObjects()` from within the `Disposing(true)` call; illustrated in the following C# examples.

The following `Foo` class implements the `IDisposable` interface:

```
public class Foo : VaultObject, IDisposable
{
    public Foo() : base(VaultObjectType.User)
    {
        // additional constructor code
    }
    #region IDisposable members
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
    #endregion
    ...
}
```

For a sealed class, implement the `Dispose` method as follows:

```
private void Dispose(bool disposing)
{
    if (disposing)
    {
        // dispose of all managed objects
        DeleteAllObjects();
    }
}
```

For a virtual class, implement the `Dispose` method as follows:

```
protected virtual void Dispose(bool disposing)
{
    if (disposing)
    {
        // dispose of all managed objects
        DeleteAllObjects();
    }
}
```

The following `Bar` class overrides the base `Dispose` method in the `Foo` class. The `Bar.Dispose` method disposes the managed object named `myObject` and then calls the base `Dispose` method in `Foo`:

```
public class Bar : Foo
{
    // create an object (this is for illustration purposes only)
    Bar myObject = new Bar();
    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            myObject.Dispose();
        }
        base.Dispose(disposing);
    }
}
```

```

    {
        // dispose of all managed objects
        myObject = null;>
    }
    // apply disposal to the parent class
    base.Dispose(disposing);
}
}

```

Omit Vault Object Content Compression

The Framework normally compresses content after the Vault objects have been written to the stream using the `WriteContent(Stream stream)` method. However, content compression can be omitted by overriding the `OmitContentCompression` property in a derived Vault object class:

```

public override bool OmitContentCompression
{
    get { return true; }
}

```

This indicates that content compression can be omitted when saving the content for this instance. You should generally omit compression if the data is already compressed or if the amount of data is very small, less than 100 bytes.

XML type content such as XML-based serializers like the `DataContractSerializer` benefit greatly from compression, so you should always leave them compressed.

Prevent Concurrent Updates

To protect Vault objects from concurrent updates, you must check out objects from Vault. This means that if an object is checked out by a user, modification to that object can only be performed by that user.

However, a check out is not required to update an object. For example, the Administration tool allows edits to users and groups which are directly saved against Vault. This gives application developers either option as needed.

Debug the Framework

See the `Symyx.Notebook.Application.exe.config` file for information on how to enable log4net for debugging.

Debug a Remote Service

Use the remote debugging tools for Visual Studio to debug a Vault service. For more information and to download the tools, see [Remote Debugging](http://msdn.microsoft.com/en-us/library/bt727f1t.aspx) (<http://msdn.microsoft.com/en-us/library/bt727f1t.aspx>).

WCF Tracing for Vault Diagnostics

You can enable Windows Communication Foundation (WCF) tracing to use Vault system diagnostics. The output log file from WCF tracing can contain details about authentication errors. You can activate WCF tracing by editing the `Symyx.Notebook.Application.exe.config` file on the client or the `web.config` file on the Vault Server computer. In the configuration section, add a `system.diagnostics` element such as the following:

```

<system.diagnostics>
  <sources>

```

```

<source name="System.ServiceModel" switchValue="All">
  <listeners>
    <add name="traceListener"
          type="System.Diagnostics.XmlWriterTraceListener"
          initializeData= "c:\log\wcfVault.svclog" />
  </listeners>
</source>
</sources>
</system.diagnostics>

```

For more information, see [the Microsoft documentation for Configure Tracing](#).

Script From External Assemblies

The Python script editor embedded within Workbook works for creating small scripts that are less likely to change. For larger scripts that are more likely to require enhancements and maintenance over time, separate and encapsulate the custom code in to a .NET DLL or assembly.

When you use an external .NET assembly with your script, you can:

- Use the rich editing capabilities of the Integrated Development Environments (IDEs) such as Microsoft Visual Studio.

There are free and inexpensive IDEs available that you can use for editing and debugging the scripts, using any of the programming languages supported by .NET, including C# and VB.NET.

- Avoid updating scripts in older documents.

If you update a script on a document template, only documents created after the updates contain the updated script. Existing documents continue to use the older script. New documents created by cloning old documents continue to use the older script. If your script invokes a .NET assembly, the most recent version of the assembly is used. By updating and re-distributing the assembly, you can propagate code changes immediately to all users, and for all documents including existing documents, allowing you to retire the old code.

- Deploy custom code more efficiently.

Use the built-in capability of Workbook to publish .NET assemblies to the Vault server, and to deliver the assemblies to the client computers, without any manual intervention.

- Simplify a potentially complex script into a simple script.

A script that uses a .NET assembly only needs to load the assembly and create an instance of its custom object in one method call, and then call the custom object's methods for processing.

Use an External .NET Assembly

To use an external .NET assembly in your script, call the `CreateInstanceFromLatestAssembly` method of the `AssemblyCache` class. The `AssemblyCache` class is in the `Symyx.Framework.Extensibility` namespace which is included in the `Symyx.Framework.dll`.

The following example shows a script that uses an external .NET assembly:

```

# Import AssemblyCache
from Symyx.Framework.Extensibility import AssemblyCache
# Create the .NET object containing your custom code.
my_object = AssemblyCache.CreateInstanceFromLatestAssembly
               ("CompanyName.ProjectName.ClassName", CompanyName.ProjectName")
# Pass Notebook script objects to the "MethodName" on

```

```
# the custom object
my_object.MethodName(active_workspace, editor, sender, e)
```

CreateInstanceFromLatestAssembly Method

The `AssemblyCache` provides services related to retrieving assemblies from Vault, downloading them to the local computer, and creating objects from them. The `CreateInstanceFromLatestAssembly` method combines these services:

- If the assembly has been used during the user's current session, it uses the currently loaded assembly.
- If the assembly exists in the Workbook working directory, the assembly is used.
- If the assembly has not been loaded, it quickly searches Vault to determine the latest version of the assembly. In offline mode, the latest version of the assembly that has been previously downloaded from Vault is used.
- If the latest version of the assembly has not been downloaded, the method downloads it from Vault to the local computer.

In offline mode, this step is skipped.

- Loads the assembly, creates an object from it, and returns the object.

The `AssemblyCache.CreateInstanceFromLatestAssembly` method accepts a string containing a partially qualified type name, for example, `CompanyName.ProjectName.ClassName`, `CompanyName.ProjectName`. The string used in the preceding example specifies that the assembly to be searched and loaded is `CompanyName.ProjectName.dll`, and the object to be created is `CompanyName.ProjectName.ClassName`.

The string passed to the `AssemblyCache.CreateInstanceFromLatestAssembly` method is partially qualified because it does not include all of the information needed for .NET to identify the assembly. The method does not specify which version of the assembly to use because the method determines which version is the latest and gets it from Vault.

Create Custom .NET Assembly

1. Acquire access to `VaultADMStoreManager.exe` or ask your system administrator to use the `VaultADMStoreManager.exe` to download the Workbook Client package.
2. Get the list of profiles using the following command:


```
VaultADMStoreManager /vault <server.Domain>
<Domain\User> Password list-profiles
```

Example 1

```
F:\Downloads\SN6.7 installers & Documentation\SymyxNotebook6.7_
SP1\sp2\VaultDeploymentUtility\VaultADMStoreManager /vault vm-avs66 vm-
avs66\vault.admin "" list-profiles
Password: *****
V6.x.0.744-Offline
V6.x.0.744-Offline-PerfLog
V6.x.0.744-Roaming
[...]
```

Select the profile you want to use for your working directory Command:

```
>VaultADMStoreManager /vault
<server.Domain>
```

```
<Domain\User> Password generate  
<Profile Name>  
<Folder Name>
```

Example 2

```
F:\Downloads\Workbook installers & Documentation\Workbook  
\2017\VaultDeploymentUtility\VaultADMStoreManager>VaultADMStoreManager  
/vault vm-avs66 vm-avs66\vault.admin "" generate 2017-Citrix  
"f:\temp\2017-Citrix"  
Password: *****
```

Create a .NET project

Choose an IDE to create your .NET project. There are free or low cost .NET IDEs available that are capable of compiling code written in different programming languages. The Microsoft Visual Studio Express editions are free versions of Visual Studio capable of compiling code written in Visual Basic.NET (VB.NET) or C#.

Use an IDE that supports debugging external applications. Microsoft Visual Studio Express does not support debugging. While you can use any .NET language or IDE, for the purposes of this tutorial the VB.NET in Visual Studio, and C# in #develop are used.

For information about Visual Studio, see <http://www.microsoft.com/visualstudio/en-us/products/#develop>.

If you are new to .NET programming, take some time to learn the basics of your IDE, such as how to create projects, edit source files, and compile. In the rest of this topic, we assume knowledge of these basics, and discuss some of the key concepts necessary for programming with the Workbook SDK.

If the .NET Framework 3.5.2 or 4.5.1 is not set, the IDE ignores references to Workbook DLLs leading to *type not found* compilation errors wherever Workbook classes are used. If you forgot to reference the Workbook DLLs, you can change it later from the Project Properties screen.

If you are creating an EXE assembly, in Advanced Compiler Settings, set the Target CPU to *x86* to direct the compiler to create a 32-bit EXE. On 64-bit operating systems, the IDE might set the value to *Any CPU* by default causing the build target to compile as a 64-bit executable. Workbook DLLs are 32-bit binaries, building a 64-bit executable could cause compilation or run time errors.

Sign Your Assembly

You should sign the assembly by creating a new strong name key file or by using an existing one. Some organizations have a single strong name key file that is used for all assemblies and applications. Do not check the Delay sign only option.

Writing Classes and Methods

Naming Conventions

When naming your .NET assembly, Microsoft recommends that you follow the naming convention `CompanyName.ProjectName.dll`, with as many qualifiers as necessary. For example, if the custom assembly is for the Analytical Workbook project at Acme Pharmaceutical Company, name the assembly `Acme.ELN.dll` or `Acme.ELN.Analytical.dll`.

Microsoft also recommends that the first part of a namespace matches the assembly name. For example, an assembly named `Acme.ELN.dll` should contain namespaces like `Acme.ELN.ImportUtilities` or `Acme.ELN.Import.Utilities`. An assembly named

Acme.ELN.Analytical.dll would contain namespaces like
Acme.ELN.Analytical.ImportUtilities or Acme.ELN.Analytical.Import.Utilities.

Adding references to Workbook assemblies

In your .NET project, add references to the Workbook assemblies your code needs. You need to reference the Symyx.Framework.dll and Symyx.Notebook.dll. Some assemblies called from PropertySet scripts might only require the Symyx.Framework.dll. If you require other references such as Symyx.Windows.dll, the IDE provides hints about additional requirements. The Workbook assemblies are included in the BIOVIA Workbook SDK.

To work with sections in your .NET assembly, see [BIOVIA Workbook Sections](#).

Define a Class

In your .NET project, add the classes that contain your custom code implementation. You can use any namespaces, class names, method names, and assembly names that meet your requirements.

Your project must contain a class to instantiate in a Workbook script. You must contain the class in an assembly that is loaded from the Workbook script.

The class name and partially qualified assembly name must be passed to the AssemblyCache.CreateInstanceFromLatestAssembly method which is invoked by the Workbook script.

The following shows the sample Workbook script that calls the AssemblyCache.CreateInstanceFromLatestAssembly method.

```
my_object = AssemblyCache.CreateInstanceFromLatestAssembly
    ("CompanyName.ProjectName.ClassName, CompanyName.ProjectName")
my_object.MethodName(active_workspace, editor, sender, e)
```

The following example shows a sample Visual Basic .NET code that implements the CompanyName.Project.ClassName class.

```
Imports Symyx.Notebook.ApplicationManagement
Imports System.Windows.Forms
Imports Symyx.Notebook.Vault
Public Class ClassName
    Public Sub MethodName(activeworkspace As NotebookWorkspace,
        editor As IDocumentEditor, sender As Object, e As EventArgs)
        MessageBox.Show(editor.Title)
    End Sub
End Class
```

Use document.Title to refer to the title of a document. Use editor.Title to refer to the title bar of a window.

The following shows a sample C# code that implements the CompanyName.Project.ClassName class:

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
using Symyx.Notebook.ApplicationManagement;
using Symyx.Notebook.Vault;
namespace CompanyName.ProjectName
{
    public class ClassName
```

```
{
    public void MethodName(Notebookworkspace activeworkspace,
        IDocumentEditor editor, Object sender, EventArgs e)
    {
        MessageBox.Show(editor.Title);
    }
}
```

In Visual Basic .NET, the root namespace is not shown in the code window as it is in C#. You view root namespace from the Application tab on the Project Properties window.

Define a Method

When defining a method to call from a Workbook script, include the Workbook script variables as method parameters. You must also include the objects that are not used by the method. For example, define `activeWorkspace` as a method parameter even if the method does not use it. If you don't include it in your method parameters, you cannot use the parameter with existing documents in the future.

Different script variables are available to the method depending on the event that is captured. For a list of event-specific script variables, see [Scripting Variables](#).

The following example shows a method signature for a toolbar button script.

In Visual Basic .NET:

```
Imports Symyx.Notebook.ApplicationManagement
Imports System.Windows.Forms
Imports Symyx.Notebook.Vault
Public Sub MethodName(activeworkspace As Notebookworkspace,
    editor As IDocumentEditor, sender As Object, e As EventArgs)
```

In C#:

```
using System;
using System.Windows.Forms;
using Symyx.Notebook.ApplicationManagement;
using Symyx.Notebook.Vault;
public void MethodName(Notebookworkspace activeworkspace,
    IDocumentEditor editor, Object sender, EventArgs e)
```

If you need to use a Workbook section in your method, see the [Sections](#) for information about referencing assemblies containing Workbook sections and for an example of a method that uses the File section.

Chapter 9:

Sections

To work with Workbook sections in your .NET assembly, use the supported interfaces implemented by the Workbook sections. The supported interfaces are contained in the `Symyx.Framework.dll`, `Symyx.Framework.*.dll`, or `Symyx.Notebook.dll`, `Symyx.Notebook.*.dll`, or in the `Symyx.Notebook.Sections.<Section Name>.Extensibility.dll` assemblies. In your .NET project, add references to the appropriate assemblies.

Do not add references to section assemblies in your .NET assembly. Section assemblies refer to the following assemblies, which you cannot reference:

- `Symyx.Notebook.ImageAnnotation.dll`
- `Symyx.Notebook.Sections.Equipment.dll`
- `Symyx.Notebook.Sections.ExternalFile.dll`
- `Symyx.Notebook.Sections.Formulation.dll`
- `Symyx.Notebook.Sections.Forms.dll`
- `Symyx.Notebook.Sections.Materials.dll`
- `Symyx.Notebook.Sections.ReactionScheme.dll`
- `Symyx.Notebook.Sections.Reference.dll`
- `Symyx.Notebook.Sections.SamplePreparation.dll`
- `Symyx.Notebook.Sections.StructuredRecipe.dll`
- `Symyx.Notebook.Sections.Table.dll`
- `Symyx.Notebook.Sections.Text.dll`

To prevent any possibility of changing existing experiment data, newer versions of section assemblies generally do not replace older versions. Your code might therefore have to execute against multiple, different versions of these assemblies. But if you reference one of them directly, your code is permanently bound to that version of the section. If so, attempting to run against other versions will generate type mismatch errors.

Instead of referencing a version directly, access a Workbook section using the properties and methods of its base class and implemented interfaces, which are defined in assemblies that you can reference.

The following are the assemblies that you can reference from your .NET assembly:

- `Symyx.Framework.Controls.dll`
- `Symyx.Framework.dll`
- `Symyx.Framework.EquationParser.dll`
- `Symyx.Framework.MaterialInfoLookup.dll`
- `Symyx.Framework.MaterialInfoLookup.XmlSerializers.dll`
- `Symyx.Framework.Materials.dll`
- `Symyx.Framework.Quantity.dll`
- `Symyx.Framework.RAS.dll`
- `Symyx.Framework.Reporting.dll`
- `Symyx.Notebook.AnalyticalMaterials.dll`
- `Symyx.Notebook.AnalyticalMaterialsContracts.dll`

- Symyx.Notebook.Applications.FormEditor.dll
- Symyx.Notebook.dll
- Symyx.Notebook.Sections.ExternalFile.Extensibility.dll
- Symyx.Notebook.Sections.Text.Extensibility.dll
- Symyx.PipelinePilot.dll
- Symyx.Windows.dll

The `IEternalFileSection` is the main interface defined in `Symyx.Notebook.Sections.ExternalFile.Extensibility.dll` implemented by the file section. In your assembly, add a reference to `Symyx.Notebook.Sections.ExternalFile.Extensibility.dll` and access the file section as shown in the following example.

Visual Basic .NET:

```
Imports Symyx.Notebook.ApplicationManagement
Imports Symyx.Notebook.Vault
Imports Symyx.Notebook.Sections.ExternalFile.Extensibility

Public Class ClassName

    Public Function MethodName(ByVal activeworkspace As NotebookWorkspace,
        ByVal editor As IDocumentEditor, ByVal sender As Object,
        ByVal e As EventArgs) As String

        Dim fileSection As IEternalFileSection = TryCast
            (editor.ActiveDocumentSection, IEternalFileSection)
        If Not fileSection Is Nothing Then
            Dim fileName As String = "C:\Temp\Test.pdf"
            fileSection.AddFile(fileName)
            Return fileName End If

        Return "ActiveDocumentSection does not implement IEternalFileSection"
    End Function
End Class
```

C# Example:

```
using System;
using Symyx.Notebook.ApplicationManagement;
using Symyx.Notebook.Vault;
using Symyx.Notebook.Sections.ExternalFile.Extensibility;

namespace CompanyName.ProjectName
{
    public class ClassName
    {
        public string MethodName(NotebookWorkspace activeworkspace,
            IDocumentEditor editor, object sender, EventArgs e)
        {
            var fileSection = editor.ActiveDocumentSection as
                IEternalFileSection;
            if (fileSection != null)
            {

```

```

        var fileName = @"C:\Temp\Test.pdf";
        fileSection.AddFile(fileName); return fileName;
    }
    return "ActiveDocumentSection does not implement
        IExternalFileSection";
}
}
}

```

For example, if you need to use the `TableSection` in the `Symyx.Notebook.Sections.Table.dll` assembly, see its class description in the API Reference.

The `TableSection` inherits from the base class `DocumentSection` and implements the interfaces including `IEditable`, `IIndexableText`, `ICHILDData`, and `IReportable`. You can access the `TableSection` using the public properties and methods of its base class and implemented interfaces, which are defined in assemblies that you can reference.

The summary list of `TableSection` members in the API Reference shows which methods and properties are inherited from the supported base class and interfaces, as denoted by `Inherited from ClassOrInterface`.

Currently, not all public properties and methods of the `TableSection` are accessible through interfaces, so you cannot use them in your .NET assembly. However, you can use them in your Python script.

Clone an Experiment to the Latest Template Version

To take advantage of the latest scripts, sections, and template properties, you can use `clone to latest` to update experiments that use older template versions. `Clone to latest` is available in Workbook version 6.8 and later.

Experiments created using earlier versions have parent templates. You can:

- Update the relevant sections in the parent template to the latest version of the template.
- Check out the experiment template, for example, a version 6.4 template.
- Update all document sections with the newest versions.
- Configure the new sections including scripts, mapping, toolbars, and conditional formatting.
- Expand the set of insertable sections to include newest versions.
- Delete unused sections.
- Users can create child experiments by cloning sections from the source experiments.

Sections in a New Document

- Get sections that are in both the Source Experiment and the Parent Template matched by Name and section type
- Get unmatched sections that are only in the Source Experiment
- Get unmatched sections that are only in the Parent Template
- Get the set of insertable sections from the Parent Template Each section has a `No Clone` property value.
- If the Parent Template property value = `true`, the data comes from the Parent Template
- If the Parent Template property value = `false`, the data comes from the Source Experiment

The Source Experiment `No Clone` property value is not relevant if `false` and:

- Allow Clone Without Data = true, the user can choose to include data or not
- Allow Clone Without Data = false, the user must include the data

If a section has been inserted into the Source Experiment, the user gets the option Do not Include Section.

Forms and Tables

A Form (SN. form) could have a Parent Template. If it is, the new experiment or clone gets the version of the Form that is in the Parent Template. If it is not, the clone gets the newest version of the Form.

Each form widget has a Cloneable property value:

- If false in either the Parent Template or Source Experiment, the data comes from the Parent Template.
- If true in both Parent Template and Source Experiment, the data comes from the Source Experiment.

Table Sections: The new document maintains the Source Experiment column display settings ONLY IF the user chooses to include the section data by using the following options:

- Pinned columns
- Selected set of columns to display
- Right to left position of the columns
- Ordered or Sorted mode

Insert Forms

This C# snippet shows how to insert a Form into the FormsSectionView.

```
using Symyx.Notebook.Sections.Forms;
namespace Symyx.Notebook.Examples
{
    public class FormsSectionViewExamples
    {
        public static void InsertForm(Form form, FormsSection formsSection)
        {
            var view = formsSection.View as FormsSectionView;
            view.InsertForm(form.VaultId, form.Version, form.Title);
        }
    }
}
```

Import Forms

This C# example shows how to import a form from a form (.snform) file and add it to a repository.

```
using System.IO;
using Symyx.Framework.Vault;
using Symyx.Notebook.Applications.FormEditor;
namespace Symyx.Notebook.Examples
{
    public class FormEditorExamples
    {
        public static Form ImportForm(string path, Folder folder,
            string title)
        {

```

```

    var form = new Form
    {
        Title = title,
        Definition = File.ReadAllText(path)
    };
    var formEditor = new FormEditor(form, false, false);
    formEditor.UpdateFormData();
    var repository = VaultWorkspace.Current.GetRepository
        (folder.SourceRepositoryId);
    repository.Add(form, folder);
    return form;
}
}
}

```

Populate Form Controls

The `samples\Symyx.SDK.Samples.DataCreation` example is a C# project that also works with form controls for a synthetic chemistry section.

The `PopulateFormsSection` method sets synthetic information.

```

var syntheticInformation = backgroundForm.Controls
    ["SythenticInformation"].Controls;
syntheticInformation["textCompoundID"].Text = "SMMX-120998";
syntheticInformation["comboReaction"].Text = "Alkylation";
return formsSection;

```

Populate a Form Section

The source code file `controls` continues with the following code.

```

public static FormsSection PopulateFormsSection(Document document)
{
    var formsSection = FindSingleSectionByTitle(document,
        FORMS_SECTION_TITLE) as FormsSection;
    if(formsSection == null)
    {
        formsSection = new FormsSection {Title = FORMS_SECTION_TITLE};
        formsSection.ConvertToTemplate();
        Repository.Add(formsSection, Folder);
        formsSection = Repository.Get(formsSection.VaultId, DataScope.All)
            as FormsSection;
        document.Add(DocumentSection.Create(formsSection));
        formsSection = FindSingleSectionByTitle(document,
            FORMS_SECTION_TITLE) as FormsSection;
    }
    var view = formsSection.View as FormsSectionView;
    view.InsertForm(Form.VaultId, Form.Version, Form.Title);
    var backgroundForm = Iterator.Find
        (Iterator.Cast<BaseForm>(view.Forms), f=> f.Title == Form.Title);
    var corporateInformation = backgroundForm.Controls
        ["CorporateInformation"].Controls;
    corporateInformation["textboxTitle"].Text =
        "SMMX-1210998 Resynthesis";
    corporateInformation["comboName"].Text = "Symyx 1";
}

```

```

corporateInformation["comboDepartment"].Text = "Medicinal Chemistry";
corporateInformation["comboSite"].Text = "Camberley";
corporateInformation["textSummary"].Text = "Resynthesize SMMX-120998
    on 150 mg scale.Need material for assays.";

(corporateInformation["typeResearch"] as CheckBox).Checked = true;
(corporateInformation["typeGXP"] as CheckBox).Checked = false;
(corporateInformation["typeOutsource"] as CheckBox).Checked = false;
(corporateInformation["typeOther"] as CheckBox).Checked = false;

var syntheticInformation = backgroundForm.Controls
    ["SytheticInformation"].Controls;
syntheticInformation["textCompoundID"].Text = "SMMX-120998";
syntheticInformation["comboReaction"].Text = "Alkylation";
return formsSection;
}

```

The form is imported from the Resource\SymyxForm.snform file. You can inspect the form in Workbook using the Form Editor.

To import a form:

1. In Workbook, click the Notebook Explorer tab.
2. From **Create > New Form**.
3. From **Create > Import Form** to import the .snform file.
4. Verify that in the Import Form window, the name SymyxForm.snform is listed.

Populate a List Using Vault Vocabulary

Use code similar to the following to automatically update the values in list elements. The following IronPython example is an onEdit script that ensures that every time the user starts editing the form, the vocabulary is refreshed.

```

import clr
clr.AddReference('Symyx.Framework')
from Symyx.Framework import Vault
cboProtocol = ActiveForm.Controls["gbxExptSummary"].Controls["cboProtocol"]

# Find the vocabulary
vname = "Elements"
workspace = Vault.VaultWorkspace.Current
vocabularies = workspace.SiteRepository.Get
    (Vault.VaultObjectTypes.Vocabulary, Vault.DataScope.All)
voc1 = vocabularies.FindByTitle(vname)
if voc1 != None:
    # clear the combo box
    cboProtocol.Items.Clear()
    # Now add in the new vocabulary
    vocPhrase = voc1.Phrases
    for vs in voc1.Phrases:
        if vs is not None:
            cboProtocol.Items.Add(vs)

```

Form Examples

In the API Reference, the following items have sample code related to populating a form:

- Namespaces > Symyx.Notebook.Applications.FormEditor > FormEditor > FormEditor Constructor (Form, Boolean, Boolean)
- Namespaces > Symyx.Notebook.Applications.FormEditor > FormEditor > UpdateFormData()
- Namespaces > Symyx.Notebook.Sections.Forms > FormsSectionView > InsertForm(VaultId, Version, string)
- Namespaces > Symyx.Notebook.Sections.Forms.Widgets > CheckBox
- Namespaces > Symyx.Notebook.Sections.Forms.Widgets > ComboBox
- Namespaces > Symyx.Notebook.Sections.Forms.Widgets > GroupBox
- Namespaces > Symyx.Notebook.Sections.Forms.Widgets > TextBox

Populate Widgets in Forms

The following C# example shows how to populate widgets in a form.

```
using Symyx.Framework.Collections;
using Symyx.Notebook.Forms;
using Symyx.Notebook.Sections.Forms;
using Symyx.Notebook.Sections.Forms.Widgets;
namespace Symyx.Notebook.Examples
{
    public class FormsWithWidgetsExamples
    {
        public static BaseForm PopulateWidgets(FormsSectionView view,
            string formTitle)
        {
            // Find the form in view by Title
            var form = Iterator.Find(Iterator.Cast<BaseForm>(view.Forms),
                f=> f.Title == formTitle);
            // Get the child controls of CorporateInformation
            var corporateInformation = form.Controls
                ["CorporateInformation"].Controls;
            corporateInformation["textboxTitle"].Text = "SMMX-1210998
                Resynthesis";
            corporateInformation["comboName"].Text = "Symyx 1";
            corporateInformation["comboDepartment"].Text = "Medicinal
                Chemistry";
            corporateInformation["comboSite"].Text = "Camberley";
            corporateInformation["textSummary"].Text = "Resynthesize
                SMMX-120998 on 150 mg scale. Need material for assays.";
            (corporateInformation["typeResearch"] as CheckBox).Checked = true;
            (corporateInformation["typeGXP"] as CheckBox).Checked = false;
            (corporateInformation["typeOutsource"] as CheckBox).Checked = false;
            (corporateInformation["typeOther"] as CheckBox).Checked = false;
            // Get the child controls of SythenticInformation var
            syntheticInformation = form.Controls
                ["SythenticInformation"].Controls;
            syntheticInformation["textCompoundID"].Text = "SMMX-120998";
            syntheticInformation["comboReaction"].Text = "Alkylation";
            return form;
        }
    }
}
```

References

The new document gets all references whether they exist in the source experiment or the parent template.

Property Set Definitions

Property	Variable	Description
Cloneable		
	Allowed	Permits cloning the data.
	NotAllowed	Restricts the ability to clone the data in the source experiment.
	AllowedNotData	Permits null values in the property setting.
AllowNulls		
	AllowNulls	Permits null values in the property.
	CannotBeNull	Requires a value in the property; if the user tries to delete data and check in, the original data displays in the cell when the file is checked out.
	ShouldNotBeNull	Requires user input, a red x renders in the experiment until user fills in a value.
Allow Updates		
	Always	always allows updates
	Never	never allows updates
	Once	Allows cloning the data one time in an experiment that contains properties with this setting; updating the data in these properties is not allowed. <i>Once</i> refers to the original data in the source experiment prior to saving.
	Until Saved	Allows updates until saved.
	Until Managed	Allows updates until the experiment is checked into a managed repository.

Limitations

The following versions are supported in versions 6.5 and later.

- Clone to latest does not support a section being both upgraded to a newer version and renamed.
- Reaction scheme sections that were previous un-linked cannot be cloned into a linked reaction scheme section.
- If the parent template had linked sections such as synthetic chemistry linked to parallel chemistry, you cannot clone to latest if you remove the link. In addition, after cloning to latest, you must relink those sections.

Clone to Latest Limitations

Clone to latest does not support upgrading and renaming a section to a newer version.

You cannot link previous unlinked reaction scheme sections and clone those sections to a linked reaction scheme section.

If the parent template had linked sections such as synthetic chemistry linked to parallel chemistry, you cannot clone to latest if you remove the link. In addition, after cloning to latest, you must relink those sections.

File Sections

An `ExternalFileSection` permits users to include formatted text within a Document. The following C# snippet shows how to add an external file section to a document.

```
public static ExternalFileSection AddExternalFileSectionToDocument
(Document document)
{
    ExternalFileSection externalFileSection = new ExternalFileSection();
    document.Add(externalFileSection);
    return externalFileSection;
}
```

The following C# example is more complete, and shows adding a file section to a document and then manipulating the section's file content. The example is available in the `Symyx.Notebook.Sections.ExternalFile.InsertFile` method.

```
using System; using System.IO;
using Symyx.Notebook;
using Symyx.Notebook.Examples.Resources;
using Symyx.Notebook.Sections.ExternalFile;
using Symyx.Notebook.Sections.ExternalFile.Extensibility;
using System.Reflection;
```

```
namespace Symyx.Notebook.Examples.FileSection
{
    public static class ExternalFileSectionExample
    {
        public static Document CreateExampleDocument()
        {
            Document document = CreateBlankDocument();
            ExternalFileSection externalFileSection =
                AddExternalFileSectionToDocument(document);
            AddFilesToExternalFileSection(externalFileSection);
            return document;
        }
        public static Document CreateBlankDocument()
        {
            Document documentTemplate = new Document();
            return Document.Create(documentTemplate);
        }
        public static ExternalFileSection AddExternalFileSectionToDocument
            (Document document)
        {

```

```
ExternalFileSection externalFileSection = new ExternalFileSection();
document.Add(externalFileSection);
return externalFileSection;
}
public static void AddFilesToExternalFileSection
    (ExternalFileSection externalFileSection)
{
    externalFileSection.AddFile(ImageFilePath);
    externalFileSection.AddFile(PdfFilePath);
    externalFileSection.AddFile(WordFilePath);
    externalFileSection.AddFile(ExcelFilePath);
    externalFileSection.AddFile(ExcelFilePath);
    // Remove last file by index.
    externalFileSection.RemoveFile(externalFileSection.Files.Count - 1);
    // Insert a file at the beginning.
    ExternalFilePackage filePackage = externalFileSection.InsertFile
        (0, RichTextFilePath);
    // Remove first file by package.
    externalFileSection.RemoveFile(filePackage);
}
static string ImageFilePath = EmbeddedResourceManager.GetResourcePath
    ("MagT.emf");
static string PdfFilePath = EmbeddedResourceManager.GetResourcePath
    ("PMPA diester - resynthesis.pdf");
static string WordFilePath = EmbeddedResourceManager.GetResourcePath
    ("test.doc");
static string ExcelFilePath = EmbeddedResourceManager.GetResourcePath
    ("Chart.xls");
static string RichTextFilePath =
    EmbeddedResourceManager.GetResourcePath("varied.rtf");
}
}
```

Add and Remove Files

The following C# examples show how to add, insert, and remove files.

To add a file, use the `Symyx.Notebook.Sections.ExternalFile.AddFile` method.

```
externalFileSection.(ExcelFilePath);
```

To insert a file, use the `Symyx.Notebook.Sections.ExternalFile.InsertFile` method.

```
// insert file at beginning of package
ExternalFilePackage filePackage =
    externalFileSection.InsertFile(0, RichTextFilePath);
```

To remove a file, use the `Symyx.Notebook.Sections.ExternalFile.RemoveFile` method.

```
// remove by package
externalFileSection.RemoveFile(filePackage);
```

You can also use:

```
// remove by index
externalFileSection.RemoveFile(externalFileSection.Files.Count - 1);
```

If you require a signature policy for actions in a file section, setting the `FileActionSignature` property displays a signature dialog for the user to complete before the action is saved.

Visualizations

To enable a computer to generate visualizations for the File Section:

- Install the utility that calls the `Symyx.Notebook.Sections.ExternalFile.Extensibility.ExternalFilePackage.GenerateVisualization` method.

Install the Workbook client that provides the BlackIce Printer driver as a Windows printer.

Ensure that the DynaPDF library, `dynapdf.dll` is present. Do one of the following:

- Execute the utility from Workbook's bin directory.
- Copy the `dynapdf.dll` to the directory with the utility executable.

Preview of a File

A visualization is a preview of the contents of a file. The namespace, `Symyx.Notebook.Sections.ExternalFile`, supports the generation of a visualization through a signature of the `AddFile` method in the `ExternalFileSection` class of the `Symyx.Notebook.Sections.ExternalFile` namespace:

```
ExternalFileSection.AddFile(String, Boolean)
```

Where `String` represents the path of the file to add, and `Boolean`, if true, generates a visualization for the file. For example,

```
ExternalFileSection.AddFile("c:\test\myFile.xls", true);
```

Preview Multiple Files

If you are adding multiple files to be visualized, add all of them by passing false as the value to the argument that determines whether to generate a visualization.

Generate visualizations for all the files by calling `GenerateVisualization()` for each `ExternalFilePackage` in `ExternalFileSection.Files`.

Checking for the Existence of Visualizations

The `ExternalFilePackage.VisualizationStates` enumeration in the `Symyx.Notebook.Sections.ExternalFile.Extensibility` namespace describes the possible states of a package's visualization such as the following:

- Unknown
- NoVisualizationSource
- NoVisualizationSourceSelected
- Present
- GenerationNotAttempted
- GenerationPreviouslyUnsuccessful

Required Software for Visualizations Utility

To enable a computer to generate visualizations for the File Section:

1. Install your utility that calls the `Symyx.Notebook.Sections.ExternalFile.Extensibility.ExternalFilePackage.GenerateVisualization` method.
2. Install the Workbook client that provides the BlackIce Printer driver as a Windows printer.
3. Verify the presence of the DynaPDF library, `dynapdf.dll` by doing one of the following:
 - a. Execute the utility from Workbook's bin directory.
 - b. Copy the `dynapdf.dll` to the directory containing the utility's executable file.

Create File Section with Table Rows

The following IronPython example shows how to:

- Get selected rows from a table
- Create a File section for each selected row in the table

```
from System.Windows.Forms import MessageBox
from Symyx.Framework.Vault import VaultWorkspace
from Symyx.Framework.Vault import VaultObjectTypes
from Symyx.Framework.Vault import DataScope

# Get the active section (table)
activeSection = editor.ActiveDocumentSection
# Get the currently selected rows from the table
selectedRows = activeSection.GetSelectedRows()
# Get the current experiment document
ActiveDocument = editor.Document

# Find the template for the external file section
SectionTemplates = active_workspace.Current.SiteRepository.Get
    (VaultObjectTypes.DocumentSectionTemplate, DataScope.All)
for SectionTemplate in SectionTemplates : if
    SectionTemplate.Title == "File" :
        neededSection = SectionTemplate

# For each selected row create an external file section for row
in selectedRows :
    # use the sample id field and the test type as the new section title
    sampleid = row.PropertySets["Samples"]["SampleID"].Value.ToString()
    test = row.PropertySets["Samples"]["Specification"].Value.ToString()

    newSectionTemplate = neededSection.Clone(False)
    newSectionTemplate.Title = sampleid + " - " + test;
    ActiveDocument.Add(newSectionTemplate)
editor.RefreshView()
```

List the Property Set Definitions for a Table

This C# example shows how to get a list of the property set definitions in a table section.

```
// setting up the test case TableSection
ts = new TableSection();
```

```

ICollection<PropertySetIdentifier> selectedPSDs =
ts.TableSectionProperties.GetValue<ICollection<PropertySetIdentifier>>
(TableSectionP roperty.SelectedPropertySetDefinitons);

selectedPSDs.Add(new PropertySetIdentifier("Test"));
selectedPSDs.Add(new PropertySetIdentifier("Test2"));

// listing psd's without knowing anything about them
foreach(PropertySetIdentifier identifier in selectedPSDs)
{
    // the identifier can identify a PSD by its Key, identifier.Key,
    // which is unique within a Vault Server
    PropertySetDefinition aSelectedPropertySetDef =
PropertySetManager.GetDefinition(identifier.Key);
    Debug.WriteLine("The table has columns corresponding to this PSD: " +
aSelectedPropertySetDef.Key);
    foreach(PropertyClass propertyClass in aSelectedPropertySetDef)
    {
        Debug.WriteLine("The table has a column corresponding to this
PropertyClass: " + propertyClass.DisplayName);
    }
}

```

List Values from a Table

The following IronPython script shows how to lists the values of table's rows.

```

from System import *
sectionSource = "Test Articles"
propertySetName = "TestArticles"
labelColName = "CompoundID"

# find Table section that contains the source data
dictionarySource = None
for section in table.Document.Sections: if
    section.Title == sectionSource:
        dictionarySource= section
        break
if (dictionarySource is not None) and (e.Property is not None):

    # only populate property CompoundID
    if e.Property.Key == labelColName:
        e.Items.Clear()

    for row in dictionarySource.Rows:
        if not row.PropertySets[propertySetName][labelColName].IsNull:
            value = row.PropertySets[propertySetName][labelColName].Value.ToString()
            e.Items.Add(value)

```

Invoke a Form and Add Rows

To add a row, use `table.AddRow`. The following IronPython example invokes the `Symyx.Notebook.Sections.Table.AddNewRowsForm` to prompt the user for the number of rows to add, and then uses `table.AddRow()` to add the rows.

```

from Symyx.Notebook.Sections.Table import AddNewRowsForm
from System.Windows.Forms import DialogResult
try:
    addNew = AddNewRowsForm()
    result = addNew.ShowDialog()
    if result == DialogResult.OK:
        i = 0
        while i < addNew.RowCount:
            table.AddRow()
            i = i + 1
finally:
    addNew.Dispose()

```

Tip: Turn off row re-sizing during insert to speed up multi-row operations.

Notes:

If you use this sample script for a custom toolbar item, owner is the only script variable that is available. The owner variable represents the Table Section where the dynamic toolbar is defined. To use this example in a custom toolbar script, insert the following line before table is first used:

```
table = owner
```

```
# global table is not available in toolbar item scripts table = owner
```

```
newRow = table.AddRow() newRow.PropertySets["Material"]["Name"].Value =
"New Material"
```

Set Values in a Table

The following C# example combines two property set definitions and sets a specified configuration.

```

TableSection section_ = new TableSection();
var newList = new List<PropertySetIdentifier>();

// The rows of the table in the table section might use
// properties from more than one property set definition.
newList.Add(new PropertySetIdentifier(psd1.Key));
newList.Add(new PropertySetIdentifier(psd2.Key));

// Use this signature: SetValue(PropertyKey, Object)
section_.TableSectionProperties.SetValue
(TableSectionProperty.SelectedPropertySetDefinitons, newList);

// config1 might specify which columns to show or hide and
// the order of columns
section_.TableSectionProperties.SetValue
(TableSectionProperty.Configuration, config1);

```

Add a Property Set Definition

The following C# example adds a property set definition to a TableSection.

```

var document = new Document();
var tableSection = new TableSection();
tableSection.Title = "Tabular Data";

```

```
document.Add(tableSection);

// most common: add a property set definition to the table section,
// is a visible part of the table
tableSection.TableSectionProperties
[TableSectionProperty.SelectedPropertySetDefinitions].Add(new
PropertySetIdentifier(MaterialPropertySets.Material));

// also possible: add a property set definition that is only visible
// as a property to the table within a table section
tableSection.PropertySetDefinitions.Add(new PropertySetDefinition());
```

Insert Rows

To add a row to a Table section using the .NET API, call the `Symyx.Notebook.Sections.Table.TableSection.AddRow` method.

For example:

```
public IPropertySetHost AddRow()
```

Disable row resizing during insert to speed up multi-row operations.

TableSection Script Variables

Event: Row Removed

Script variable	Description
e	<code>Symyx.Framework.Collections.ItemEventArgs<IPropertySetHost></code>
row	Represents the object to remove
sender	<code>Symyx.Notebook.Sections.Table.TableSection</code>

Event: After Record Weights

Script variable	Description
e	<code>System.EventArgs</code>
sender	<code>Symyx.Notebook.Sections.Table.TableSection</code>

Event: Before Record Weights

Script variable	Description
e	<code>Symyx.Notebook.Sections.Table.RecordWeightCancelEventArgs</code>
sender	<code>Symyx.Notebook.Sections.Table.TableSection</code>

Event: Request Column Dictionary

Script variable	Description
e	<code>Symyx.Framework.Properties.DictionaryEventArgs</code>
row	Represents the object containing the dictionary property
sender	<code>Symyx.Notebook.Sections.Table.TableSection</code>

Event: Row Added

Script variable	Description
e	<code>Symyx.Framework.Collections.ItemEventArgs<IPropertySetHost></code>
row	Represents the object that was added
sender	<code>Symyx.Notebook.Sections.Table.TableSection</code>

Event: Row Changed

Script variable	Description
e	<code>Symyx.Framework.Properties.ValueChangedEventArgs</code>
row	Represents the object with changed property
sender	<code>Symyx.Notebook.Sections.Table.TableSection</code>

Script with Table Section Properties

Each column in a Table Section corresponds to a property. A group of properties are defined in a property set definition. A Table Section can use one or more property set definitions. Because a table section can have multiple property sets to access a property value in a row, use a two-part index:

- The first key is the name of the property set.
- The second key is the name of the property.

The following example shows how to specify and update a property value in a row. The first index is a property set named, *Dilutions*, the second is a property named, *SampleName*.

```
row.PropertySets["Dilutions"]["SampleName"].Value = "test sample 145"
```

If a property belongs to a predefined property set such as the Material property set, you can use static fields in its corresponding **Property* class as property keys. For example, instead of specifying the following for the Structure property in the Material property set:

```
row.PropertySets["Material"]["Structure"].Value
```

specify:

```
from Symyx.Framework.Material import MaterialProperty
row.PropertySets[MaterialProperty.Structure].Value
```

See Material-related property set definitions for a list of predefined property sets related to the Material property set.

Your scripts can access properties and property sets as objects. The Framework also provides script variables that represent the objects related to properties and property sets.

Access a Table and its Rows

The `Symyx.Framework.Properties.IPropertySetHost` object represents a row in a table in a Workbook experiment containing table sections. Event handler scripts that are run on a table section can manipulate rows by using the row script variable to access `IPropertySetHost` and its API. Scripts can access a `Symyx.Notebook.Sections.Table.TableSection` object and its API by using the table script variable.

Table Section Script Variables

The following are the script variables available to a Table Section script:

Variable	Description
section	Specifies the <code>Symyx.Notebook.DocumentSection</code> or a section.
Table	Specifies the <code>Symyx.Notebook.Sections.TableSection</code> .
row	Specifies the <code>Symyx.Framework.Properties.IPropertySetHost</code> representing a row in the table.
owner	Added by generic scripting and dynamic tool bars and is the object that has the property on which the script is defined. The other scope variables are not present for the owner, but might exists for the subject of the event.

Import and Export Data

The Symyx Framework provides the capability to transfer data to and from a Table Section by using:

- View-based data transfer mechanism - a user can copy and paste data between separate grids such as copying from and pasting to a `TableSection`.
- Property-based data sharing - the property set is mediated with the object when it is imported or exported. When importing or exporting a row, the object properties can be directly manipulated by using `Symyx.Framework.Properties.IPropertySetHost`. The `Symyx.Framework.Properties.ImportExport.ImportExportData` class is a data container that can be used for data transfer, and inherits from `IPropertySetHost`.
- Shared object reference - a direct reference to the underlying data object is used if multiple views are sharing the same model.

The `Symyx.Notebook.Sections.Table.TableSection` class implements the `System.Framework.TabularData.ITabularData` interface that:

- Contains the Rows property. Each row in the Rows property is represented by an `IPropertySetHost`.
- Has methods for handling row data, such as methods for counting, adding, getting index of, getting rows.
- Implements the `System.Framework.TabularData.IImportExportTabularData`.

The `IImportExportTabularData` interface contains the following methods for importing and exporting data (using `ImportExportData`), and for getting the schema of the data:

```
Schema GetSchema();
Schema GetSchema(bool includeReadOnlyProperties);
ImportExportDataList ExportAll();
ImportExportData Export(string id);
ImportExportData AddAndExport();
```

```
void Import(IEnumerable<ImportExportData> dataToImport);
void Import(ImportExportData dataToImport);
void Update(IEnumerable<ImportExportData> updateData);
void Update(ImportExportData updateData);
```

A Schema, `Symyx.Framework.Properties.ImportExport.Schema`, is a property set definition (PSD) containing all property set definitions of the data to import or export. The schema contains the definition of the data (metadata) rather than data. For import and export purposes, each property in the schema is represented by the `Symyx.Framework.Properties.ImportExport.ImportExportPropertyClass`.

The `ImportExportData` object,

`Symyx.Framework.Properties.ImportExport.ImportExportData`, is a temporary `IPropertySetHost` data storage object that conforms to the schema. `ImportExportData` is the object that contains the data for importing or exporting.

The properties in a Schema include all the properties from all the property set definitions used by the table. Optionally, you can exclude or include the read-only properties. The Schema class provides methods for marking properties in the schema for inclusion or exclusion during the import process. The following example imports only one property from a schema. The example uses the `Schema.ExcludeAllForImport` method to exclude all properties, and the `ImportExportPropertyClass.Include` property to specify only one property to be imported.

```
# Get the schema of the table, and include only the Demo.Name property.
schema = table.GetSchema()
schema.ExcludeAllForImport() schema["Demo.Name"].Include = True
# Create an ImportExportData for the schema.
importData = schema.CreateImportData()
# Import the data. table.Import(importData)
```

Schemas for import do not have the read-only properties. Schemas for export can have the read-only properties, but those cannot update the same row.

Import Summary Data

To create an object containing a table schema and data, use

`Symyx.Framework.Properties.ImportExport.ImportExportData`.

To import the `ImportExportData` object to a `TableSection`, use `section.Import(ImportExportData)`.

To access a property in an `ImportExportData` object, use the "

`[propertyName.propertyName]` format, for example, `Material.Name`.

The following example counts the number of times a name appeared in the rows of a Materials Section table, and uses `ImportExportData` to import information into another Materials Section table called `Table2`.

```
# Create an enumeration
table = owner
a = {}
# while iterating through the rows of a table,
# count the occurrence of each color for row in table.Rows:
name = row.PropertySets["Material"]["Name"].Value if a.has_key(name):
a[name] = a[name] + 1 else:
a[name] = 1
# To make this script work, insert a Materials Section, and rename it to
"Table2".
```

```

# Table2 will be the summary table. for section in table.Document.Sections:
if section.Title == "Table2": summary = section
break
else:
raise RuntimeError, "Table2 not found"
# Clear the Summary table summary.Clear()
# Get the schema of the Summary table schema = summary.GetSchema()
# For each name in the enumeration, create an ImportExportData object,
# update its Name and Comments properties,
# and import that row into the Summary table. for key in a.keys():
importData = schema.CreateImportData() materialName = str(key) importData
["Material.Name"].Value = materialName
importData["Material.Comments"].Value = "# rows with " + materialName + ": "
+ str(a[key])
summary.Import(importData)
# Display a completion message after the import. from System.Windows.Forms
import MessageBox
MessageBox.Show("Summary Import Complete. See summary rows in Table2
section.")

```

To run this script, add it to a custom toolbar item on a Materials Section. To create the summary table, insert another Materials Section and rename it to Table2. Add some rows with some repeating names to the first Materials Section to summarize in Table2.

ImportExportData to Update Data

To update rows in a table, export the rows and use the `table.Update` method. The following example uses `table.ExportAll()` to export all rows of a table. It gets the `IPROPERTYSET.Host.Id` of the first row, creates an `ImportExportData` object, sets its `Id` and `Demo.Name` properties, and uses `table.Update(importData)` to update the row in the same table with the specified `Id`.

To access a property in an `ImportExportData` object, use the "[propertyName]" format.

```

# Export all rows of the table. table = owner
rows = table.ExportAll()
# Get the Id of the first row, and display it.
id = rows[0].Id
import System System.Windows.Forms.MessageBox.Show(id)
# Get the schema of the table, and include only the
# Material.Name property.
schema = table.GetSchema()
schema.ExcludeAllForImport()
schema["Material.Name"].Include = True
# Create an ImportExportData for the schema.
importData = schema.CreateImportData()
# Set the Id and the Demo.Name property with a new value
importData.Id = id
importData.Data["Material.Name"].Value = "Hello world"
# Update that row in the same table.
table.Update(importData)

```

To run this script, add it to a custom toolbar item on a Materials Section.

Lock Imported Rows

To lock an imported row, set the `ImportExportData.IsLocked` property to `true`.

The following script creates an `ImportExportData` from the table, creates multiple rows for importing, locks those rows, and imports them to the table.

```
# Get the table
Table = owner
# Get the schema
schema = Table.GetSchema()
# Create the ImportExportData list
importList = schema.CreateImportData(3)
# Set the values for multiple rows for i in (0,1,2):
importList[i]["Material.Name"].Value = "test material" + str(i)
importList[i]["Material.Comments"].Value = "test material comment" + str(i)
# Lock the row
importList[i].IsLocked = True
# Import the rows table.Import(importList)
```

To run this script, add it to a custom toolbar item on a Materials Section. After this script executes, the added rows are locked.

Prevent the Removal of Locked Rows

To prevent users from removing a row, check the row `.IsLocked` property and set `e.Cancel` to `true`.

```
if row.IsLocked:
    e.Cancel = True
    e.CancelMessage = "Cannot delete locked row."
```

To run this script, add it to the Removing Row Event Scripting on a Materials Section. Lock a row, and attempt to delete it. You should see the message from the script.

Export or Import All Table Rows

To export all rows, use `table.ExportAll`.

To import rows into a table, use `table.Import`. The following IronPython example uses `table.ExportAll()` to export all rows from a table, and uses `table.Import(rows)` to import them to a target table section. The following IronPython example exports all rows from the current table and imports them to another table.

```
# Find the target "Table2" section
for section in table.Document.Sections: if section.Title == "Table2":
    target = section break
# This "else:" belongs with the "for" loop, so keep it aligned with "for".
# The "else:" executes if "break" is never called. else:
    raise RuntimeError, "Table2 section not found"
# Export all rows from the current table rows = table.ExportAll()
# Import the rows to the target table target.Import(rows)
# Display the number of imported rows
from System.Windows.Forms import MessageBox
MessageBox.Show("Imported " +
    rows.Length.ToString() + " rows into Table 2")
```

If you use this sample script for a custom toolbar item, `owner` is the only script variable that is available. The variable `owner` represents the Table Section on which the dynamic toolbar is defined. To use this example as a custom toolbar script, insert `table = owner` before `table` is first used.

Request Column Dictionary Event

The Table Section `RequestColumnDictionary` event occurs when a column that is defined with a dictionary, `ComboBox`, is creating the list of values. For the `RequestColumnDictionary` event, you can specify an IronPython script that dynamically builds a dictionary based on values in a column or table. The `RequestColumnDictionary` event script, uses the `e` script variable that uses the `Symyx.Notebook.Sections.Table.DictionaryEventArgs` arguments:

- `AllowFreeText` to indicate if free text entry is allowed.
- Property to represent the `Symyx.Framework.Properties.Property` containing the `ComboBox`.
- `Items` to represent the `System.Collections.CollectionBase` containing the items in the list.

Table Section Script Events

The Table Section provides scripting capabilities for the events described in the following table.

Event	Description
AddingRow	Occurs when a row is being added to the table. The EventArgs type is <code>Symyx.Framework.Collections.PendingItemEventArgs<IPropertySetHost></code> , whose properties are: <ul style="list-style-type: none"> ■ <code>e.Cancel</code> - Indicates whether or not to cancel the pending action on the item. If True, the <code>CancelMessage</code> displays and the user must press the Escape key to cancel the action. ■ <code>e.CancelMessage</code> - A message describing why the pending action on the item should be cancelled
RowAdded	Occurs after a row is added to the table. The EventArgs type is <code>Symyx.Framework.Collections.ItemEventArgs<IPropertySetHost></code> , whose property is: <ul style="list-style-type: none"> ■ <code>e.Item</code> - the row added
RemovingRow	Occurs when a row is being deleted from the table. The EventArgs type is <code>Symyx.Framework.Collections.PendingItemEventArgs<IPropertySetHost></code> , whose properties are: <ul style="list-style-type: none"> ■ <code>e.Cancel</code> - Indicates whether or not to cancel the pending action on the item. If True, the <code>CancelMessage</code> displays and the user must press the Escape key to cancel the action ■ <code>e.CancelMessage</code> - A message describing why the pending action on the item should be canceled.
RowRemoved	Occurs after a row is deleted from the table. The EventArgs type is <code>Symyx.Framework.Collections.ItemEventArgs<IPropertySetHost></code> , whose property is: <ul style="list-style-type: none"> ■ <code>e.Item</code> - the row removed

Event	Description
RowChanged	<p>Occurs after a row changed.</p> <p>The EventArgs type is <code>Symyx.Framework.Properties.ItemChangedEventArgs<IPropertySet Host></code>, whose properties are:</p> <ul style="list-style-type: none"> ■ <code>e.Key</code> - the key of other object whose value was changed ■ <code>e.NewValue</code> - the new value of the changed property ■ <code>e.NewValueIsNull</code> - indicates whether or not the new value is null ■ <code>e.OldValue</code> - the old value of the changed property ■ <code>e.OldValueIsNull</code> - indicates whether or not the old value is null
RequestColumnDictionary	<p>Occurs for each dictionary-type property in the row.</p> <p>The EventArgs type is <code>Symyx.Framework.Properties.DictionaryEventArgs</code>, whose properties are:</p> <ul style="list-style-type: none"> ■ <code>e.AllowFreeText</code> - indicates whether or not free text entry is allowed ■ <code>e.Items</code> - the list of items contained in the dictionary ■ <code>e.Property</code> - the property

Event	Description
SigningOptions	<p>Occurs if the property changed and the property contains Signing Options. The EventArgs type is <code>Symyx.Framework.Properties.ValueChangingEventArgs</code>, whose properties are:</p> <ul style="list-style-type: none"> ■ <code>e.Cancel</code> - Indicates whether or not to cancel the pending action on the item. If True, the <code>CancelMessage</code> displays and the user must press the Escape key to cancel the action ■ <code>e.CancelMessage</code> - A message describing why the pending action on the item should be cancelled ■ <code>e.NewValue</code> - The new value of the changed property ■ <code>e.NewValueIsNull</code> - Indicates whether or not the new value is null ■ <code>e.OldValue</code> - the old value of the changed property ■ <code>e.OldValueIsNull</code> - Indicates whether or not the old value is null ■ <code>e.AddValidationResult(ValidationResults)</code> - Adds a list of <code>Symyx.Framework.Review.ValidationResult</code> objects containing a message and severity level ■ <code>e.AddValidationResult(string, SeverityLevel)</code> - Adds a message and <code>Symyx.Framework.Review.SeverityLevel</code> ■ <code>e.AddValidationResult(string)</code> - Adds a message to be displayed as an error <p>Use this event to add validation scripts for properties in a Table Section. To handle validation errors, either:</p> <ul style="list-style-type: none"> ■ Use <code>e.Cancel = True</code> (along with a <code>e.CancelMessage</code>) to prevent the change altogether or ■ Use <code>e.AddValidationResult</code> to add a validation message (visible in Review) at either the Error, Warning, or Info level.

Table Section Event Variables

Event: Row Removed

Script variable	Description
<code>e</code>	<code>Symyx.Framework.Collections.ItemEventArgs<IPropertySetHost></code>
<code>row</code>	Represents the object to remove
<code>sender</code>	<code>Symyx.Notebook.Sections.Table.TableSection</code>

Event: After Record Weights

Script variable	Description
<code>e</code>	<code>System.EventArgs</code>
<code>sender</code>	<code>Symyx.Notebook.Sections.Table.TableSection</code>

Event: Before Record Weights

Script variable	Description
e	<code>Symyx.Notebook.Sections.Table.RecordWeightCancelEventArgs</code>
sender	<code>Symyx.Notebook.Sections.Table.TableSection</code>

Event: Request Column Dictionary

Script variable	Description
e	<code>Symyx.Framework.Properties.DictionaryEventArgs</code>
row	Represents the object containing the dictionary property
sender	<code>Symyx.Notebook.Sections.Table.TableSection</code>

Event: Row Added

Script variable	Description
e	<code>Symyx.Framework.Collections.ItemEventArgs<IPropertySetHost></code>
row	Represents the object that was added
sender	<code>Symyx.Notebook.Sections.Table.TableSection</code>

Event: Row Changed

Script variable	Description
e	<code>Symyx.Framework.Properties.ValueChangedEventArgs</code>
row	Represents the object with changed property
sender	<code>Symyx.Notebook.Sections.Table.TableSection</code>

Table Section Script Examples**To run the example scripts:**

1. Login to Workbook as a user with the `SectionTemplate.Editor` permission.
2. Create a new experiment template in Experiment Editor.
3. Add a Material Section.
4. On the Properties pane of the section, configure (click) the Dynamic Toolbars property.
5. Add a toolbar, and create a toolbar item, for example, Insert Excel file.
6. Add the script to the toolbar item.
7. Save the template.
8. Add data to the Material Section.
9. Click the toolbar item to run the script.

Get Table Schema

The following example gets the schema of a table and lists its properties:


```

table = owner
schema = table.GetSchema() s = ""
    for property_class in schema:
        s = s + property_class.Key + " - " + property_class.TypeName
    MessageBox.Show(s)

```

To run the script, add it to a custom toolbar item on a Materials Section.

Access Rows, Property Sets, and Properties

To access the rows of a table, use `table.Rows`. To access the property sets used in a row, use `row.PropertySets`. The following example uses `table.Rows` to iterate through the rows of a table. For each row, it uses `table.GetIndexof(row)` to display the row index, iterates through the property sets (`row.PropertySets`) and displays all the property names (`property.Key`) and their values (`property.DisplayValue`).

```

table = owner s = ""
# Iterate through all rows in the table for row in table.Rows:
    s = s + "Row #" + str(table.GetIndexof(row)) + " -- "
# Iterate through all property sets in each row for propertySet in
row.PropertySets:
    if propertySet.Key != "Core":
        s = s + "Property Set: " + propertySet.Key + "\r\n"
# Iterate through all properties in each property set for property in
propertySet:
    s = s + "" + property.Key + ":" + \ property.DisplayValue + "\r\n"
    s = s + "\r\n"
# Display the string from System.Windows.Forms import MessageBox
MessageBox.Show(s)

```

To run this script, add it to a custom toolbar item on a Materials Section.

Invoke a Form and Add Rows

To add a row, use `table.AddRow`. The following example invokes `Symyx.Notebook.Sections.Table.AddNewRowsForm` to prompt the user for the number of rows to add, and then uses `table.AddRow()` to add the rows.

```

from Symyx.Notebook.Sections.Table import AddNewRowsForm
from System.Windows.Forms import DialogResult
    table = owner try:
        addNew = AddNewRowsForm() result = addNew.ShowDialog() if result ==
        DialogResult.OK:
            i = 0
            while i < addNew.RowCount: table.AddRow()
                i = i + 1
            finally:
                addNew.Dispose()

```

To run the script, add it to a custom toolbar item on a Materials Section.

Update a Table Property

To update a property of a table, use `SetValue` on the table property. The following example uses `table.ExtendedProperties.GetValue(propName, defaultvalue)` to increment the value of the `MaxRowNumber` property of a table, and uses `table.ExtendedProperties.SetValue(propName, defaultvalue)` to update the value of `MaxRowNumber`.

This example also shows how to update the value of a field in a new row in a table. It sets the Value of the Comments field of a new row in the Materials Section table.

```
table = owner
maxRow = table.ExtendedProperties.GetValue("MaxRowNumber", 0) + 1
row = table.AddRow()
row.PropertySets["Material"]["Comments"].Value = "test" + str(maxRow)
table.ExtendedProperties.SetValue("MaxRowNumber", maxRow)
```

To run the script, add it to a custom toolbar item on a Materials Section.

Export and Importing all Rows

To export all rows, use `table.ExportAll`. To import rows into a table, use `table.Import`. The following example uses `table.ExportAll()` to export all rows from a table, and uses `table.Import(rows)` to import them to a target table section.

```
# Find the target "Table2" section
table = owner
for section in table.Document.Sections:
    if section.Title == "Table2":
        target = section
        break
# This "else:" belongs with the "for" loop, so keep it aligned with "for".
# The "else:" executes if "break" is never called.
else:
    raise RuntimeError, "Table2 section not found"
# Export all rows from the current table
rows = table.ExportAll()
# Import the rows to the target table
target.Import(rows)
# Display the number of imported rows
from System.Windows.Forms import MessageBox
MessageBox.Show("Imported " + rows.Count.ToString() + " rows into Table 2")
```

To run the script, add it to a custom toolbar item on a Materials Section.

To create the target table, insert another Materials Section and rename it to `Table2`. Add some rows to the first Materials Section to export in to `Table2`.

Material Section Script Variables

The Material Section uses the property set definition defined in *the material property set definition*. The events and script variables that are defined in the Table Section are also available to the Material Section.

The following table shows the script variables are available to the Material Section.

Property	Description
Name	Specifies a string containing the name of the material.
Density	Specifies a <code>Symyx.Framework.Quantity</code> object containing the density of the material.
Comments	Specifies a string containing the comments about the material.
Structure	Specifies a <code>Symyx.Framework.Chemistry.Structure</code> object representing the structure contained in the material.
MF	Specifies a string containing the molecular formula of the material.
MW	Specifies a <code>Symyx.Framework.Value</code> object containing the molecular weight of the

Property	Description
	material.
Role	Specifies a string containing the role of the material. By default, this property is hidden from the Material Section table. To display it, select this property from the Column Chooser dialog in the Material Section.
InitialAmount	Specifies a <code>Symyx.Framework.Quantity</code> object containing the initial amount of the material. By default, this property is hidden from the Material Section table. To display it, select this property from the Column Chooser dialog in the Material Section.
Location	Specifies a <code>Symyx.Framework.Materials.Location</code> object containing the location of the material. This property is always hidden from the Material Section table.
Parent	Specifies a <code>Symyx.Framework.Materials.Material</code> object containing the parent of the material. This property is always hidden from the Material Section table.
Preparation	Specifies a <code>Symyx.Framework.Materials.Preparation</code> object containing the instructions for preparing the material. This property is always hidden from the Material Section table.

The `Symyx.Framework.Materials` namespace contains classes and interfaces that support the objects used with the Material property set.

Material Property Set Definitions

Workbook provides several material-related property sets that are usable in table section such as in the Material, Synthetic Chemistry, Analytic Chemistry, Reaction Materials, and Equipment sections. In a Material section that already contains the Material property set, a template editor can insert the Actual Amount, Container, and Equipment property sets, providing the ability to add related data about the material.

The `Symyx.Framework.Materials` namespace provides the property set definitions. The keys or names for the property set definitions are defined in the `Symyx.Framework.Materials.MaterialPropertySets`. For each property set definition, `Symyx.Framework.Materials` also provides a corresponding Property class, identified by its property class key, that contains its properties, identified by their property keys. For example, for the ActualAmount property set definition, the `ActualAmountProperty` class contains the Amount, CalcMoles, Yield, CalcMass, and CalcVol properties.

Property set Keys	Property class keys	Property keys
ActualAmount	ActualAmountProperty	Amount CalcMoles Yield CalcMass CalcVol
Container	ContainerProperty	Type Amount Label

Property set Keys	Property class keys	Property keys
		Barcode Capacity TareWeight Comments Location Material
Equipment	EquipmentProperty	EquipmentName EquipmentId Category Classification Department EquipmentType IntendedUse LastCalibrationDate LastServiceDate Location Manufacturer Model NextCalibrationDate NextServiceDate ReferenceNumber Role SerialNumber Status
Material	MaterialProperty	For more information, see Material Class Properties .
MaterialRegistration	MaterialRegistrationProperty	Registered SubstanceId BatchId NormalizedStructure Response
Operation	OperationProperty	Name Number Notes Icon Observations Category ContentFields IsActive SelectedPropertySetDefinitions UsedPropertySets ExecutionState TimeStamp

Property set Keys	Property class keys	Property keys
		Phrase Alerts
PlannedAmount	PlannedAmountProperty	CalcEquiv Amount CalcMass CalcVolume CalcMoles
ReactionMaterial	ReactionMaterialProperty	Label LimitingReagent NEMAKey PurityConcentration StoichiometricCoefficient Step StepId
ReactionStep	ReactionStepProperty	Description Name PathDescription Name Path Step Reaction
SampleIdentification	SampleIdentificationProperty	SampleId
UnitProcedure	UnitProcedureProperty	Name Number Description Icon Notes Observations Category EnforceOrder Location TimeStamp Repeatable ExecutionState Predecessors Alerts

Use the property classes and keys to reference a specific property in a particular property set used in a material-related section. For descriptions of the property keys listed above, see the corresponding Property class documentation.

Because a material-related section uses the Table Section, a row in a material-related section corresponds to a `Symyx.Framework.Properties.IPropertySetHost` object. The

`IPropertySetHost.PropertySets` contains the property set definitions used in the section. To reference a property in a particular row, use the following syntax:

```
row.PropertySets[MaterialPropertySets.propertySetKey]
[propertyClassKey.propertyName]
```

The row is an object that represents a row, `IPropertySetHost`, in the section.

The following example gets the Name property of a material on a row in a Material Section:

```
row = aSection.GetRow()
aName = row.PropertySets
[MaterialPropertySets.Material][MaterialProperty.Name].Value
```

The following example sets the MW property values of rows in a Material Section:

```
rows = scSection.GetRows()
rows[0].PropertySets[MaterialPropertySets.Material]
[MaterialProperty.MW].Value = Value (0, 0)
rows[1].PropertySets[MaterialPropertySets.Material]
[MaterialProperty.MW].Value = Value (115.131, 6)
rows[2].PropertySets[MaterialPropertySets.Material]
[MaterialProperty.MW].Value = Value (0, 0)
```

The following example shows another way of referencing a property in the property set. The `limitingReagentRM` variable is initially assigned the `ReactionMaterial` property set. Subsequently, one of its properties, `ReactionMaterialProperty.LimitingReagent` is referenced using the `limitingReagentRM` variable.

```
# rows[] is defined in preceding example
limitingReagent = rows[1] # Cyclohexane 6 significant figures of MW value
limitingReagentRM = \ limitingReagent.PropertySets
[MaterialPropertySet.ReactionMaterial]
limitingReagentRM[ReactionMaterialProperty.LimitingReagent].Value = True
```

Nullable for Primitive Types

If you are using the API to create an application that programmatically defines new property set definitions, for best performance you should not create any `PropertyClass` using nullable types. In the following example, use the standard, non-nullable integer (`int`) and not the nullable integer (`int?`).

```
var notRecommendedNullablePrimitive = new PropertyClass("Content",
AnalyticalPreparationProperty.DilutionFactor, typeof (int?));
```

Use:

```
var recommendedPrimitiveNotNullable = new PropertyClass("Content",
AnalyticalPreparationProperty.DilutionFactor, typeof (int));
```

Unless the `AllowNulls` property of the `PropertyClass` is set to false, the Property allows null values even if defined with a non-nullable type.

To set the value to null, use `propertySet.SetValue(propertyKey, null)`.

Material Section Script Examples

The following example is a material validation script for the `Role` property. In this example, `e` is the `EventArgs` script variable, and `material` is the script variable representing the row in the Material Section.

```
if e.NewValue and \
(material.Role is None or material.Role not in ('Sample', 'Standard')):
```

```
e.Cancel = True
e.CancelMessage = "The role must be 'Sample' or 'Standard'."
```

To run this script, add it to the Signing Options event on a Materials section. To add the validation script to the Signing Options event:

1. Click the ellipses button on the **Signing Options** event scripting property.
2. In the PropertySigningOptionsList window, click **Add**.
3. In the Applies To field, select a property, for example, **Material > Role**.
4. Add the script to the **Validation > Custom Script** property.

Access the Material Structure

The `Material.Structure` property contains a `Symyx.Framework.Chemistry.Structure` object. The Structure object provides both the Molfile and Chime string formats. The following example displays the Molfile string of the current structure:

```
from System.Windows.Forms import MessageBox
molfileString = material.Structure.Molfile
MessageBox.Show(molfileString)
```

Create a Review Message

The following script checks the expiry date in an `AnalyticalMaterial` property set and creates a review message if the validation fails. An error is returned, if the `ExpiryDate` is expired, and a warning is returned if the `ExpiryDate` is the current date.

```
from System import DateTime
from Symyx.Framework.Review
import SeverityLevel

try:
    expiry = e.NewValue.PropertySets['AnalyticalMaterial'].GetValue
    ('ExpiryDate').Date
    today = DateTime.Now.Date
    # expired already
    if today.CompareTo(expiry) > 0 :
        e.AddValidationResult('This sample expired on %s' % expiry.ToString
        ('D'), SeverityLevel.Error)
    # expired today
    elif today.CompareTo(expiry) == 0 :
        e.AddValidationResult('This sample expires today',
        SeverityLevel.Warning)
except:
    # if this did not work, it is likely that the material
    # does not have an ExpiryDate property, so continue pass
```

To run this script:

1. Login to Workbook as a user with the `SectionTemplate.Editor` permission.
2. Create a new experiment template in Experiment Editor. Add a Table Section or any of the materials-related section.
3. On the Properties pane of the section, in the Property Sets property, select `AnalyticalMaterial`.

4. On the Signing Options event scripting property, add the script to the **AnalyticalMaterial > ExpiryDate** property.
5. Enter a past date in the **ExpiryDate** on the table, and choose **Tools > Run Review**.
A message displays in the Review Results pane indicating that the date expired.

Scripting Material Import

The Material section, the Synthetic Chemistry section, and the Analytical Materials section have a material import feature. Users can invoke the Import Material dialog and search by name, CAS number, or structure using the DiscoveryGate database.

You can modify the default behavior of the material import functionality in any section template using an IronPython script that runs the `BeforeImportMaterials` event or the `AfterImportMaterials` event.

- When the end-user clicks on the Import Form toolbar button, Workbook fires the `BeforeImportMaterial` event, and displays the Material Import dialog.

Note: If `e.HideDefaultDialog` is set to `True`, the process stops. In the initial execution of the before import material script, only the `e.HideDefaultDialog` is used by the code.

- The end-user selects a data source and enters search criteria such as the structure name, CAS number, or a structure in the dialog. One or more results display in the dialog's table.

Note: The `e.ImportedList` contains a list of the materials selected by the user in the Material Import dialog. The script can modify the list, for example, you can write the script to modify name to use a standard capitalization.

- The end-user selects one or more results for import and clicks the Import button, which fires the `BeforeImportMaterials` event.

In the script's execution, the template editor has access to both the original list from the import dialog and the imported materials in the table section, available as `e.ImportedMaterials`, for example, the template editor could concatenate the chemical name with the diluents name and the concentration.

- Workbook imports the selected materials into the section, and then fires the `AfterImportMaterial` event. The Material Import dialog closes.
- If you enter a past date in the `ExpiryDate` on the table, and choose **Tools > Run Review**. You should see a message in the Review Results pane indicating that the date expired.

For more information, see *Import Materials* in the BIOVIA Workbook online help.

You can learn how to set up material import from the Database Web Service through the MaterialInfoManager LookupService. For more information, see the *Vault Administrative Tools Guide Application Permissions* chapter.

Refer to the [OnReview Script Example](#) for a similar example.

BeforeImportMaterials Event Script Example

You could use a script to modify values such as changing to lowercase, truncating insignificant decimal places, or converting currency.

The following IronPython script for the `BeforeImportMaterials` events blocks the default dialog and posts a custom message box.

```
from Symyx.Framework.Properties import IPropertySetHost
from Symyx.Framework.Materials import Material
```



```
from System import String
from System.Windows.Forms import MessageBox

MessageBox.Show('my custom dialog')
e.HideDefaultDialog = True
```

AfterImportMaterials Event Script Example

The following IronPython script for the synthetic chemistry section modifies the imported material objects with the following line:

```
material.Name = material.Name.ToLower() + ' in ' +
diluentProperty.Value.ToString()
```

The statement changes the following:

- Salicylic Acid to salicylic acid
- The name of the chemical by concatenating the diluents information. The salicylic acid that has Diluent='WATER' becomes Salicylic Acid in water.

`e.ImportedList` contains a list of the materials selected by the user in the Material Import dialog. The script can modify this list in place. For example, you can modify names to have a standard capitalization.

In this script execution, the template editor has access to both the original list from the import dialog, and the imported materials within the table section, available as `e.ImportedMaterials`. For example, you could concatenate the chemical name with the diluent's name and the concentration.

Testing Examples

To test the examples:

1. Click the Synthetic Chemistry section.
2. Click the **Import From** list and choose the newly created option, **Import From Test**.

A new row is created and displays the incremented row count in the leftmost column.

The most recently chosen option of the Import From dropdown list becomes the default.

Script Variables for Experiment and Common Section Events

Scripts that handle experiment editor events can be defined at the experiment or section level. The experiment and each Workbook section has an Event Scripting property that can execute a script for the following events:

- OnApplicationClosing
- OnApplicationLoaded
- OnInsertingSection
- OnLockingSection
- OnMenuItemEnabledStatesUpdated
- OnRemovingSection
- OnSaving
- OnSaved
- OnSectionActivated
- OnSectionDeactivated
- OnSectionInserted

- OnSectionLocked
- OnSectionRemoved
- OnSectionUnlocked
- OnToolBarButtonEnabledStatesUpdated
- OnUnlockingSection

The following are the script variables that can be used for the events listed above. These script variables are also available to custom toolbar scripts:

Script variable	Represents
active_workspace	<code>Symyx.Notebook.Vault.NotebookWorkspace</code>
editor	<code>Symyx.Notebook.ApplicationManagement.IDocumentEditor</code> Note: If you have the document or section, the Application property return the IDocumentEditor editor.
sender	<code>Symyx.Notebook.ApplicationManagement.IDocumentEditor</code>
e	One of the following, depending on the corresponding event: <code>Symyx.Framework.ApplicationManagement.ApplicationLoadedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.InsertingSectionEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.LockingSectionEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.SectionLockedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.UnlockingSectionEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.SectionUnlockedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.RemovingSectionEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.SectionRemovedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.ToolBarButtonEnabledStatesUpdatedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.SectionActivatedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.SectionDeactivatedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.SavingEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.SavedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.MenuItemEnabledStatesUpdatedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.ApplicationClosingEventArgs</code>
owner	<code>Symyx.Notebook.Document</code> if script is defined at the experiment level, or the section object if script is defined at the section level.

Script Variables for Events

You can define scripts that handle experiment editor events at the experiment or section level. The experiment and each experiment section has an Event Scripting property that can execute a script for the following events:

- OnApplicationClosing
- OnApplicationLoaded

- OnInsertingSection
- OnLockingSection
- OnMenuItemEnabledStatesUpdated
- OnRemovingSection
- OnSaving
- OnSaved
- OnSectionActivated
- OnSectionDeactivated
- OnSectionInserted
- OnSectionLocked
- OnSectionRemoved
- OnToolBarButtonEnabledStatesUpdated
- OnUnlockingSection

You can use the following script variables with the events listed above, and with custom toolbar scripts.

Script Variable	Represents
active_workspace	<code>Symyx.Notebook.Vault.NotebookWorkspace</code>
editor	<code>Symyx.Notebook.ApplicationManagement.IDocumentEditor</code> Note: If you have the document or section, the Application property returns the editor, <code>IDocumentEditor</code> .
sender	<code>Symyx.Notebook.ApplicationManagement.IDocumentEditor</code>
e	One of the following: <code>Symyx.Framework.ApplicationManagement.ApplicationLoadedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.InsertingSectionEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.LockingSectionEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.SectionLockedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.UnlockingSectionEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.SectionUnlockedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.RemovingSectionEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.SectionRemovedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.ToolBarButtonEnabledStatesUpdatedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.SectionActivatedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.SectionDeactivatedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.SavingEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.SavedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.MenuItemEnabledStatesUpdatedEventArgs</code> <code>Symyx.Notebook.ApplicationManagement.ApplicationClosingEventArgs</code>
owner	<code>Symyx.Notebook.Document</code> if script is defined at the experiment level, or the section object if script is defined at the section level.

Workbook Sections

The following table lists the Workbook sections and their object types.

Do not use the following sections in scripts:

- Grouped Materials
- Plate Layout
- Reaction List
- Spreadsheet

Section	Object Type
Analytical Materials	<code>Symyx.Notebook.Sections.Materials.IMaterialsSection</code>
Equipment	<code>Symyx.Notebook.Sections.Equipment.EquipmentSection</code>
File Section	<code>Symyx.Notebook.Sections.ExternalFile.ExternalFileSection</code>
Form Section	<code>Symyx.Notebook.Sections.Forms.FormsSection</code>
Formulation Section	<code>Symyx.Notebook.Sections.Formulation</code>
Grouped Materials	For internal use only. Do not use this in your script
Materials	<code>Symyx.Notebook.Sections.Materials.IMaterialsSection</code>
Plate Layout	For internal use only. Do not use this in your script.
Preparation s	<code>Symyx.Notebook.Sections.SamplePreparation.SamplePreparationSection</code>
Text Section	<code>Symyx.Notebook.Sections.Text.TextSection</code>
Reaction List	For internal use only. Do not use this in your script.
Reaction Scheme	<code>Symyx.Notebook.Sections.ReactionScheme.ReactionSchemeSection</code>
Reference	<code>Symyx.Notebook.Sections.Reference.ReferenceSection</code>
Sample Analysis Table	<code>Symyx.Notebook.Sections.Table.TableSection</code>
Spreadsheet	For internal use only. Do not use this in your script.
Synthetic Chemistry	<code>Symyx.Notebook.Sections.Materials.IMaterialsSection</code>

Section	Object Type
Table	<code>Symyx.Notebook.Sections.Table.TableSection</code>

Except for `Symyx.Notebook.Sections.Materials.IMaterialsSection`, the namespaces containing the section objects are found in `namespace-name.dll` assemblies located in the `lib` directory of the ELN SDK installation. The `Symyx.Notebook.Sections.Materials.IMaterialsSection` is in the `Symyx.Notebook.dll` assembly.

Materials Section Event Script Variables

`MaterialsSection` inherits from `TableSection`, the following variables are available to the `MaterialsSection`:

Event: Before Import Materials

Script variable	Represents
<code>e</code>	<code>Symyx.Notebook.Sections.Materials.ImportMaterialsEventArgs</code>
<code>sender</code>	<code>Symyx.Notebook.Sections.Materials.MaterialSection</code>

Event: After Import Materials

Script variable	Represents
<code>e</code>	<code>Symyx.Notebook.Sections.Materials.ImportMaterialsEventArgs</code>
<code>sender</code>	<code>Symyx.Notebook.Sections.Materials.MaterialSection</code>

Event: Signing Options

Script variable	Represents
<code>properties</code>	<code>Symyx.Framework.Properties.Property.PropertySet</code>
<code>host_object</code>	<code>Symyx.Framework.Materials.Material</code>
<code>vault_object</code>	<code>Symyx.Framework.Materials.Material</code>
<code>document_template</code>	<code>Symyx.Notebook.Document</code>
<code>document_section</code>	<code>Symyx.Notebook.Sections.Materials.MaterialsSection</code>
<code>table</code>	<code>Symyx.Notebook.Sections.Materials.MaterialsSection</code>
<code>e</code>	<code>Symyx.Framework.Properties.ValueChangingEventArgs</code>

Sample Preparation Section Script Variables

Event: Row Added

Script variable	Description
e	<code>Symyx.Framework.Collections.ItemEventArgs<IPropertySetHost></code>
row	The row object that was added
sender	<code>Symyx.Notebook.Sections.SamplePreparation.SamplePreparationSection</code>
preparationSectionConcentrationUnit	<code>Symyx.Framework.UnitKey</code> for the calculated concentration
preparationSectionMassUnit	<code>Symyx.Framework.UnitKey</code> for the calculated mass
preparationSectionVolumeUnit	<code>Symyx.Framework.UnitKey</code> for the calculated volume
preparationSectionMaterialRoles	<code>Symyx.Framework.UnitKey</code> for the material roles for the preparation

Event: Removing Row

Script variable	Description
e	<code>Symyx.Framework.Collections.PendingItemEventArgs<IPropertySetHost></code>
row	The row object to be removed
sender	<code>Symyx.Notebook.Sections.SamplePreparation.SamplePreparationSection</code>
preparationSectionConcentrationUnit	<code>Symyx.Framework.UnitKey</code> for the calculated concentration
preparationSectionMassUnit	<code>Symyx.Framework.UnitKey</code> for the calculated mass
preparationSectionVolumeUnit	<code>Symyx.Framework.UnitKey</code> for the calculated volume
preparationSectionMaterialRoles	<code>Symyx.Framework.UnitKey</code> for the material roles for the preparation

Event: Row Removed

Script variable	Description
e	<code>Symyx.Framework.Collections.PendingItemEventArgs<IPropertySetHost></code>
row	The row object that was removed

Script variable	Description
sender	<code>Symyx.Notebook.Sections.SamplePreparation.SamplePreparationSection</code>
preparationSectionConcentrationUnit	<code>Symyx.Framework.UnitKey</code> for the calculated concentration
preparationSectionMassUnit	<code>Symyx.Framework.UnitKey</code> for the calculated mass
preparationSectionVolumeUnit	<code>Symyx.Framework.UnitKey</code> for the calculated volume
preparationSectionMaterialRoles	<code>Symyx.Framework.UnitKey</code> for the material roles for the preparation

Event: Component Added

Script variable	Description
e	<code>Symyx.Framework.Collections.ItemEventArgs<IPropertySetHost></code>
row	The row object that was added
sender	<code>Symyx.Notebook.Sections.SamplePreparation.SamplePreparationSection</code>
preparationSectionConcentrationUnit	<code>Symyx.Framework.UnitKey</code> for the calculated concentration
preparationSectionMassUnit	<code>Symyx.Framework.UnitKey</code> for the calculated mass
preparationSectionVolumeUnit	<code>Symyx.Framework.UnitKey</code> for the calculated volume
preparationSectionMaterialRoles	<code>Symyx.Framework.UnitKey</code> for the material roles for the preparation

Event: Removing Component

Script variable	Description
e	<code>Symyx.Framework.Collections.PendingItemEventArgs<IPropertySetHost></code>
row	The row object to be removed
sender	<code>Symyx.Notebook.Sections.SamplePreparation.SamplePreparationSection</code>
preparationSectionConcentrationUnit	<code>Symyx.Framework.UnitKey</code> for the calculated concentration
preparationSectionMassUnit	<code>Symyx.Framework.UnitKey</code> for the calculated mass
preparationSectionVolumeUnit	<code>Symyx.Framework.UnitKey</code> for the calculated volume

Script variable	Description
nit	
preparationSectionMaterialRoles	<code>Symyx.Framework.UnitKey</code> for the material roles for the preparation

Event: Component Removed

Script variable	Description
e	<code>Symyx.Framework.Collections.PendingItemEventArgs<IPROPERTYSetHost></code>
row	The row object that was removed
sender	<code>Symyx.Notebook.Sections.SamplePreparation.SamplePreparationSection</code>
preparationSectionConcentrationUnit	<code>Symyx.Framework.UnitKey</code> for the calculated concentration
preparationSectionMassUnit	<code>Symyx.Framework.UnitKey</code> for the calculated mass
preparationSectionVolumeUnit	<code>Symyx.Framework.UnitKey</code> for the calculated volume
preparationSectionMaterialRoles	<code>Symyx.Framework.UnitKey</code> for the material roles for the preparation

Event: Audit Script

Script variable	Description
e	<code>Symyx.Notebook.Sections.Materials.CustomAuditEventArgs</code>
row	The row object that the audit message applies to
sender	<code>Symyx.Notebook.Sections.SamplePreparation.SamplePreparationSection</code>
preparationSectionConcentrationUnit	<code>Symyx.Framework.UnitKey</code> for the calculated concentration
preparationSectionMassUnit	<code>Symyx.Framework.UnitKey</code> for the calculated mass
preparationSectionVolumeUnit	<code>Symyx.Framework.UnitKey</code> for the calculated volume
preparationSectionMaterialRoles	<code>Symyx.Framework.UnitKey</code> for the material roles for the preparation

Event: Component Row Added

Script variable	Description
e	<code>Symyx.Framework.Properties.ValueChangedEventArgs</code>
row	The row object with added component
sender	<code>Symyx.Notebook.Sections.SamplePreparation.SamplePreparationSection</code>
preparationSectionConcentrationUnit	<code>Symyx.Framework.UnitKey</code> for the calculated concentration
preparationSectionMassUnit	<code>Symyx.Framework.UnitKey</code> for the calculated mass
preparationSectionVolumeUnit	<code>Symyx.Framework.UnitKey</code> for the calculated volume
preparationSectionMaterialRoles	<code>Symyx.Framework.UnitKey</code> for the material roles for the preparation

Event: Dilution Created

Script variable	Description
preparation	<code>Symyx.Notebook.AnalyticalMaterials.Entities.AnalyticalPreparation</code> that has been created
dilution_number	int for one based dilution number
is_serial	boolean which designates whether the dilution is a serial dilution
is_retain_all	boolean which when <code>is_serial</code> is true determines whether all dilutions are retained
owner	<code>Symyx.Notebook.Sections.Table.TableSection</code> context in which this is occurring

Event: Replicate Created

Script variable	Description
preparation	<code>Symyx.Notebook.AnalyticalMaterials.Entities.AnalyticalPreparation</code> that has been replicated
replicate_number	int for one based replicate number
owner	<code>Symyx.Notebook.Sections.Table.TableSection</code> context in which this is occurring

Export Preparation Section Data

The example in this section shows how data from a Preparation Section can be exported into a comma-separated value (CSV) file. In particular this example shows how to:

- Get the active section

```
section = editor.ActiveDocumentSection
```

- Get the property sets that are used in a component of a section, for example, the `SamplePreparationSectionProperty.SelectedPropertySetDefinitions`:

```
from System.Collections.Generic import List as DotNetList
from Symyx.Notebook.Sections.SamplePreparation import
SamplePreparationSectionProperty
from Symyx.Framework.Properties import PropertySetManager

# Get the PSDs for the preparation
preparationPsds = DotNetList[PropertySetDefinition]()
psIds = section.TableSectionProperties.GetValue
(SamplePreparationSectionProperty.SelectedPropertySetDefinitions)
enumerator = psIds.GetEnumerator() while enumerator.MoveNext():
    psId = enumerator.Current
    defn = PropertySetManager.GetDefinition(psId)
    if defn != None:
        preparationPsds.Add(defn)
```

- Go through the property set definitions and read each property key into an array, and then append to a string buffer.

```
from System.Collections import ArrayList
# ArrayList to save the property definition keys
defKeys = ArrayList()
# ArrayList to save all columns headers for later usage
columnKeys = ArrayList()

columnKeys.Clear() defKeys.Clear()
br = StringBuilder()
defIndex = 0
while defIndex < psds.Count:
    # psds is the list of Preparation property set definitions
    psd = psds[defIndex]
    defIndex += 1
    index = 0
    while index < psd.Count:
        pc = psd[index]
        if pc.UserDisplay != UserDisplay.Hidden:
            # pc contains property key as column name
            columnKeys.Add(pc.Key.ToString())
            # psd contains property set key
            defKeys.Add(psd.Key.ToString())
            # Write column name to string buffer
            br.Append(pc.Key.ToString() + ",")

        index += 1
builder.AppendLine(br.ToString().TrimEnd(', '))
```

- Go through the array lists of property set keys and property keys, to get the each property value, and write to a string buffer.

```

index = 0
# columnKeys is ArrayList containing column names
while index < columnKeys.Count:
# defkeys is ArrayList containing property set keys
defky = defkeys[index]
colky = columnKeys[index]
s = GetDataInStringHelper(host, defky, colky)
if not String.IsNullOrEmpty(s):
# replace special characters.
s = s.Replace(",", "','")
s = s.Replace("\r", "|")
s = s.Replace("\n", "|") br.Append(s)

br.Append(",")
index += 1

builder.AppendLine(br.ToString())

def GetDataInStringHelper(host, setky, propertyKy):
try:
# Get the value of propertyKy in setky
p = host.PropertySets[setky][propertyKy]
if p.Value == None or p.ValueIsNull:
return String.Empty else:
s = p.Value.ToString()
return s

except Exception: return
String.Empty

```

The following example is the complete example script that exports Preparation data to a CSV file.

To run this script:

1. Login to Workbook as a user with the SectionTemplate.Editor permission.
2. Create a new experiment template in Experiment Editor.
3. Add a Preparations section. The experiment must include a Preparations section.
4. Add the script to a custom toolbar item.
5. Save the template.
6. Add some data in the Preparations section. When you click the toolbar item, the specified file in the script will be generated (default filename and location is c:\SampleExport.csv).

You can change the script, for example, to export in tab delimited format by changing `br.Append(pc.Key.ToString() + ",")` to `br.Append(pc.Key.ToString() + "\t")`.

```

#=====
# Example to export SamplePreparationSection's data to a csv file.
#=====
import clr
clr.AddReference("System.Drawing")
clr.AddReference("System.Windows.Forms")
from System.Drawing import *
from System.IO import File

```

```
from System import String
from System.Collections.Generic import List as DotNetList
from System.Windows.Forms import *
from System.Text import StringBuilder
from System.Collections import ArrayList
from Symyx.Notebook.AnalyticalMaterials import Entities
from Symyx.Notebook.AnalyticalMaterials.Entities import
AnalyticalPreparation, AnalyticalPreparationStep
from Symyx.Notebook.AnalyticalMaterials import *
from Symyx.Notebook.Sections.SamplePreparation import
SamplePreparationSectionProperty
from Symyx.Framework import Properties
from Symyx.Framework.Properties import Property
from Symyx.Framework.Properties import PropertySetDefinition
from Symyx.Framework.Properties import PropertySet
from Symyx.Framework.Properties import PropertySetKey
from Symyx.Framework.Properties import PropertySetManager
from Symyx.Framework.Properties import UserDisplay

#=====
# Export data in SamplePreparationSection to a file in CSV file format.
# <param name="section">SamplePreparationSection.</param>
# <param name="fileName">Name of the output file.</param>
# <param name="exportComponents">True if export both preparations and
# components. False if only export preparations.</param>
#=====
def ExportToCsvFileFormat(section, fileName, exportComponents):
    if section == None:
        return False
    masterRows = section.GetRows()
    if masterRows == None or masterRows.Length == 0:
        return False

    psIds = None

    # Get the PDSSs for the preparation
    preparationPsds = DotNetList[PropertySetDefinition]()
    psIds = section.TableSectionProperties.GetValue
    (SamplePreparationSectionProperty.SelectedPropertySetDefinitions)
    enumerator = psIds.GetEnumerator()
    while enumerator.MoveNext():
        psId = enumerator.Current
        defn = PropertySetManager.GetDefinition(psId)
        if defn != None:
            preparationPsds.Add(defn)

    # Get the PSDs for the component
    componentPsds = DotNetList[PropertySetDefinition]()
    psIds = section.TableSectionProperties.GetValue
    (SamplePreparationSectionProperty.Analyticals
    tepSelectedPropertySetDefinitions)
    enumerator = psIds.GetEnumerator()
    while enumerator.MoveNext():
```

```

        psId = enumerator.Current
        defn = PropertySetManager.GetDefinition(psId)
        if defn != None:
            componentPsds.Add(defn)

    builder = StringBuilder()

    # export the master columns info.
    preparationDefKeys = ArrayList()
    preparationPropertyKeys = ArrayList()
    CsvColumnsKeys(preparationPsds, preparationDefKeys,
preparationPropertyKeys, builder)

    # export the component columns info if needed.
    componentDefKeys = ArrayList()
    componentPropertyKeys = ArrayList()
    if exportComponents:
        CsvColumnsKeys(componentPsds, componentDefKeys, componentPropertyKeys,
builder)

    # export the data.
    i = 0
    while i < masterRows.Length:
        sample = section.GetRow(i)
        CsvData(sample, preparationDefKeys, preparationPropertyKeys, builder,
String.Empty)

        if exportComponents and section.DetailRowCount(i) > 0:
            for detailRow in section.GetDetailRows(i):
                CsvData(detailRow, componentDefKeys, componentPropertyKeys, builder,
"-")
            i += 1

    #write to file.
    sw = File.CreateText(fileName) sw.Write(builder.ToString())
    sw.Close()
    MessageBox.Show("Finish exporting data to " + fileName + ".")
    return True

#=====
# Exports the columns headers based on input key.
# <param name="defKeys">ArrayList to save the property definition
# keys.</param>
# <param name="columnKeys">ArrayList to save all columns headers
# for later usage.</param>
# <param name="builder">The string builder for saving output.</param>
#=====
def CsvColumnsKeys(psds, defKeys, columnKeys, builder):
    if psds == None or psds.Count == 0:
        return

    columnKeys.Clear() defKeys.Clear()
    br = StringBuilder() defIndex = 0

```

```
while defIndex < psds.Count:
    psd = psds[defIndex]
    defIndex += 1
    index = 0
    while index < psd.Count:
        pc = psd[index]
        if pc.UserDisplay != UserDisplay.Hidden:
            columnKeys.Add(pc.Key.ToString())
            defKeys.Add(psd.Key.ToString())
            br.Append(pc.Key.ToString() + ",")

        index += 1
    builder.AppendLine(br.ToString().TrimEnd(', '))

#=====
# Exports the columns data
#
# <param name="host">The host - it can be the preparation or the
# component.</param>
# <param name="defKeys">ArrayList to save the property definition
# keys.</param>
# <param name="columnKeys">ArrayList. See CsvColumnsKeys for
# detail.</param>
# <param name="builder">The string builder.</param>
# <param name="upFront">Optional string before the component
# data.</param>
#=====
def CsvData(host, defKeys, columnKeys, builder, upFront):
    if host == None or columnKeys == None or columnKeys.Count == 0:
        return

    if columnKeys.Count != defKeys.Count:
        return

    br = StringBuilder()
    br.Append(upFront)
    index = 0
    while index < columnKeys.Count:
        defky = defKeys[index]
        colky = columnKeys[index]
        s = GetDataInStringHelper(host, defky, colky)
        if not String.IsNullOrEmpty(s):
            # replace special characters.
            s = s.Replace(",", "','")
            s = s.Replace("\r", "|")
            s = s.Replace("\n", "|")
            br.Append(s)

        br.Append(",")
        index += 1

    builder.AppendLine(br.ToString())
```

```
#=====
# Helper
#=====
def GetDataInStringHelper(host, setKy, propertyKy):
    try:
        p = host.PropertySets[setKy][propertyKy]
        if p.Value == None or p.ValueIsNull:
            return String.Empty
        else:
            s = p.Value.ToString()
            return s

    except Exception:
        return String.Empty
#=====
# This script is executed from the DynamicToolBar in the section.
# The section is an instance of SamplePreparationSection.
#=====
section = editor.ActiveDocumentSection
ExportToCsvFileFormat(section, "c:\\SampleExport.csv", True)
```

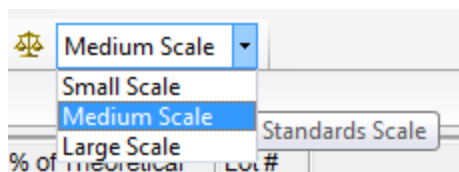
Data Exported as CSV File

The following is the exported comma-separated-value (csv) data from the Preparation Section:

```
Name,Density,Structure,Role,Comments,InitialAmount,MW,MF,
FormattedMolFormula,Planned Amount,TotalAmount,FinalpH,Notes,
PreparationDate,LotNumber,Manufacturer,Grade,Expiry Date,
CorrectionFactor,SampleId,TargetReference Material,Description,
PlanAmount,Adjust,PlanConcentration,PlanMass,PlanVolume,
PlanDrugConcentration,ActualAmount,AdditionalActualAmount,
AdditionalAmountLot,ActualConcentration,DrugConcentration,
DilutionFactor
Calculation3,,,,,,,,,1500 mL,,,,,,,,,
-water,,,Total Volume,,,,,,,,,
-sample1,,,,,,,,,52.4455 mg,,,,,,,,,
```

Change the Scale Used In Calculations

You can customize the combination of units used for calculation values in a Formulation section. Suppose you want to use milligram and liter, which is a unit combination that is not available in the existing scales.



- Specify names for the custom values to name that are meaningful to your team.
- You can change the unit keys you specify such as UnitKey.MILLIGRAM and UnitKey.LITRE. The unit key value must use the UnitKey enum type. For more information, see List of Units to choose from.
- Use the same units defined for the mass and volume as used in the get_event_handler and units_match methods.

To change the scale:

1. In Workbook, open a Formulation Experiment and save it as a **Template**.
2. Select the **Formulation** section, and click **View > Properties**.
3. On the **Experiment** tab, in **Event Scripting**, select **Scripts**.
4. Select the **OnApplicationLoaded** event, and paste in a script that specifies a custom combination of units. For more information, see Custom Scale.
5. Click **OK** and save the template.
6. Reload the application to fire the OnApplicationLoaded event.
The OnApplicationLoaded event fires when you reload the application.
7. Verify that your custom combination of units is available.

Script for Custom Scale

You can customize the combination of units used for calculation values in a Formulation section. Suppose you want to use milligram and liter, which is a unit combination that is not available in the existing scales.

```
from Symyx.Framework import UnitKey
from Symyx.Notebook.Sections.Formulation import FormulationSection
from System.Windows.Forms import MessageBox

control = "ComboBox"
newScaleName = "MILLIGRAM, LITER"
propertySetDefinition = "FormulationSection"
propertySetDefinition_MassKey = "CalculatedMassUnits"
propertySetDefinition_VolumeKey = "CalculatedVolumeUnits"

def get_event_handler(formulationSection):
    def toolStripComboBox_SelectedIndexChanged(sender, e):
        if (sender.SelectedItem.ToString().Equals(newScaleName)):
            if (formulationSection.PropertySets.Contains(propertySetDefinition)
and formulationSection.PropertySets[propertySetDefinition].Contains
(propertySetDefinition_MassKey)):

(formulationSection.PropertySets[propertySetDefinition]
[propertySetDefinition_Mas sKey]).Value = UnitKey.MILLIGRAM
            if (formulationSection.PropertySets.Contains(propertySetDefinition)
and formulationSection.PropertySets[propertySetDefinition].Contains
(propertySetDefinition_VolumeKey)):

(formulationSection.PropertySets[propertySetDefinition]
[propertySetDefinition_Vol umeKey]).Value = UnitKey.LITER
            #formulationSection.View.RefreshAllDetailView() return
toolStripComboBox_SelectedIndexChanged

def get_formulation_sections():
    return [section for section in owner.Sections if isinstance(section,
FormulationSection)]

def units_match(formulationSection): return
formulationSection.PropertySets[propertySetDefinition]
```



```
[propertySetDefinition_Mass Key].Value == UnitKey.MILLIGRAM \
    and formulationSection.PropertySets[propertySetDefinition]
[propertySetDefinition_VolumeKey].Value == UnitKey.LITER

def configure_toolbar(formulationSection):

    for bar in formulationSection.View.ToolBars:
        for item in bar.Items:
            if(item.GetType().ToString().Contains(control)):
                comboBox = item

                comboBox.Items.Add(newScaleName)
                if units_match(formulationSection):
                    comboBox.Text = newScaleName
                    comboBox.SelectedIndexChanged += get_event_handler(formulationSection)
    sections = get_formulation_sections()
    for section in sections:
        configure_toolbar(section)
```

Change the Scale of Calculated Values for a Formulation

You can customize the combination of units used for calculation values in a Formulation section. Suppose you want to use milligram and liter, which is a unit combination that is not available in the existing scales.

1. In Workbook, open a Formulation Experiment, and save it as a Template.
2. Click the Formulation section, and click **View > Properties**.
3. On the Experiment tab, under Event Scripting, click **Scripts**.
4. Select the **OnApplicationLoaded** event and paste in a script that specifies a custom combination of units. For more information, see [Script for Custom Scale](#).
5. Click OK and save the template.
6. Reload the application to fire the OnApplicationLoaded event.
7. Verify that your custom combination of units such as MILLIGRAM and LITER is available.
8. In this example, newScaleName = "MILLIGRAM, LITER" was used, but you can set newScaleName to any name you want.
9. The unit keys you specify such as UnitKey.MILLIGRAM and UnitKey.LITRE can be changed, but the values must be of the UnitKey enum type. For more information, see the section entitled List of Units to choose from.
10. The units defined for the mass and volume must be same in the get_event_handler and units_match methods.

Script for Custom Scale

You can customize the combination of units used for calculation values in a Formulation section. Suppose you want to use milligram and liter, which is a unit combination that is not available in the existing scales.

Unit Types

Chose appropriate units from the UnitKey enum.

Unit Type	Description
	UNDEFINED
Mass Units	MILLIGRAM, GRAM, MICROGRAM, KILOGRAM
Volume Units	MICROLITER, MILLILITER, LITER
Mass Ratio	GG,MGMG, MILLIGRAMKG, NANOGRAMKG, MICROGRAMKG, MASSPERCENT, GRAMPERHUNDREDGRAM, MASSFRACTION, MILLIGRAMG, KGPERKG, KGPERG,KGPERHUNDREDGRAM, GRAMPERKG

DataCreation Example

Review this example to learn how to add data to Reaction Scheme and Synthetic Chemistry sections in a document. The .NET solution for the Program.cs source code of this sample project is installed with the Workbook SDK in the Symyx.SDK.Samples.DataCreation example in the samples folder.

For details on how to set up and run the Symyx.SDK.Samples.DataCreation sample, navigate to the sample directory and open TemplateCreation.mht in Microsoft Internet Explorer.

Program.cs contains the Main method, which logs into Vault, gets the template, creates a new document that is populated with a ReactionScheme, and adds that document to Vault.
`static void Main(string[] args).`

```
{
// Begin by enabling the application to get from vault the latest
// versions of workbook and Framework assemblies.
AssemblyCache.Initialize();
// Get the command line args, such as TemplateVaultPath.
// The args can be any of the ApplicationSettings ParseArguments(args);
// Set storage as offline or roaming from the application settings.
UserRepository.DefaultUserRepositoryStorageOption =
(UserRepository.UserRepositoryStorageOptions)Enum.Parse(typeof
(UserRepository.UserRepositoryStorageOptions),
ApplicationSettings.Default.UserRepositoryStorage);
// Authenticate the user from either application settings or the command
line.
LoginToVaultServer(ApplicationSettings.Default.Server,
ApplicationSettings.Default.Username, ApplicationSettings.Default.Password);
// Set the template from the commandline or the applications settings.
Document template = GetTemplate();
// CreateDocument is most important method in this application.
Document document = CreateDocument(template);
// Finally, store the newly populated document to vault in the folder
// that the commandline arg sets to either vaultId or vaultFolder.
AddDocumentToVault(document);
}
```

CreateDocument Method

The CreateDocument method creates an empty document to import data into, and then fills in that document by calling:

```
PopulateFormsSection(document);
PopulateMaterialsSection(materialsSection);
PopulateReactionSchemeSection(reactionSchemeSection, materialsSection);
AddNewTextSectionWithVariedRtf(document);
AddNewTextSectionWithRtfContainingImages(document);
AddNewTextSectionWithSamplePlainText(document);
AddNewTextSectionWithImage(document);
AddNewTextSectionWithStructure(document);
AddNewTextSectionWithTextAndStructure(document);
AddFileSectionWithAllDocumentTypes(document);
```

Here is the C# code:

```
public static Document CreateDocument(Document template)
{
    Console.WriteLine("Creating experiment"); Document document =
    Document.Create(template); document.Title = String.Format("{0}-{1}",
    typeof(Program).FullName, Guid.NewGuid()); PopulateFormsSection(document);
    var materialsSection = (MaterialsSection)document.Sections.Find(ds =>
    ds.Title
    == CHEMISTRY_SECTION_NAME);
    PopulateMaterialsSection(materialsSection); var reactionSchemeSection =
    (ReactionSchemeSection)FindSingleSectionByTitle(document, REACTION_
    SECTION_NAME); PopulateReactionSchemeSection(reactionSchemeSection,
    materialsSection);
    AddNewTextSectionWithVariedRtf(document);
    AddNewTextSectionWithRtfContainingImages(document);
    AddNewTextSectionWithSamplePlainText(document); AddNewTextSectionWithImage
    (document); AddNewTextSectionWithStructure(document);
    AddNewTextSectionWithTextAndStructure(document);
    AddFileSectionWithAllDocumentTypes(document);
    Console.WriteLine("Creating experiment...done"); return document;
}
```

Reaction Scheme C# Examples

The following C# example shows how to:

- Find the reaction scheme in a document
- List the contents
- Add a reaction
- Add and remove reaction steps
- Remove the reaction scheme

```
public void DocumentationTest()
{
    Document template = new Document();
    template.Add(new ReactionSchemeSection(){Title = "My Scheme"});
    // To add a material section that could be linked to
    // the reaction scheme section,
```

```

    template.Add(new MaterialSection());
    Document document = Document.Create(template);
    // Finding the reaction scheme in a document by looking in the
    // document sections, which is what ds stands for
    ReactionSchemeSection schemeSection = document.Find(ds => ds.Title == "My
Scheme") as ReactionSchemeSection;
    //Listing the contents
    IList<ReactionStep> stepsToBeRemoved=new List<ReactionStep>();
    if (schemeSection != null)
    {
        ChemistryModel.ReactionScheme scheme = schemeSection.ReactionScheme;
        string rxnFormatOne = ResourceLoader.LoadResource
("Symyx.Notebook.Sections.ReactionScheme.Tests.Reactions.
MultiStep.Experiment_186_1.Step1.rxn");
        string rxnFormatTwo = ResourceLoader.LoadResource
("Symyx.Notebook.Sections.ReactionScheme.Tests.Reactions.
MultiStep.Experiment_186_1.Step2.rxn");
        // In this case, we have aliased a namespace: using
        //ChemistryModel = Symyx.Notebook.Sections.ReactionScheme.Chemistry;
        ChemistryModel.ReactionStep reactionStepOne = new
ChemistryModel.ReactionStep();
        reactionStepOne.Reaction.ReactionFile = rxnFormatOne;
        // adding a reaction step
        scheme.Add(reactionStepOne);
        ChemistryModel.ReactionStep reactionStepTwo = new
ChemistryModel.ReactionStep(); reactionStepOne.Reaction.ReactionFile =
rxnFormatTwo;
        scheme.Add(reactionStepTwo);
        // Removing reaction steps:
        // if the reaction steps has more than ten children
        // (reactants or products), remove its steps
        foreach (var step in scheme.Steps)
        {
            if(step.ChildrenCount>10)
            {
                stepsToBeRemoved.Add(step);
            }
        }
        foreach (var step in stepsToBeRemoved)
        {
            scheme.Remove(step);
        }
    }
    // clear the reaction scheme
    scheme.Clear();
}
}

```

Locate the Reaction Scheme Section

The CreateDocument method, called by Main, locates the reaction scheme section by calling FindSingleSectionByTitle.

```

public static Document CreateDocument(Document template)
{

```

```

Console.WriteLine("Creating experiment");
Document document = Document.Create(template);
document.Title = String.Format("{0}-{1}", typeof(Program).FullName,
Guid.NewGuid()); PopulateFormsSection(document);
var materialsSection = (MaterialsSection)document.Sections.Find(ds =>
ds.Title == CHEMISTRY_SECTION_NAME);
PopulateMaterialsSection(materialsSection);
var reactionSchemeSection =
(ReactionSchemeSection)FindSingleSectionByTitle(document, REACTION_SECTION_
NAME);

```

Locate a Reaction Step

The CreateDocument method calls the PopulateReactionSchemeSection method, which locates a reaction step by calling Find for a step in the ReactionStep list.

```

public static void PopulateReactionSchemeSection(ReactionSchemeSection
reactionSection, MaterialsSection materialsSection)
{
    var step1 = new List<ReactionStep>
(reactionSection.ReactionScheme.Steps).Find(s=> s.Name == "Step 1");
    step1.Description = "Adenine alkylation in DMF, 140 C";
    step1.DisplayReaction = ReadFile(@"Resources\HPA.rxn");
    ReactionParticipantStep step1Reactant1 = newList<ReactionParticipantStep>
(step1.Reactants)[0];

```

Add a Reaction Step

Use Find because we already have a first step:

```

var step1 = new List<ReactionStep>
(reactionSection.ReactionScheme.Steps).Find(s => s.Name == "Step 1");

```

You must construct the other steps.

```

var step2 = new ReactionStep();

```

If your template already has steps, you can find them, and do not need to construct them.

Add a Reaction From a File

Use Find to get the first step.

```

var step1 = new List<ReactionStep>
(reactionSection.ReactionScheme.Steps).Find(s => s.Name == "Step 1");
step1.Description = "Adenine alkylation in DMF, 140 C";
step1.DisplayReaction = ReadFile(@"Resources\HPA.rxn");
ReactionParticipantStep step1Reactant1 = new List<ReactionParticipantStep>
(step1.Reactants)[0]; [...]

```

Assign the content of the reaction file to the DisplayReaction property, and then import legacy data to the row.

PopulateReactionSchemeSection adds a reaction file by reading in the rxnfile to a DisplayReaction, and then calling ModifyRow to update the fields.

```

step2.DisplayReaction = ReadFile(@"Resources\PMPA diethyl ester.rxn");
ReactionParticipantStep step2Reactant1 = new List<ReactionParticipantStep>
(step2.Reactants)[0];
ModifyRow(step2Reactant1.Material, "(2R)-1-(6-aminopurin-9-yl)propan-2-ol",

```

```
null, null, new Quantity(95.50, 4, UnitKey.MASSPERCENT), true, null, "C",
null, new Quantity(190.0, 4, UnitKey.MILLIGRAM), new Quantity(190.0, 4,
UnitKey.MILLIGRAM), new Quantity(0.9391, 5, UnitKey.MILLIMOLE), new Value
(1.000, 4), new Measurement(new Value(190.0, 4), UnitKey.MILLIGRAM), new
Quantity(0.9391, 5, UnitKey.MILLIMOLE));
ReactionParticipantStep step2Reactant2 = new List<ReactionParticipantStep>
(step2.Reactants)[1];
ModifyRow([...])
```

Link Corresponding Materials Section

The reaction scheme section can have a linked materials section.

The `CreateDocument` method calls the `PopulateReactionSchemeSection` method that associates a `reactionSchemeSection` with a `materialsSection`, as shown in the following example:

```
PopulateReactionSchemeSection(reactionSchemeSection, materialsSection);
```

Add a Material Using AddRow

Reaction and materials are linked as follows:

```
var materialsSectionMaterial = materialsSection.Materials.Find
(participantStep.MaterialId);
```

The first row for materials exists in the Materials section. As a result, Workbook uses the `ModifyRow` method modify to enable adding the material details. You can also use the `AddRow` method, as follows:

```
public static void PopulateMaterialsSection(MaterialsSection
materialsSection)
```

Modify a Material

The reaction file puts some of the values in the materials row. To import from a legacy location, the data about a reaction file that is external to the rxnfile.

You can use the `PopulateReactionSchemeSection` method to modify the material used in a reaction step.

```
public static void ModifyStepMaterial(ReactionSchemeSection reactionScheme,
MaterialsSection materialsSection, ReactionParticipantStep participantStep,
string name, string structureFilename, string role, Quantity? purity, bool?
lr, int? sc, string label, string step, Quantity? planAmount, Quantity?
planCalcMass, Quantity? planCalcMoles, value? planCalcEquiv, Measurement
actAmount, Quantity? actCalcMoles)
```

Chapter 10:

Build and Debug a Custom .NET Assembly

When building your custom .NET assembly in an IDE, you might need to debug the assembly.

To debug your assembly:

- Set the Build Output Path to the folder containing the `Symyx.Notebook.Application.exe`.
- Make the `Symyx.Notebook.Application.exe` the start up program in the IDE.
- Run the project from within the IDE to launch Workbook.
- Set a breakpoint in the IDE, run the scripts that calls your assembly.

The IDE stops before it executes the breakpoint line.

- From within the debugger, you can read all of the properties of the objects passed to your method.

Visual Studio supports Edit and Continue that enables typing and executing changes without the need to stop the debugging session and re-compile.

The open source `#develop` enables editing from within the debugger, but does not execute the changes until you have stopped debugging and recompiled them.

IronPython Script For Calling a Custom Assembly Example

The following example illustrates using an IronPython script to call a custom .NET assembly. The example shows importing custom records into any section based on a Materials section such as a section for Synthetic Chemistry, Analytical Materials, Formulation Materials, or Recipe Materials.

Set Up the Example

To set up the example:

1. Create a Microsoft Visual Studio class library project named `TestImportHooks` with a class named `ImportDataTestClass`.
2. Add the following References:
 - `Symyx.Framework`
 - `Symyx.Framework.Controls`
 - `Symyx.Notebook`
 - `Symyx.Notebook.Sections.Table`
 - `symyx.windows`
 - `System System.Core System.Data`
 - `System.Data.DataSetExtensions System.Windows.Forms System.Xml`
 - `System.Xml.Linq`

3. Paste the following code:

```
using System;
using Symyx.Windows;
using System.Windows.Forms; using Symyx.Framework.Vault;
using Symyx.Notebook.Sections.Table;
```

```
namespace TestImportHooks
```

```
{
    /// <summary>
    /// Demonstrate that the hooks allow us to attach a menu item to the
    /// import and to use a click event on that item to import data.
    /// The section used MUST be a table section with the sample
    /// identification PSD applied.
    /// Compile with .NET Framework 3.5.1 or 4.5.2 into an assembly named
    /// TestImportHooks.
    /// </summary>
    public class ImportDataTestClass : ToolStripMenuItem
    {
        private int invokeCount_;
        private TableSection tableSection_;

        /// <summary>
        /// Set up the menuItem
        /// </summary>
        /// <param name="section"></param> public void Setup(TableSection
        section)
        {
            tableSection_ = section;
            Text = "Import From TestCase"; ToolTipText = "Click me to import
            data"; Click += ImportHooksTestClass_Click;
        }

        /// <summary>
        /// Demonstrate import using external code.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        void ImportHooksTestClass_Click(object sender, EventArgs e)
        {
            var view = tableSection_ != null ? tableSection_.View as
            DefaultTableSectionView : null;

            if (view != null)
            {
                read
                // Accumulate a series of records to import
                // from file or a service. The data is
                // an integer that is incremented.
                ++invokeCount_; // increment the number to display for new row

                // You might use
                // fixed mapping like this or use a mapping dialog.

                var row = view.Table.CreateBlankRow(); if (row != null)
                {
                    apply
                }
            }
        }
    }
}
```



```

        display a number
// Iterate through the collected records,
// the mapping, and add the values row.PropertySets.SetValue("SampleId",
invokeCount_); //
        var newRow = view.Table.AddRow() as VaultObject; if (newRow != null)
        {
            newRow.CopyFrom(row as VaultObject);
        }
    }
}
// Make the most recently used menu item the default
view.SetAsDefaultImportButton(this);
}
}

/// <summary>
/// Clean up.
/// </summary>
/// <param name="disposing"></param>
protected override void Dispose(bool disposing)
{
    base.Dispose(disposing);>
    Click -= ImportHooksTestClass_Click;
}
}
}

```

4. Copy the newly created `TestImportHooks.dll` to the bin subdirectory of your Workbook client application.
5. Add the IronPython script that calls your custom assembly. Use the following:

```

import System
import clr
aso = System.Reflection.Assembly.LoadFrom('TestImportHooks.dll')
clr.AddReference('TestImportHooks')
from TestImportHooks import ImportDataTestClass
menuItem = ImportDataTestClass()
menuItem.Setup(sender.ActiveDocumentSection)
sender.ActiveDocumentSection.View.AddToImportDropDown(menuItem)

```

6. Select the `OnSectionActivated` event.

To restrict the script to a specific section, your code must check the name of the section. This example does not check the section name.

Test the Example

To test the example:

1. In Workbook, click the Synthetic Chemistry section.
2. Click the Import From list, and select Import From TestCase.

A row is created that displays the incremented row count in the leftmost column.

Note: The item most recently chosen from the Import From list becomes the default.

C# Code for Importing Custom Data

```
using System;
using Symyx.Windows;
using System.Windows.Forms;
using Symyx.Framework.Vault;
using Symyx.Notebook.Sections.Table;

namespace TestImportHooks
{
    /// <summary>
    /// Demonstrate that the hooks allow us to attach a menu item to the
    /// import and to use a click event on that item to import data.
    /// The section used MUST be a table section with the sample
    /// identification PSD applied.
    /// Compile with .NET Framework 3.5.1 or .NET 4.5.2 into an assembly named
    /// TestImportHooks.
    /// </summary>
    public class ImportDataTestClass : ToolStripMenuItem
    {
        private int invokeCount_;
        private TableSection tableSection_;

        /// <summary>
        /// Set up the menuItem
        /// </summary>
        /// <param name="section"></param>
        public void SetUp(TableSection section)
        {
            tableSection_ = section;
            Text = "Import From TestCase";
            ToolTipText = "Click me to import data";
            Click += ImportHooksTestClass_Click;
        }

        /// <summary>
        /// Demonstrate import using external code.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        void ImportHooksTestClass_Click(object sender, EventArgs e)
        {
            var view = tableSection_ != null ? tableSection_.View as
DefaultTableSectionView : null;

            if (view != null)
            {
                read
                // Accumulate a series of records to import from file or a
                // service. Use the following data:
                // an integer we increment.
                ++invokeCount_;
                // increment the number to display for new row
            }
        }
    }
}
```

```

// You might have fixed mapping like this or a mapping dialog.
var row = view.Table.CreateBlankRow();
if (row != null)
{
    // Here you would iterate through the collected records,
    // apply the mapping and add the values
    row.PropertySets.SetValue("SampleId", invokeCount_);
    // display a number
    var newRow = view.Table.AddRow() as VaultObject;
    if (newRow != null)
    {
        newRow.CopyFrom(row as VaultObject);
    }
}
// Make the most recently used menu item the default
view.SetAsDefaultImportButton(this);
}
}

/// <summary>
/// Clean up
/// </summary>
/// <param name="disposing"></param>
protected override void Dispose(bool disposing)
{
    base.Dispose(disposing);
    Click -= ImportHooksTestClass_Click;
}
}
}

```

Publish a Custom .NET Assembly

After you have tested your .NET assembly and profiled its performance, see [Script Performance Profile](#), you can publish the .NET assembly for deployment. Publishing an assembly upload the assembly to Vault. Once published the custom assembly is immediately accessible by all Workbook client computers, use caution when publishing changes to your production server.

Call an External Assembly with IronPython

```

import System
import clr
aso = System.Reflection.Assembly.LoadFrom('TestImportHooks.dll')
clr.AddReferenc('TestImportHooks')
from TestImportHooks import ImportDataTestClass
menuItem = ImportDataTestClass()
menuItem.SetUp(sender.ActiveDocumentSection)
sender.ActiveDocumentSection.View.AddToImportDropDown(menuItem)

```

AssemblyCache.Publish Method

You can create a program that uses the `AssemblyCache.Publish` method to publish your assemblies to BIOVIA Vault Server.

1. Create a console application and change the target .NET version to .NET 3.5.1 or 4.5.2. Set the Target CPU to X86.
2. Add the following Workbook assemblies to the bin folder of your .NET assembly:
 - `Symyx.Framework.dll`
 - `Accelrys.AEP.Authentication.dll`
3. Add the following lines of code to the application:

In Visual Basic .NET:

```
Imports Symyx.Framework.Vault
Imports Symyx.Framework.Extensibility

Module Module1
    Sub Main()
        Login()
        Publish()
    End Sub

    Sub Publish()
        Dim assembly = System.Reflection.Assembly.LoadFile("D:\Projects\Visual
Studio\CompanyName.ProjectName\CompanyName.ProjectName\bin\Debug\Company
Name.ProjectName.dll")
        If (AssemblyCache.IsPublished(assembly)) Then
            Throw New ApplicationException("Assembly has already been
published. Increment the assembly version number and try again.")
        Else
            AssemblyCache.Publish(assembly)
        End If
    End Sub

    Sub Login()
        Dim workspace = New VaultWorkspace("servername")
        Dim loginState = workspace.Login("domain\username", "password")

        If (loginState = AuthenticationState.Yes) Then
            workspace.MakeCurrentWorkspace()
        Else
            Throw New ApplicationException("Login failed.")
        End If
    End Sub

End Module
```

In C#

```
using System;
using Symyx.Framework.Extensibility;
using Symyx.Framework.Vault;
```

```

namespace CompanyName.AssemblyPublisher
{
    class Program
    {
        public static void Main(string[] args)
        {
            LogIn();
            Publish();
        }

        private static void Publish()
        {
            var assembly = System.Reflection.Assembly.LoadFile
(@"D:\Projects\Visual
Studio\CompanyName.ProjectName\CompanyName.ProjectName\bin\Debug\Company
Name.ProjectName.dll");
            if (AssemblyCache.IsPublished(assembly))
            {
                throw new ApplicationException(@"Assembly has already been
published. Increment the assembly version number and try again.");
            }
            else
            {
                AssemblyCache.Publish(assembly);
            }
        }

        private static void LogIn()
        {
            var workspace = new VaultWorkspace(@"servername");
            var loginState = workspace.Login(@"domain\username", @"password");

            if (loginState == AuthenticationState.Yes)
            {
                workspace.MakeCurrentWorkspace();
            }
            else
            {
                throw new ApplicationException(@"Login failed");
            }
        }
    }
}

```

4. Replace the assembly path with the path to your assembly.
5. Replace server name, domain\user name, and password with your login credentials.
6. After your compile and build the console application, you can use the application to publish your assembly to BIOVIA Vault Server.

List Assemblies in Vault

Assemblies published to Vault are stored in the Site repository as VaultObjects whose Titles are the fully qualified type names of the assemblies. You can use a Console Application to list the assemblies in the Site repository similar to the one you used for publishing assemblies in with the [AssemblyCache.Publish](#) method. This example uses the ListAssemblies method:

Visual Basic .NET:

```
Sub ListAssemblies()  
Dim assemblies = VaultWorkspace.Current.SiteRepository.Get  
(VaultObjectTypes.Assembly, DataScope.Minimal)  
  
For Each assembly As VaultObject In assemblies Console.WriteLine  
(assembly.Title)Next  
  
Console.WriteLine("Press any key to continue...") Console.ReadKey()  
  
End Sub
```

C#:

```
private static void ListAssemblies()  
{  
    var assemblies = VaultWorkspace.Current.SiteRepository.Get  
(VaultObjectTypes.Assembly, DataScope.Minimal);  
  
    foreach (var assembly in assemblies)  
    {  
        Console.WriteLine(assembly.Title);  
    }  
    Console.WriteLine(@"Press any key to continue..."); Console.ReadKey  
();  
}
```

Publishing Referenced .NET Assemblies

When you are using a .vozip file or using the AssemblyCache.Publish method to publish your assembly, if your assembly contains a reference to another .NET assembly, then publishing your assembly also publishes the referenced assembly.

When the Workbook client loads the CompanyName.ProjectName.dll, the CompanyName.ProjectName.CommonFunctions.dll is also downloaded.

If the referenced version of CompanyName.ProjectName.CommonFunctions.dll already exists in Vault, then CompanyName.ProjectName.CommonFunctions.dll is not published, and CompanyName.ProjectName.dll is linked with the existing CompanyName.ProjectName.CommonFunctions.dll. The Workbook client loads that CompanyName.ProjectName.CommonFunctions.dll with CompanyName.ProjectName.dll.

When publishing assemblies it is important that all referenced assembly version numbers are correct. If CompanyName.ProjectName.CommonFunctions.dll has changed, then apply new version number at the time you publish CompanyName.ProjectName.dll.

Publish a New Version of a .NET Assembly

To make changes available to end users, you must publish the new version. Do this from the Assembly Information screen in Visual Studio.

Note: By default, #develop generates a new version number with every compile. The same version number cannot be used more than once.

Unpublish an Assembly

Vault has no facility to remove or unpublish a .NET assembly. For this reason and others, it is important to maintain a copy of the source code for every .NET assembly version that you publish. To back out a published version, get the copy of the earlier version, and then recompile and republish that earlier version. When you recompile the earlier version, it is assigned a new version number.

List of Assemblies in Vault

Assemblies published to Vault are stored in the Site repository as VaultObjects fully qualified type names of the assemblies as titles. You can use a console application to list the assemblies in the Site repository similar to the one you used for publishing assemblies.

In Visual Basic .NET

```
Sub ListAssemblies()

    Dim assemblies = VaultWorkspace.Current.SiteRepository.Get
    (VaultObjectTypes.Assembly, DataScope.Minimal)

    For Each assembly As VaultObject In assemblies
        Console.WriteLine(assembly.Title)
    Next

    Console.WriteLine("Press any key to continue...")
    Console.ReadKey()

End Sub
```

In C#

```
private static void ListAssemblies()
{
    var assemblies = VaultWorkspace.Current.SiteRepository.Get
    (VaultObjectTypes.Assembly, DataScope.Minimal);

    foreach (var assembly in assemblies)
    {
        Console.WriteLine(assembly.Title);
    }
    Console.WriteLine(@"Press any key to continue...");
    Console.ReadKey();
}
```

Chapter 11:

Workflow Designer

A workflow is a sequence of activities that achieve a specific organizational goal. For example, a workflow can define how documents are reviewed, approved, and archived. A workflow can also send notifications to reviewers and create a task list for reviewers.

The Workflow Designer allows you to create a workflow. The workflow appears in the design pane, the toolbox pane contains the workflow activities, and the properties pane shows an activity's properties. For more information, see the Workflow Designer Help.

A workflow contains a number of activities. Example activities include `InvokeWebServiceActivity` and `HandleExternalEventActivity`. The `ActivityBase` class is the base class and is the fundamental building block for the activities available in Workflow Designer. You can create your own custom workflow activities to perform a task that is not present in the supplied activities.

Vault objects

Workflow interacts with Vault workspaces and other objects. The SDK Help describes the methods and properties for working with a Vault workspace, see the `Symyx.Framework.Vault.Workspace` class, and the Vault objects, see the `Symyx.Framework.Vault.VaultObject` class in the.

Custom Workflow Activities

You create a custom workflow activity to perform a task that cannot be done using one of the standard activities listed in the Workflow Designer toolbox. The example in this section shows a simple custom workflow activity that sets a Vault object description, and illustrates how you use the C# workflow activity template that ships with the Symyx Framework SDK.

For more information, see how to send a message to a Microsoft MSMQ queue in the product documentation. For more information about Vault workflows, see Vault Workflow in the *Vault Server Administrative Tools Guide*.

Prerequisites

You should be familiar with:

- Microsoft Visual Studio
- Microsoft C# programming
- Microsoft Windows system administration
- The Workflow Designer
- The Administration Console
- Notebook Explorer
- The SDK

Before you can create a custom workflow activity, you must install:

- Microsoft .NET Framework 3.5 SP1 or 4.5.2
- Microsoft Visual Studio 2010 or later.

- The SDK
- The Workflow Designer

If you want to run the example, you must also have access to a server running:

- Vault
- Microsoft MSMQ

Create a Custom Workflow Activity

The [OnExecute](#) method in the code contains the custom activity functionality. The OnExecute method is called when the activity is run by the Vault Workflow service. The class also contains a call to the [GetLogger](#) method that writes messages to the log. For more information, see [Change the Logging Level](#).

Workflow Designer requires placing your dll in the C:\Program Files\Symyx\SymyxWorkflow Designer<version_number>\Designer folder.

The `Symyx.workflow.ExampleActivity` custom workflow activity is located in the SDK samples folder.

Note: If a custom workflow activity name does not end with the word *Activity*, an error displays when you open the workflow in the Workflow Designer.

1. Start Visual Studio.
2. Choose **File > New > Project**.
3. Verify that the **Project types** tree is expanded, and the Visual C# > Symyx > `Symyx.workflow.ExampleActivity` is visible.

Note: The `Symyx.workflow.ExampleActivity` template is added after you install the Symyx Framework SDK.

4. Click `Symyx.workflow.ExampleActivity`.
5. Enter a **Name**, for example, `Symyx.workflow.MyExampleActivity`.
The name of the activity assembly must match the pattern of `Symyx.Workflow.*Activity`. The file names `Symyx.workflow.MyExampleActivity.dll` and `Symyx.workflow.PutStageInFormActivity.dll` are valid, but a name like `Symyx.SDK.workflow.MyExampleActivity.dll` is not valid.
6. Specify a **Location**, for example, `C:\MyExampleActivity`.
7. Verify that the **Solution Name** is correct, for example, `Symyx.workflow.MyExampleActivity`.
8. Click **OK**.
9. In the **Solution Explorer**, right-click **ExampleActivity.cs**, and change the file name to `MyExampleActivity.cs`.
10. In the Solution Explorer, double-click **Properties**.
11. Click **Application**.
12. Verify that the Assembly name is set to `Symyx.workflow.MyExampleActivity`.
13. Verify that the **Default namespace** is set to `Symyx.workflow`.
14. Verify that the **Target Framework** is set to .NET Framework 3.5.1 or 4.5.2
15. In the Solution Explorer, right-click **MyExampleActivity.cs**, and choose **View Code**.

16. Examine the [code](#) that contains the definition of the `MyExampleActivity` class.
17. Choose **File > Save All**.
18. In the Solution Explorer, double-click **Properties**.
19. Click **Application > Assembly Information**.
20. Set the title, for example, `MyExampleActivity`.
21. Set the left three numbers of the Assembly Version number to 1.0.0.
22. Set the right number to 1.

Note: Every time you compile a new version of the assembly included in a published workflow, you must increment the right number of the version by one to ensure that the latest dll is loaded for your activity by the Vault Workflow service.

23. Click **OK**.
24. In the Solution Explorer, double-click **Properties > Signing**.
25. Select **Sign the assembly**.
26. Choose **New**, enter the file name, for example, `MyExampleActivity.snk`.
27. Remove the check from the password protection option, and click **OK**.
28. Verify that **Delay sign only** is not selected.

Configure the Build Location for Your DLL

To configure Visual Studio to build the dll directly in the `\\Workflow Designer\Designer` folder:

1. In the Solution Explorer, double-click **Properties**.
2. Choose **Build**.
3. Set the **Output path** to `C:\Program Files\Symyx\Symyx workflow Designer <version_number>\Designer`.
4. Click **OK**.

Note: If you do not want to set the Output path as specified in the previous steps, you must copy the activity DLL to `C:\Program Files\Symyx\Symyx workflow Designer <version_number>\Designer` after you have built the dll. You might want to do that if you are running Workflow Designer on a different computer from Visual Studio.

5. Choose **File > Save All**.
6. Choose **Build > Build Solution**.
7. Verify that a dll named, `Symyx.workflow.MyExampleActivity.dll`, was saved in the `C:\Program Files\Symyx\Symyx workflow Designer <version_number>\Designer` folder.

Custom Activity OnExecute Method

The `OnExecute` method in the code contains the custom activity functionality. The `OnExecute` method is called when the activity is run by the Vault Workflow service. The `MyExampleActivity` class also contains a call to the `GetLogger` method that writes messages to the log

The `OnExecute` method sets the description of the Vault object currently processed by the Workflow service to set by `MyExampleActivity`, for example, after processing a document object, the document's description is updated.

```

namespace Symyx.SDK.Workflow
{
    /// <summary>
    /// Example activity
    /// </summary>
    [ToolboxItem(typeof(ActivityToolboxItem))]
    [Description("MyExampleActivity")]
    public class MyExampleActivity: ActivityBase
    {
        #region privates
        private static readonly log4net.ILog log = log4net.LogManager.GetLogger
(System.Reflection.MethodBase.GetCurrentMethod(). DeclaringType);
        #endregion privates
        /// <summary>
        /// Initializes a new instance of <see cref="MyExampleActivity"/>
        /// </summary>
        public MyExampleActivity()
        {
            Name = "MyExampleActivity";
        }

        /// <summary>
        /// The OnExecute method is run when the custom activity is run
        /// by the vault workflow service.
        /// </summary>
        /// <param name="context">The context for the activity
        /// execution.</param>
        protected override void OnExecute(ActivityExecutionContext context)
        {
            try
            {
                // get the current vault object being processed by the workflow
                // service from the workspace object
                VaultObject exampleObject = workspace.Get(new VaultId
(ActivityObjectId), DataScope.Properties);
                if (exampleObject != null)
                {
                    // update the vault object description and save it to vault
                    exampleObject.Description = "set by MyExampleActivity";
                    workspace.Update(exampleObject, SaveBehavior.PropertiesOnly);
                }
            }
            catch (Exception ex)
            {
                if (log.IsErrorEnabled)
                {
                    log.ErrorFormat(string.Format("MyExampleActivity failed to processed
OnExecute for object ID {0}",ActivityObjectId), ex);
                }
            }
        }
    }
}

```

The following line in the previous code saves the updated object to Vault:

```
workspace.Update(exampleObject, SaveBehavior.PropertiesOnly);
```

Configure Workflow Designer to Use a Custom Activity

1. Using a text editor.
2. Open the `Symyx.Workflow.Designer.exe.config` file located in Workflow Designer installation folder.
3. Add `Symyx.SDK.Workflow.MyExampleActivity.dll` to the `ReferencedAssemblies` section of the `Symyx.Workflow.Designer.exe.config` file:

```
<setting name="ReferencedAssemblies" serializeAs="Xml">
  <value>
    <ArrayOfString xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <string>Symyx.Workflow.BasicEvents.dll</string>
      ...
      <string>Symyx.SDK.Workflow.MyExampleActivity.dll</string>
    </ArrayOfString>
  </value>
</setting>
```

4. Add the name of the DLL to the `ToolboxItems` section of the `Symyx.Workflow.Designer.exe.config` file:

```
<setting name="ToolboxItems" serializeAs="Xml">
  <value>
    <ArrayOfString xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <string>System.Workflow.Activities.CallExternalMethodActivity,
System.Workflow.Activities</string>
      ...
      <string>Symyx.SDK.Workflow.MyExampleActivity,
Symyx.SDK.Workflow.MyExampleActivity</string>
    </ArrayOfString>
  </value>
</setting>
```

The first `Symyx.SDK.Workflow.MyExampleActivity` contains the `Symyx.SDK.Workflow` namespace plus the `MyExampleActivity` class name. The second `Symyx.SDK.Workflow.MyExampleActivity` contains the assembly DLL name.

5. Save and close the file.
6. Start Workflow Designer.
7. Verify that `MyExampleActivity` is displayed in the toolbox.

Add a Custom Activity to a Workflow

Note: If a custom workflow activity name does not end with the word *Activity*, an error displays when you open the workflow in the Workflow Designer.

To add a custom activity:

1. In Workflow Designer, open an existing workflow.
2. Click **Yes** in response to the error to load the activity.
If you are opening a workflow that contains a custom activity, an error displays stating that the activity is not loaded.
3. Double-click an activity.
4. Drag and drop **MyExampleActivity** from the toolbox to a position inside the activity.
5. In Properties, click the ... button next to **ActivityObjectId**.
6. Select **ObjectID** and click **OK** to bind ActivityObjectId.
The ID represents the Vault object that is processed by the Workflow service when the workflow is run. For example, if the workflow processes a document, then the custom activity updates the description of that document.
7. Save the workflow with a different name from the original workflow file.

Custom Workflow Activity Example

The example shows how to create and execute a custom workflow activity named the Print Report activity. The example works as follows:

When a document is created in Notebook Explorer, a workflow containing the Print Report activity is run. The Print Report activity sends a message to a Microsoft MSMQ queue.

The MSMQ message is read by a program named the Print Report Console. The Print Report Console then runs a program named the Report Printer.

The Report Printer generates a PDF containing the document details, using a layout specified in a report template.

In the Print Report custom workflow activity, the dependency properties do the following:

- Record the Microsoft MSMQ queue name where the request to print the report is sent.
- Record the report template ID.

The report template contains the layout for the PDF report. You can also use a report definition ID.

The Report queue name property is dependency property with two parts:

- An object of the class DependencyProperty.
- A string with get and set methods for the DependencyProperty object.

The following example defines a DependencyProperty public static object named QueueNameProperty.

```
public static DependencyProperty QueueNameProperty =
DependencyProperty.Register( "QueueName", typeof(string), typeof
(PrintReportActivity), new PropertyMetadata(@".\Private$\PrintReports"));
```

The following example defines a string named QueueName that is exposed to Workflow Designer. The object contains get and set methods for QueueNameProperty, for example:

```
[Description("Report queue name")] [Category("Dependency Properties")]
[Browsable(true)]
[DesignerSerializationVisibility(DesignerSerializationVisibility.Visible)]
public string QueueName
{
    get { return (string)GetValue(QueueNameProperty); }
    set { SetValue(QueueNameProperty, value); }
}
```

Compile and Publish a Workflow

1. Choose **Workflow > Publish Workflow**.
2. Click **Compile**.
3. Click **Publish**.
4. Log in to BIOVIA Vault Server, and click **OK**.
5. Enter a title and description for your workflow.
6. Click **OK**.

Note: You cannot re-publish a workflow that uses the same name as another workflow in the Vault database. You must copy your existing workflow files, rename them, and then compile and publish the new workflow with the new name. You must also create a new workflow association for the new workflow.

Change the Vault Logging Level

Vault provides logging features that enable you to debug workflows.

To add logging for the Symyx.SDK namespace:

1. Using a text editor, open the `Symyx.workflow.service.exe.config` configuration file located in the folder in which the workflow service is installed
2. Go to the `log4net` section in the configuration file.
3. Locate the following section within the `log4net` section:

```
<logger name="Symyx.workflow">
  <level value="ERROR"/>
  <appender-ref ref="Symyx.workflow.FileLogger.Debug"/>
</logger>
```

4. Add the following appender and logger sections immediately after the previous logger section, and set the log level you want, for example, `DEBUG`:

```
<appender name="Symyx.SDK.FileLogger.Debug"
type="log4net.Appender.RollingFileAppender, log4net, Version=1.2.10.0,
Culture=neutral, PublicKeyToken=1b44e1d426115821">
  <param name="File" value="C:\VaultLogs\Symyx.SDK.log"/>
  <param name="AppendToFile" value="true"/>
  <param name="MaxSizeRollBackups" value="10"/>
  <param name="MaximumFileSize" value="500000"/>
  <param name="RollingStyle" value="Size"/>
  <param name="StaticLogFileName" value="true"/>
  <layout type="log4net.Layout.PatternLayout">
    <param name="ConversionPattern" value="%-5p %d [%t] %c [%x] -
%m%n"/>
  </layout>
</appender>
<logger name="Symyx.SDK">
  <level value="DEBUG"/>
  <appender-ref ref="Symyx.SDK.FileLogger.Debug"/>
</logger>
```

You can also enable debug logging for the `Symyx.workflow` namespace by setting the level to `DEBUG`:

```
<logger name="Symyx.workflow">  
  <level value="DEBUG"/>  
  <appender-ref ref="Symyx.workflow.FileLogger.Debug"/>  
</logger>
```

- Enter the following commands in a command window:

```
net stop "Vault workflow Service"  
net stop " Vault Message Processing Service"  
net start "Vault Message Processing Service"  
net start "Vault workflow Service"
```

The SDK log file is located in the C:\VaultLogs\Symyx.SDK.log, that contains log messages after you run a workflow containing the Print Report activity or any other class in the Symyx.SDK namespace.

The file is located in the Symyx Workflow file at C:\VaultLogs\Symyxworkflow.log.

Appendix A:

Potentially Breaking API Changes in Workbook 2018

IMPORTANT! If you are upgrading from a **pre-2018** release of Workbook, it is important to retest your customizations to ensure that they still work correctly.

Workbook 2018 contained some potentially breaking API changes related to DevExpress APIs. DevExpress is a third-party product used in Workbook to handle certain UI elements. To support the .NET 4 runtime environment introduced in Workbook 2018, it was necessary to upgrade from DevExpress 8.2.6 to DevExpress 16.1.8.

As a consequence of this upgrade, BIOVIA found the following:

- Objects previously compiled using DevExpress 8.2.6 must be recompiled using DevExpress 16.1.8.
- Some code that uses DevExpress objects without directly referencing the DevExpress DLLs still works, but some does not, depending on how the DevExpress objects are used. Some differences in behavior are obvious, but others are more subtle.

These issues might necessitate minor modifications, recompilation, and redeployment of your customizations.

This appendix provides some observations that might help you with this effort.

Cannot Populate Container Until DevExpress Constructor Completes

The DevExpress constructor, which triggers the "Initialize" or "OnInitialize" methods, must now be completed **before** the associated container can be populated with data. The 'OnInitialize' methods are not explicitly called, but they are attached to the control's constructor.

Example: Pre-2018

The following example code that worked in previous releases will no longer work with Workbook 2018 or later:

```
public class PagewithControls : PropertyPage
{
    private DevExpress.XtraGrid.GridControl _gridControl1;

    // This constructor is called when the page is requested. It builds the
    // page, and calls helpers to build each component as needed.
    public PagewithControls()
    {
        Title = "Bad Page";
        Control = new CustomizedGridControl();
    }
}

public class CustomizedGridControl : DevExpress.XtraGrid.GridControl
{
    // This constructor is called when a CustomizedGridControl is added to
    // the page currently being built.
    public CustomizedGridControl()
    {
        _gridControl1 = new DevExpress.XtraGrid.GridControl()
```



```

        {
            // CSS, size, margins, titles, event handlers, etc.
        };
        _gridControl1.DataSource = new List<string>() { "Value1", "Value2",
"Value3" }; // this line is the problem, as it's assigning data inside the
constructor
    }
}

```

Example: Updated for 2018 - Approach 1

In the following example, the code has been updated by assigning the data after the constructor is complete:

```

public class PageWithControls : PropertyPage
{
    private DevExpress.XtraGrid.GridControl _gridControl1;

    // This constructor is called when the page is requested. It builds the
    // page, and calls helpers to build each component as needed.
    public PageWithControls()
    {
        Title = "Good Page";
        Control = new CustomizedGridControl();
        // The control has now been initialized, so we can assign data to it
here
        _gridControl1.DataSource = new List<string>() { "Value1", "Value2",
"Value3" };
    }
}

public class CustomizedGridControl : DevExpress.XtraGrid.GridControl
{
    // This constructor is called when a CustomizedGridControl is added to
    // the page currently being built.
    public CustomizedGridControl()
    {
        _gridControl1 = new DevExpress.XtraGrid.GridControl()
        {
            // CSS, size, margins, titles, event handlers, etc.
        };
    }
}

```

Example: Updated for 2018 - An Alternate Approach

In the following example, the code has been updated by overriding the 'OnInitialize' method to populate the data there:

```

public class PageWithControls : PropertyPage
{
    private DevExpress.XtraGrid.GridControl _gridControl1;

    // This constructor is called when the page is requested. It builds the
    // page, and calls helpers to build each component as needed.
    public PageWithControls()

```

```

{
    Title = "Alternative Good Page";
    Control = new CustomizedGridControl();
}

// This is called when the PageWithControls() constructor is 'done'
protected override void OnInitialize()
{
    base.OnInitialize();
    // Initialization is complete, so we can assign data to the control
now
    _gridControl1.DataSource = new List<string>() { "Value1", "Value2",
"Value3" };
}
}

public class CustomizedGridControl : DevExpress.XtraGrid.GridControl
{
    // This constructor is called when a CustomizedGridControl is added to
the page currently being built.
    public CustomizedGridControl()
    {
        _gridControl1 = new DevExpress.XtraGrid.GridControl()
        {
            // CSS, size, margins, titles, event handlers, etc.
        };
    }
}

```

Namespace Replacements

The following namespaces were officially replaced, but without changing functionality.

Obsolete Namespace	Replacement Namespace
GridView.ShowGridMenu	GridView.PopupMenuShowing
GridView.GridMenuEventArgs	GridView.PopupMenuShowingEventArgs
XtraTreeList.TreeListMenuEventArgs	XtraTreeList.PopupMenuShowingEventArgs

Changes to Menu Item List Creation

Items that were previously created with an initial 'hidden' attribute are no longer created until they are needed. They are also now removed when no longer needed.

This change could affect scripts that modify contextual menus without first checking whether the menu options they attempt to modify actually exist.

GridView Methods Require Additional Parameters

- `GridView.CalcRowHeight()` now requires a fourth parameter that must be equal to the row's level. It might be necessary to use other methods such as `viewInfo.CalcRowHeight(graphics, rowHandle, 0, sectionGridView.GetRowLevel(rowHandle))` in place of this method, which was intended to be internal.
- `GridView.GetGridCellInfo()` now requires a `RowHandle` and a `Column`, instead of a

RowHandle and a Column Index.

- Some grid-related methods still require a RowHandle as a parameter, even though many events no longer provide row handles. In some cases, you can use ListSourceRowIndex. If you know the RowIndex, you can also use View.GetRowHandle(index).

Deprecated Events and Methods

- RowHandle is deprecated for some events and replaced with ListSourceRowIndex.
- XtraVerticalGrid.SingleRecordViewInfo.GetRowIndentwidth() method no longer exists. Use the SingleRecordViewInfo.RowIndentwidth property instead.

EditorContainer.RepositoryItems Renamed as ExternalRepository

Existing methods and scripts that use this property must be renamed. Other code changes might be required.

RepositoryItemCheckedComboBoxEdit.ShowAllItemCaption Renamed as SelectAllItemCaption

This change is for clarity and does not change functionality.

tree.OptionsBehaviour.DragNodes Renamed and Changed to Boolean

This property has been renamed as tree.OptionsBehaviour.DragNodesMode and changed to an enum from a Boolean. DragNodesMode.None is equivalent to setting DragNodes = false in previous version.

Exceptions Sometimes Thrown when Operating on non-UI Thread

DevExpress grid now throws an exception when certain operations are performed on a non-UI thread. This exception was not thrown in the previous version.

Workbook code has been adjusted where this problem has been found.

Any code that performs grid updates on a background thread will need to be modified to ensure updates occur on the UI thread.

Some Grid Methods Now Clear Status Data

Some Grid methods now will clear status data (like selected rows, focused row, etc.).

Examples include GridView.ClearGrouping() and GridColumn.Group().

If needed, save these values before calling the methods, and reassign using GridView.SelectRow(rowHandle), GridView.FocusedRowHandle = rowHandle, and so on.