TEAMCENTER

Teamcenter Server Customization in C# — Reference

Scripted customization

Teamcenter 2312
December 2023

About Siemens Digital Industries Software

Siemens Digital Industries Software is a leading global provider of product life cycle management (PLM) software and services with 7 million licensed seats and 71,000 customers worldwide. Headquartered in Plano, Texas, Siemens Digital Industries Software works collaboratively with companies to deliver open solutions that help them turn more ideas into successful products. For more information on Siemens Digital Industries Software products and services, visit www.siemens.com/plm.

Support Center: support.sw.siemens.com
Send Feedback on Documentation: support.sw.siemens.com/doc_feedback_form

SIEMENS

Teamcenter Server C# Customizers' Guide

This guide shows you how to write Tc Server customizations in C# on a local Windows development machine.

It is aimed at being a practical guide, focusing on worked examples that provide a comprehensive overview of the customization API, taking you incrementally through the task of writing customizations in C#. It is recommended to read the guide through at least once, since what comes later often builds on what has come before.

Apart from a reasonable grasp C#, the guide assumes some familiarity with the basic concepts and features of codeful Teamcenter server customizations as currently presented by BMIDE: what they are, and what they do. However, you don't need to know how to use BMIDE or how write C++ customized code in order to find the guide useful.

For more detail, you are also recommended to read the help files that are embedded at the end of this document; they contain the full documentation of the entire set of C# API classes.

Contents

How to set up the development environment

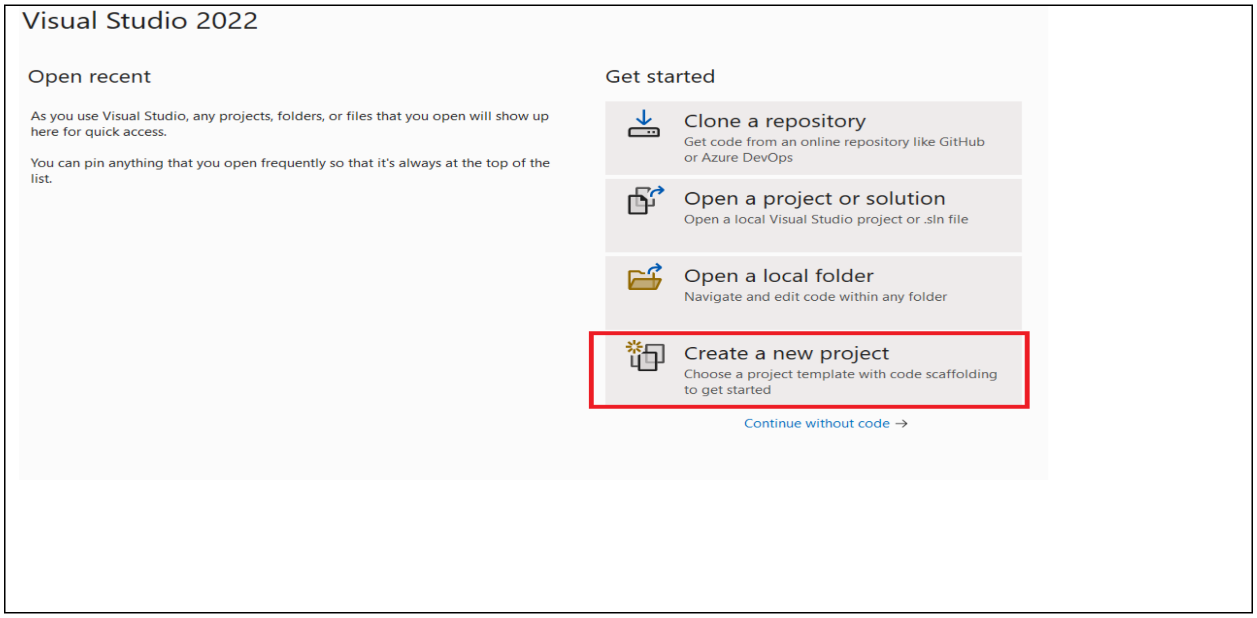
You will need

- .NET runtime at version 6.0 or later
- A C# development environment, such as Visual Studio or Visual Studio Code. It must support .NET 6.0 or later. For the purpose of this document, I will assume you're using Visual Studio.
- NuGet server setup, *see Setting up NuGet Server*
    - There is also a Directory based setup, see *Directory* based setup
- Install "Scripted Business Logic" component from Teamcenter Deployment Center (DC)
-

In Visual Studio, create a new .NET Core library project with .NET 6.0 framework.

Follow the steps below:

1. Select "Create a new project"



Visual Studio 2022

Open recent

As you use Visual Studio, any projects, folders, or files that you open will show up here for quick access.

You can pin anything that you open frequently so that it's always at the top of the list.

Get started

Clone a repository
Get code from an online repository like GitHub or Azure DevOps

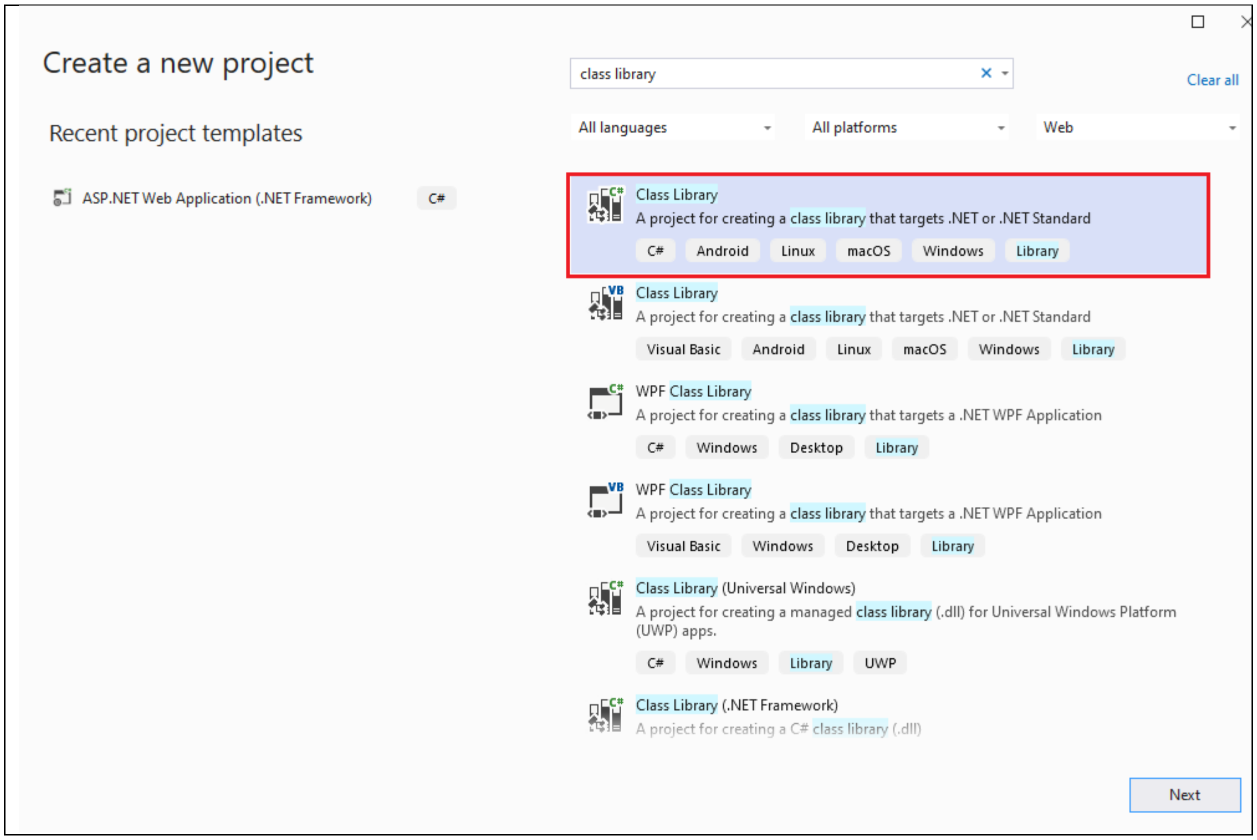
Open a project or solution
Open a local Visual Studio project or .sln file

Open a local folder
Navigate and edit code within any folder

Create a new project
Choose a project template with code scaffolding to get started

Continue without code →

2. Make sure you select C# "Class Library"


Create a new project

class library

Clear all

Recent project templates

All languages | All platforms | Web

ASP.NET Web Application (.NET Framework) C#

Class Library
A project for creating a class library that targets .NET or .NET Standard
C# Android Linux macOS Windows Library

Class Library
A project for creating a class library that targets .NET or .NET Standard
Visual Basic Android Linux macOS Windows Library

WPF Class Library
A project for creating a class library that targets a .NET WPF Application
C# Windows Desktop Library

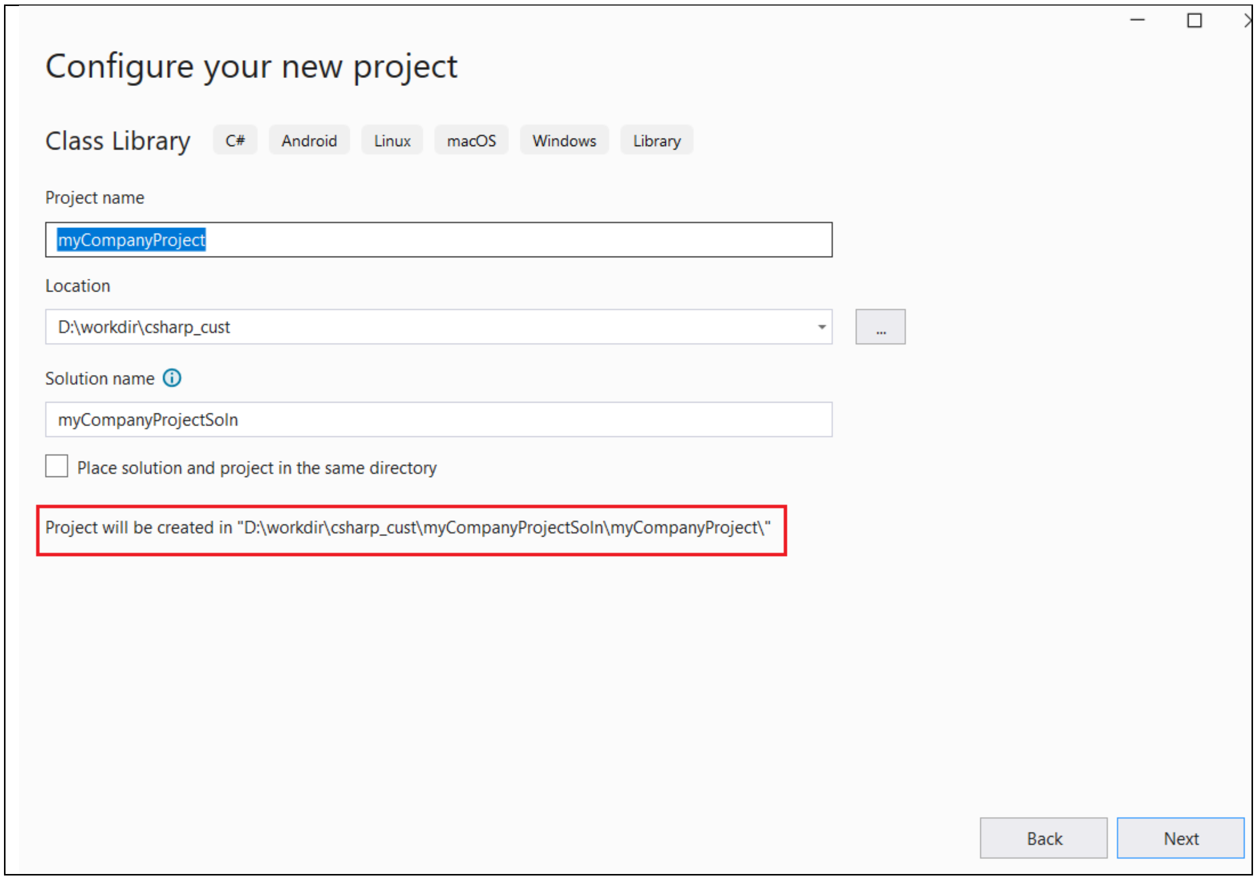
WPF Class Library
A project for creating a class library that targets a .NET WPF Application
Visual Basic Windows Desktop Library

Class Library (Universal Windows)
A project for creating a managed class library (.dll) for Universal Windows Platform (UWP) apps.
C# Windows Library UWP

Class Library (.NET Framework)
A project for creating a C# class library (.dll)

Next

3. Configure new project
   Enter “myCompanyProject” in Project Name. You can put whatever name you prefer.
   Enter “myCompanyProjectSoln” in Solution Name
   Click “Next”


Configure your new project

Class Library C# Android Linux macOS Windows Library

Project name

myCompanyProject

Location

D:\workdir\csharp_cust

Solution name

myCompanyProjectSoln

Place solution and project in the same directory

Project will be created in "D:\workdir\csharp_cust\myCompanyProjectSoln\myCompanyProject\"

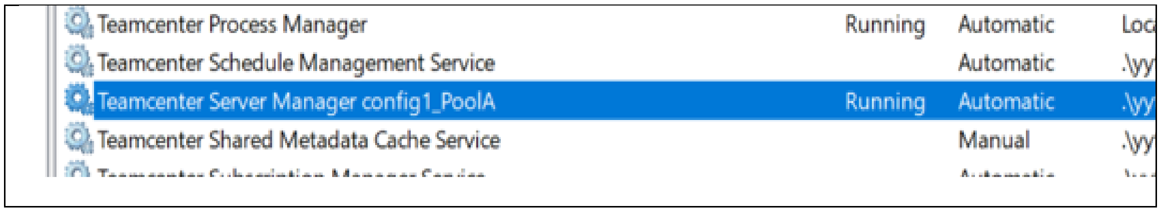
C# customization framework looks for TC_DOTNET_EXTENSION_SOURCE env var during tcserver start-up time so that it can compile the code on the fly later. The value for this variable needs to be set to the project location. See *Rapid code-and-test.*

TC_DOTNET_EXTENSION_SOURCE=C:\workdir\charp_cust\myCompanyProjectSoln\myCompanyProject

a. You MUST update the following variable in the %TC_DATA%/tc_profilevars.bat file, i.e

set TC_DOTNET_EXTENSION_SOURCE=<*path to directory containing C# customization code (i.e. *.csproj file location)*>[2]

b. Restart the pool-manager from Windows->Services dialog



Teamcenter Process Manager	Running	Automatic	Loc
Teamcenter Schedule Management Service		Automatic	.\yy
Teamcenter Server Manager config1_PoolA	Running	Automatic	.\yy
Teamcenter Shared Metadata Cache Service		Manual	.\yy

[2] The hot-deployed code must all be in this folder, or in its subfolders. If you have a .csproj file, it must be in this folder.

Also, you need to set the following env vars in %TC_DATA%/tc_profilevars.bat file so that customization asseblies are loaded from right location.

- TC_DOTNET_LIBRARY_PATH=%TC_BIN%/csharpcust

If you have previous C# customization NuGet package and you want to deploy that in tcserver, you need to include the path.

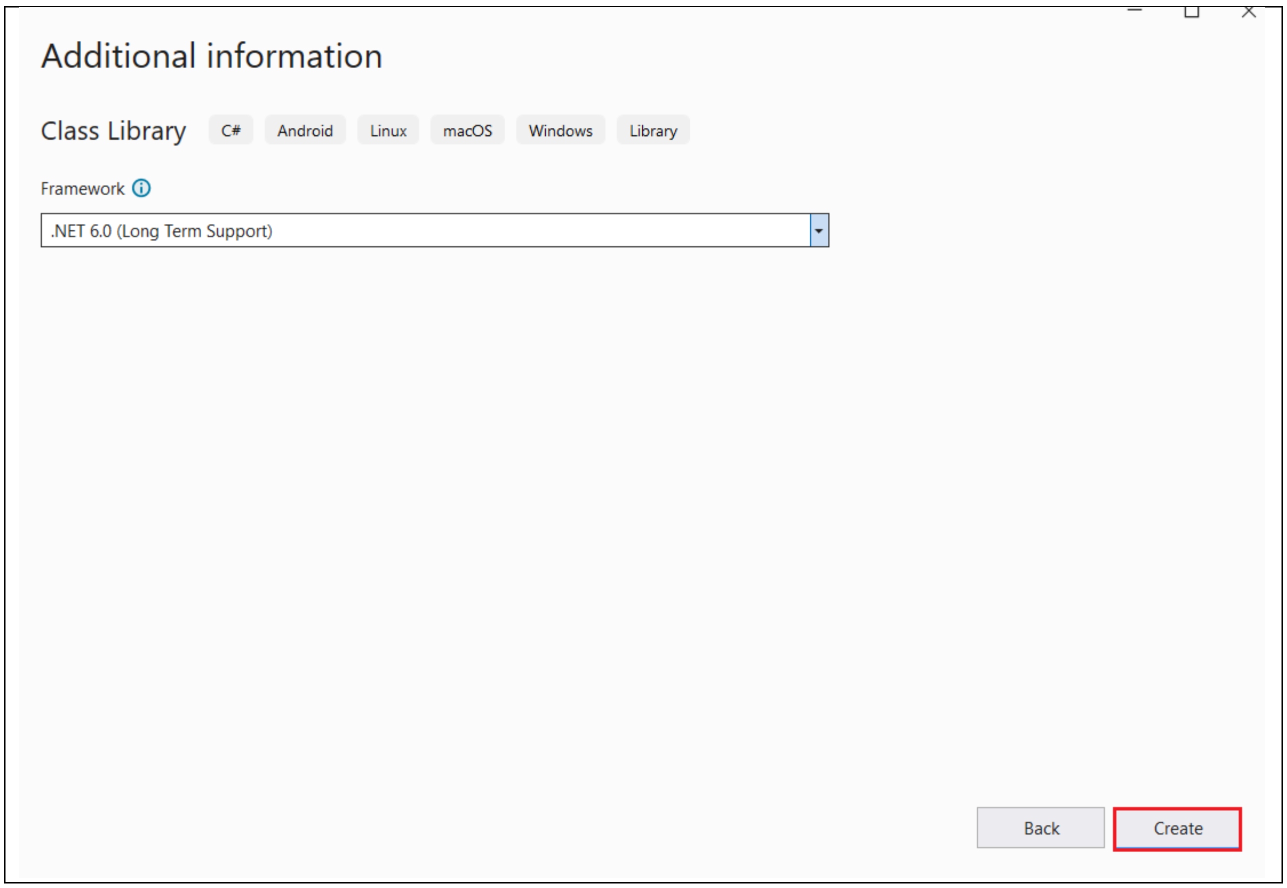
- TC_NUGET_FEEDS=<Path>

If you have NuGet server running, you need to include the NuGet server service URL as well.

See *Setting up NuGet Server*

Teamcenter Deployment Center (DC) supports deployment of this features as "Scripted Business Logic" which is an optional component in DC as deployment of this feature is not required to run Teamcenter server. See DC documentation for details.
During deployment %TC_DATA%/tc_profilevars.bat will be modified to include the above variables (TC_DOTNET_EXTENSION_SOURCE, TC_DOTNET_LIBRARY_PATH and TC_NUGET_FEEDS). You might need to change the value(s) according to this document.

4. Then press "Create" to create the project


Additional information

Class Library

Framework

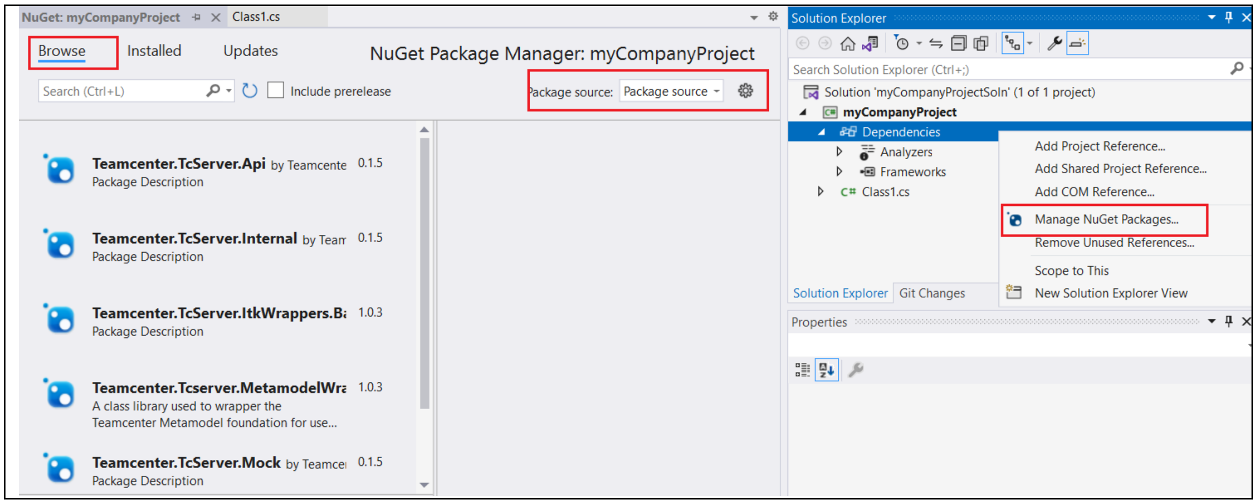
.NET 6.0 (Long Term Support)

5. Set up the NuGet package sources. Teamcenter Deployment center (DC) requires that NuGet server was set up and running in customer machine. Teamcenter.TcServer.ItkWrappers.Base and Teamcenter.TcServer.MetamodelWrappers.foundation are already pushed to NuGet server, so you can ignore this section. But later if you want to use directory based set up, you can use any of the 2 directory based set up below. Nuget package source may be any of the following:

   - Local directory
   - Shared directory
   - NuGet service (see *Setting up NuGet Server*).

   In this setup "Local directory" is used.

   Right click on the "Dependencies" and choose "Manage NuGet Packages.."
   Select "Browse"



Teamcenter.TcServer.Api by Teamcente 0.1.5
Package Description

Teamcenter.TcServer.Internal by Tean 0.1.5
Package Description

Teamcenter.TcServer.ItkWrappers.B: 1.0.3
Package Description

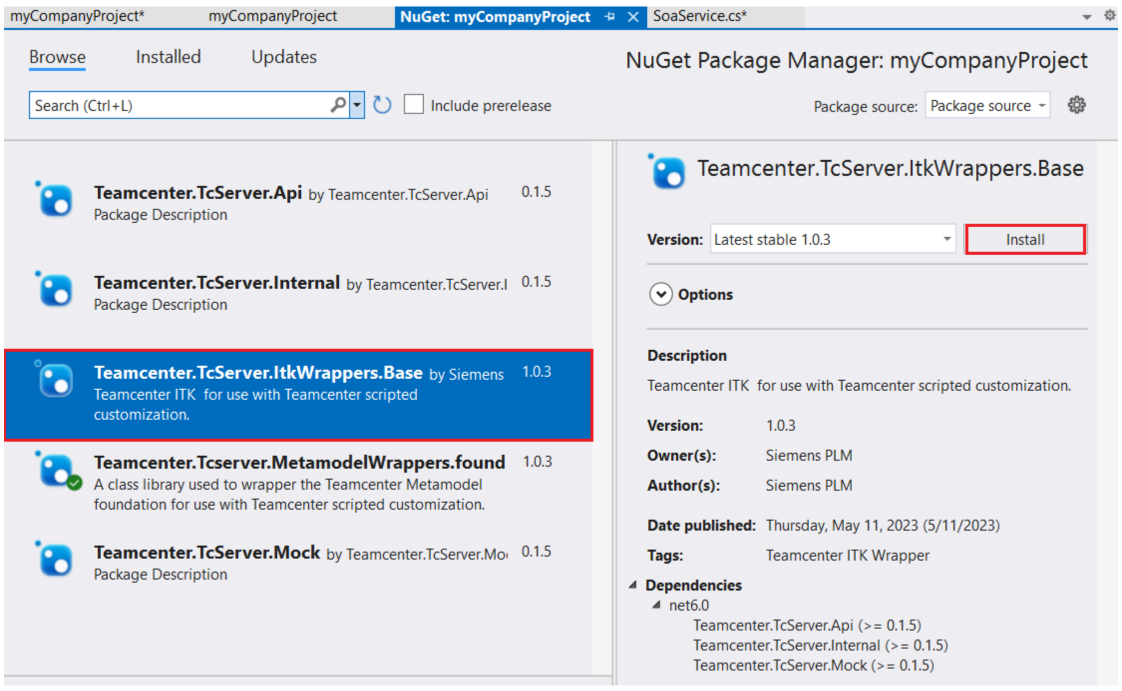
Teamcenter.Tcserver.MetamodelWr: 1.0.3
A class library used to wrapper the Teamcenter Metamodel foundation for use...

Teamcenter.TcServer.Mock by Teamce 0.1.5
Package Description

If you don't see any Teamcenter packages, press "wheel" button in the above image and add the path of the "Package Source".

For Teamcenter installed env the path value would be %TC_BIN%/csharpcust

- Add a dependency to the NuGet package Teamcenter.TcServer.ItkWrappers
  Select Teamcenter.TcServer.ItkWrappers



Teamcenter.TcServer.Api by Teamcenter.TcServer.Api 0.1.5
Package Description

Teamcenter.TcServer.Internal by Teamcenter.TcServer.I 0.1.5
Package Description

Teamcenter.TcServer.ItkWrappers.Base by Siemens 1.0.3
Teamcenter ITK for use with Teamcenter scripted customization.

Teamcenter.Tcserver.MetamodelWrappers.found 1.0.3
A class library used to wrapper the Teamcenter Metamodel foundation for use with Teamcenter scripted customization.

Teamcenter.TcServer.Mock by Teamcenter.TcServer.Mo 0.1.5
Package Description

Teamcenter.TcServer.ItkWrappers.Base

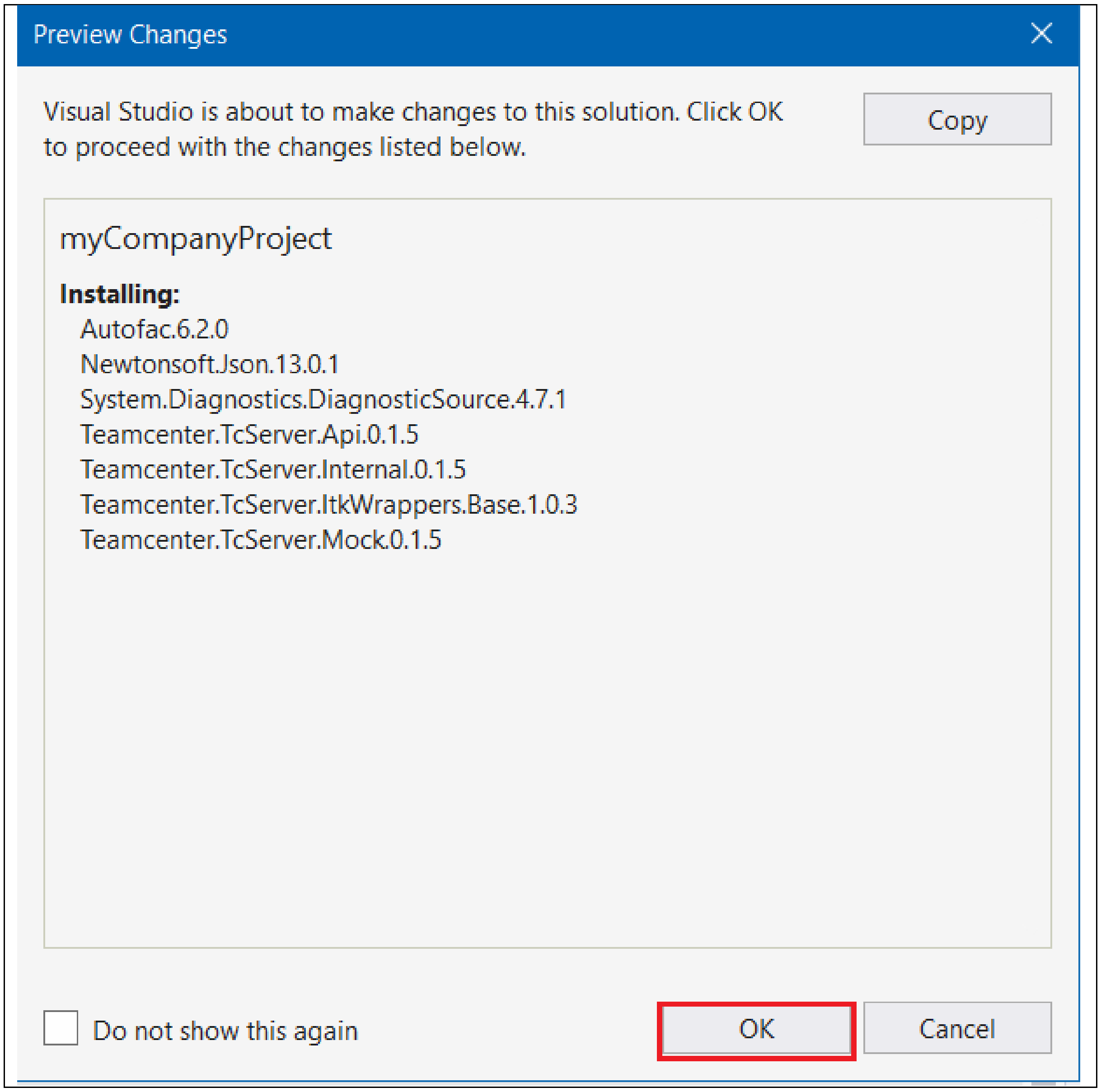
Version: Latest stable 1.0.3

Description
Teamcenter ITK for use with Teamcenter scripted customization.

Version:	1.0.3
Owner(s):	Siemens PLM
Author(s):	Siemens PLM
Date published:	Thursday, May 11, 2023 (5/11/2023)
Tags:	Teamcenter ITK Wrapper

Dependencies
- net6.0
  - Teamcenter.TcServer.Api (>= 0.1.5)
  - Teamcenter.TcServer.Internal (>= 0.1.5)
  - Teamcenter.TcServer.Mock (>= 0.1.5)

Click “Install” and Click “OK”



Visual Studio is about to make changes to this solution. Click OK to proceed with the changes listed below.

myCompanyProject

Installing:

Autofac.6.2.0
Newtonsoft.Json.13.0.1
System.Diagnostics.DiagnosticSource.4.7.1
Teamcenter.TcServer.Api.0.1.5
Teamcenter.TcServer.Internal.0.1.5
Teamcenter.TcServer.ItkWrappers.Base.1.0.3
Teamcenter.TcServer.Mock.0.1.5

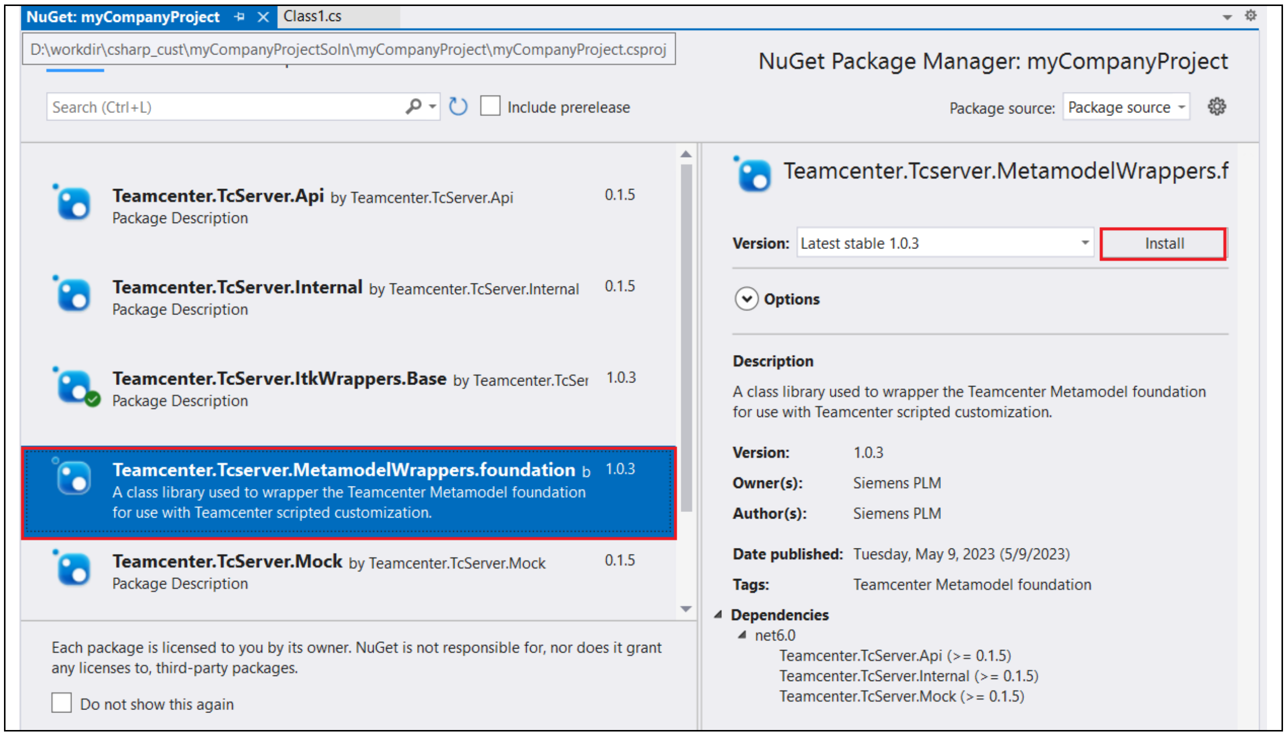
Do not show this again

- You will also need a source of Teamcenter metadata. These are deployed in packages generated from Teamcenter templates. In this example, we'll use only the Foundation package (generated from foundation template), but you may need packages for other Teamcenter templates (see *How to run TemplateGen.exe to generate metamodel wrapper for templates*) and for your custom templates as well. For now, just add the Foundation package

  Similarly Add a dependency to the NuGet package Teamcenter.TcServer.MetamodelWrappers.foundation

Select Teamcenter.TcServer.MetamodelWrappers.foundation

Click “Install”



NuGet: myCompanyProject | Class1.cs

D:\workdir\csharp_cust\myCompanyProjectSoln\myCompanyProject\myCompanyProject.csproj

NuGet Package Manager: myCompanyProject

Search (Ctrl+L) | Include prerelease | Package source: Package source

Teamcenter.TcServer.Api by Teamcenter.TcServer.Api 0.1.5
Package Description

Teamcenter.TcServer.Internal by Teamcenter.TcServer.Internal 0.1.5
Package Description

Teamcenter.TcServer.ItkWrappers.Base by Teamcenter.TcSer 1.0.3
Package Description

Teamcenter.Tcserver.MetamodelWrappers.foundation b 1.0.3
A class library used to wrapper the Teamcenter Metamodel foundation for use with Teamcenter scripted customization.

Teamcenter.TcServer.Mock by Teamcenter.TcServer.Mock 0.1.5
Package Description

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

Do not show this again

Teamcenter.Tcserver.MetamodelWrappers.f

Version: Latest stable 1.0.3 | Install

Options

Description
A class library used to wrapper the Teamcenter Metamodel foundation for use with Teamcenter scripted customization.

Version:	1.0.3
Owner(s):	Siemens PLM
Author(s):	Siemens PLM
Date published:	Tuesday, May 9, 2023 (5/9/2023)
Tags:	Teamcenter Metamodel foundation

Dependencies
- net6.0
  - Teamcenter.TcServer.Api (>= 0.1.5)
  - Teamcenter.TcServer.Internal (>= 0.1.5)
  - Teamcenter.TcServer.Mock (>= 0.1.5)

That’s it; you’re ready to go.

<u>Note</u>: you can use JetBrains’ “dotPeek” utility to find out the interfaces exposed in the above NuGet packages to use in custom code.

If you’d like to run the examples as you go, look ahead for information about *<u>Rapid code-and-test</u>* and then come back here.

How to create a custom SOA service

It's possible to write SOA services and their operations in C#.

Create the service class

Create a new C# class and give it a [SoaService] attribute

```
using System.Xml.Linq;
using Teamcenter.TcServer.InternalItk;
using Teamcenter.TcServer.Api.Metamodel;
using Teamcenter.TcServer.MetamodelWrappers.foundation;
using Teamcenter.TcServer.Api.PropertyReader;
using Teamcenter.TcServer.Api.PomEnq;
using Teamcenter.TcServer.Api.Soa;

namespace MyCustomizations
{
    [SoaService(Name = "example.service", Namespace = "example.service.2022")]
    public class MySoaService
    {
    }
}
```

The Name and Namespace parameters are optional, but you **should not keep them empty** to avoid name collision, see *Namespace Consideration* for details. If you leave them out they both default to the class name, which is not recommended. The name is the service name that is used in SOA requests for any of the service's operations.

You may write as many [SoaService] classes as you want, but they must each define a different service.

Write a C# method that implements a service operation

The method can take any number of parameters and must return some kind of object; the simplest method you can write has no arguments and returns a simple object. It is decorated by [SoaServiceMethod] attribute.

```
[SoaServiceMethod(Name = "op", ResponseTypeName = "result")]
public object MySoaOperation()
{
    return new { };
}
```

Believe it or not, this is a valid SOA service operation. It returns the SOA response

```
{
".QName":"example.service.2022.result"
}
```

The Name and ResponseTypeName are optional, the Name defaults to the method name. The ResponseTypeName is appended to the service's Namespace to form the SOA response's .QName field.

Define the operation’s response type

You set the SOA response simply by returning a suitable C# value – any value[3]. The value is serialized to json to form the SOA response[4].

All the following are valid return statements

```
        return 1;
        return true;
        return "hello, world";
        return new Dictionary<string, int>{ { "hi", 5}, { "ho", 6 } };
        return new MyStruct{ x = 1 };
```

We will look later at how to use an IServiceDataFactory to add Teamcenter SOA Service Data to the response.

If you miss out the optional ResponseTypeName from the SoaServiceMethod attribute, it defaults to the *method’s* return type. This is not necessarily the same as the type of the returned value. For example, compare the SOA responses from these two SOA methods that differ only in their declared return type:

Method

```
[SoaServiceMethod(Name = "op1")]
public string MySoaOperation1()
{
    return "hello, world";
}
[SoaServiceMethod(Name = "op2")]
public object MySoaOperation1()
{
    return "hello, world";
}
```

Response

```
{
".QName":"example.service.2022.string",
"result":"hello, world"
}

{
".QName":"example.service.2022.object",
"result":"hello, world"
}
```

Define the operation’s input parameters

You define the input parameters by adding C# parameters to the C# method. Each field in the *body* of the SOA request is matched by type and (case-insensitive) name to the method’s parameters. You can use any public concrete C# type as a parameter type, as long as it can be deserialized from json string[5]; just don’t use interfaces, abstract classes or generic templates.

Here are some valid C# SOA method declarations, alongside some examples of json strings for the SOA request’s “body” field that can be used to call the SOA operation

Method

```
[SoaServiceMethod(Name = "op"]
public object MySoaOperation()
```

Json “body” field in the SOA request

```
"body":{}
```

[3] Almost; it currently doesn’t handle char, DateTime or TcDate values (but you can return these as strings). Also, a limitation in the current C++ SOA layer means can handle collections of primitive types, but not collections of objects or structs.
[4] If you return anything that isn’t an object or a struct (e.g. a primitive or a collection), it is serialized as a json object with a field, “result” set to the returned value.
[5] Again, except char, DateTimes and TcDates. The type must have at least one a public constructor.

```
    = > "ok";
[SoaServiceMethod(Name = "op"]                    "body":{"a":1}
public object MySoaOperation(int a)
    = > a;
[SoaServiceMethod(Name = "op"]                    "body":{"a":1}
public object MySoaOperation(double a)            "body":{"a":1.2}
    = > a;
[SoaServiceMethod(Name = "op")]                   "body":{"a":[]}
public object MySoaOperation(List<string> a)      "body":{"a":["hello", "world"]}
    => a;
[SoaServiceMethod(Name = "op")]                   "body":{"a":{"x":1}}
public object MySoaOperation(MyStruct a)
    => a;

public struct MyStruct { public int x; }
[SoaServiceMethod(Name = "op")]                   "body":{"a":{"input":1}}
public object MySoaOperation(MyClass a)
    => a;

public class MyClass
{
    private int _x;
    public MyClass(int input) => _x = input;
}
[SoaServiceMethod(Name = "op")]                   "body":{"inputa":"hello", "inputb":1}
public object MySoaOperation(string inputA,       "body":{"INPUTA":"hello", "INPUTB":1}
int inputB)                                       "body":{"inputB":1, "inputA":"hello"}
    => new {inputA, inputB};
```

You can make any parameter optional by decorating it with the [MayBeNull] attribute. Null parameters may be omitted from the request *body*. You must of course handle the null values in your code.

Here is the complete example code for a SOA operation:

```
using System.Xml.Linq;
using Teamcenter.TcServer.InternalItk;
using Teamcenter.TcServer.Api.Metamodel;
using Teamcenter.TcServer.MetamodelWrappers.foundation;
using Teamcenter.TcServer.Api.PropertyReader;
using Teamcenter.TcServer.Api.PomEnq;
using Teamcenter.TcServer.Api.Soa;

namespace myCompanyProject
{
    [SoaService(Name = "SoaService", Namespace = "siemens.service.2023")]
    public class SoaService
    {
        private readonly IBulkPropertyReaderFactory _propertyReaderFactory;
        private readonly IPomEnq _pomEnquiry;
        private readonly IMetamodel _metamodel;
        private readonly IServiceDataFactory _serviceDataFactory;
        private readonly IRegisteredTcOperationDispatcher _tcOperationDispatcher;
        public SoaService(IPomEnq pomQuery, IBulkPropertyReaderFactory
propertyReaderFactory, IPomEnq pomEnquiry, IRegisteredTcOperationDispatcher
tcOperationDispatcher, IMetamodel metamodel, IServiceDataFactory serviceDataFactory)
        {
            _propertyReaderFactory = propertyReaderFactory;
            _pomEnquiry = pomEnquiry;
            _metamodel = metamodel;
            _tcOperationDispatcher = tcOperationDispatcher;
            _serviceDataFactory = serviceDataFactory;
        }

        [SoaServiceMethod(Name = "HelloSoaOperation", ResponseTypeName = "result")]
        public object Hello()
        {
            return "Hello";
        }
        [SoaServiceMethod(Name = "callCalculateCostSoaOperation", ResponseTypeName =
"result")]
        public object CalculateCost()
        {
            var objectTag =
_pomEnquiry.FromType(Item.QueryRoot).LoadInstancesFromDatabase().Take(1).Single();
            var ints = new[] { 100, 200, 400, 500 };
            var parameters = new object[] { ints, default(int) };
            // "CalculateCostOperation#const,int,*$const,int,*$int,*" is similar to BMIDE
operation signature
            _tcOperationDispatcher.Execute(objectTag,
"CalculateCostOperation#const,int,*$const,int,*$int,*", parameters);
            var result = new
            {
                sum = new { uiValues = new[] { (int)parameters.Last() } }
            };
            return result;
        }
    }
}
```

Defining additional SOA operations

You can add as many additional SOA operations as you need to your C# class. All the operations for the same SOA service must be defined in the same C# class.

See *Basic Types* provided by the C# customization framework.

For additional information about Teamcenter interfaces, see *How to access Teamcenter services from a custom SOA operation*

How to write a custom Workflow Handler

It’s possible to implement both action workflow handlers and rule workflow handlers in C#. Here we show you how.

Both types of handlers need a to be defined in a C# class, so start by creating a new class, and give it a [WorkflowExtension] attribute.

```
using Teamcenter.TcServer.Api;
using Teamcenter.TcServer.Api.Metamodel;
using Teamcenter.TcServer.Api.Workflow;
using Teamcenter.TcServer.InternalItk;
using Teamcenter.TcServer.MetamodelWrappers.foundation;

namespace myCompanyProject
{
    [WorkflowExtension]
    public class MyWorkflowHandlers
    {
    }
}
```

You may write as many C# classes with workflow handlers as you like.

Write an action handler

For a workflow action handler, add a C# method that takes a single parameter of type EpmActionMessage and returns void. Add an [ActionHandler] attribute to the method; the attribute has two parameters: the handler's name (this is what is displayed in the Workflow UI) and its description:

```
[ActionHandler("myhandler", "used in example code")]
public void MyHandler(EpmActionMessage actionMessage)
{
}
```

To implement anything useful, you will need some Teamcenter services; you can acquire these through the C# class's constructor and use them in the workflow handler in the same way as we did with SOA service (see *How to access Teamcenter services from a custom SOA operation*).

In this example, we acquire an IMetamodel object and use it to query the date on which the handler's EPMTask was started, and, if its more than seven days ago, to set its priority.

```
[ActionHandler("myhandler", "used in example code")]
public void MyHandler(EpmActionMessage actionMessage)
{
    if (actionMessage.Task.HasNullValue) return;

    var epmTaskInterface = EpmTask.Interface(_metamodel);
    // find the task's start date
    var startDate = epmTaskInterface.get_fnd0StartDate(new[] { actionMessage.Task
}).First()?.ToDateTime();

    // if it was started more than seven days ago, set its priority to 1
    if (startDate is not null && DateTime.UtcNow - startDate > TimeSpan.FromDays(7))
        epmTaskInterface.set_priority(actionMessage.Task, 1);
}
```

Write a rule handler

Writing a rule handler is very similar: write a method that takes a single parameter of type EpmRuleMessage and returns an EpmRuleDecision enum; the enum is a C# representation of the ITK

EPM_decision_t enum. Decorate the method with a [RuleHandler] attribute that has the rule handler's name and its description.

Here's an example of a rule handler that blocks an approval task once has been running for more than thirty days.

```
[RuleHandler("myrulehandler", "used in example code")]
public EpmDecisionType MyRuleHandler(EpmRuleMessage ruleMessage)
{
    if (ruleMessage.Task.HasNullValue) return EpmDecisionType.Undecided;

    var epmTaskInterface = EpmTask.Interface(_metamodel);
    // find the task's start date
    var startDate = epmTaskInterface.get_fnd0StartDate(new[] { ruleMessage.Task
}).First()?.ToDateTime();

    // if it an approve task that was started more than thirty days ago, block it
    var shouldBeBlocked = startDate is not null
                          && DateTime.UtcNow - startDate > TimeSpan.FromDays(30)
                          && ruleMessage.ProposedAction == EpmAction.Approve;

    return shouldBeBlocked ? EpmDecisionType.NoGo : EpmDecisionType.Go;
}
```

<u>Complete example code to add an actionhandler and rulehandler:</u>

```
using Teamcenter.TcServer.Api;
using Teamcenter.TcServer.Api.Metamodel;
using Teamcenter.TcServer.Api.Workflow;
using Teamcenter.TcServer.InternalItk;
using Teamcenter.TcServer.MetamodelWrappers.foundation;

namespace myCompanyProject
{
    [WorkflowExtension]
    public class Workflow
    {
        private readonly IEPMTask _epmTaskInterface;
        //private readonly IAomProp _aomProp;
        private readonly ISysLog _syslog;

        public Workflow(IMetamodel metamodel, ISysLog syslog)
        {
            _epmTaskInterface = EPMTask.Interface(metamodel);
            _syslog = syslog;
            _syslog.Log("In Workflow constructor");

        }

        [ActionHandler("myActionhandler", "used in example code")]
        public void MyHandler(EpmActionMessage actionMessage)
        {
            _syslog.Log("In myActionhandler");
            if (actionMessage.Task.HasNullValue) return;
            _epmTaskInterface.set_object_desc(actionMessage.Task, "C# Handler Task
Desc");
        }

        [RuleHandler("myRulehandler", "used in example code")]
        public EpmDecisionType MyRuleHandler(EpmRuleMessage ruleMessage)
        {
            _syslog.Log("In myRulehandler");
            if (ruleMessage.Task.HasNullValue) return EpmDecisionType.NoGo;

            var objectDesc = _epmTaskInterface.get_object_desc(new[] { ruleMessage.Task
}).First();
            _syslog.Log($"objectDesc in myRulehandler: {objectDesc}.");

            var shouldSucceed = objectDesc == "C# Handler Task Desc";

            _syslog.Log($"shouldSucceed in myRulehandler: {shouldSucceed}.");

            return shouldSucceed ? EpmDecisionType.Go : EpmDecisionType.NoGo;
        }
    }
}
```

How to customize a Metamodel type

It's possible to add any kind of codeful customization to a Metamodel type using C#. You can add:

- Runtime properties
- Property getter and setter preconditions, preactions and postactions
- Metamodel operations
- Operation preconditions, preactions and postactions

Whichever you're going to write, you start by creating a new C# class with a [MetamodelTypeExtension] attribute; pass the attribute the name of the Metamodel type you want to modify.

```
using Teamcenter.TcServer.Metamodel;

namespace MyCustomizations
{
    [MetamodelTypeExtension("Item")]
    public class MyItemExtensions
    {
    }
}
```

You may write as many [MetamodelTypeExtensions] classes as you like, and you can have more than once class that extend the same Metamodel type.

How to add a custom runtime string property

To add a runtime property, you add a C# method to the class and decorate it with a [RuntimeProperty] attribute. The method has two parameters: the object tags whose property values are being requested, and an out parameter that holds the value of the properties for each of the tags.

You should use unique prefix (similar to BMIDE) to avoid any name collision with other template(s).

Here's an example:

```
[RuntimeProperty("test_property", RecalculateValue.OnEachRequest)]
public void CalculateTestPropertyValue(IEnumerable<Tag> objects, out
IReadOnlyCollection<(Tag, string)> values)
{
    values = objects.Select(tag => (tag, "hello, world")).ToArray();
}
```

That's all you have to do.

As an aside, if you want to see the property in a SOA response, you can request it to be added to the SOA service data, by specifying it when adding the object tags, like this:

```
[SoaServiceMethod(Name = "op")]
public object MySoaOperation()
{
    var tags = _pomQuery.FromType(Item.QueryRoot)
        .Where(query => query.root.item_id.Length() == 6)
        .LoadInstancesFromDatabase();
    // return test_property values in the service data
    var serviceData = _serviceDataFactory.Create().AddResponse(tags, "test_property");
    return new { serviceData };
}
```

A Metamodel extension class can acquire Teamcenter services in the same way as SOA service class does (see *How to access Teamcenter services from a custom* SOA operation). Once acquired, you can use them to do more interesting things in your extensions. For example, here is a runtime property that uses an IBulkPropertyReader to return values that are the concatenation of the objects' object_name and object_desc:

```
[MetamodelTypeExtension("Item")]
public class MyItemExtensions
{
    private readonly IBulkPropertyReader<ItemRuntimePropertySource> _propertyReader;

    public MyItemExtensions(IBulkPropertyReaderFactory propertyReaderFactory)
    {
        _propertyReader = propertyReaderFactory.For(Item.Properties);
    }

    [RuntimeProperty("test_property", RecalculateValue.OnEachRequest)]
    public void CalculateTestPropertyValue(IEnumerable<Tag> objects, out
IReadOnlyCollection<(Tag, string)> values)
    {
        values =_propertyReader
                    .AddItem1(item => item.object_name)
                    .AddItem2(item => item.object_desc)
                    .Read(objects)
                    .Select(item => (tag: item.Tag, name: item.Item1, desc: item.Item2))
                    .Where(item => item.desc is not null)
                    .Select(item => (item.tag, $"{item.name}_{item.desc}"))
                    .ToArray();
    }
}
```

Complete example code to add a custom runtime property:

Create a new .cs file (let's say RuntimeProperty.cs) in your project and add the following.

```
using System.Diagnostics;
using Teamcenter.TcServer.MetamodelWrappers.foundation;
using Teamcenter.TcServer.Api;
using Teamcenter.TcServer.Api.Metamodel;
using Teamcenter.TcServer.Api.PropertyReader;


namespace myCompanyProject
{
    [MetamodelTypeExtension("ItemRevision")]
    public class RuntimeProperty
    {
        private readonly IBulkPropertyReader<ItemRevisionRuntimePropertySource>
_propertyReader;
        private readonly IMetamodel _metamodel;
        private readonly ISysLog _sysLog;

        public RuntimeProperty(IBulkPropertyReaderFactory propertyReaderFactory,
IMetamodel metamodel, ISysLog syslog)
        {
            _propertyReader = propertyReaderFactory.For(ItemRevision.Properties);
            _metamodel = metamodel;
            _sysLog = syslog;
        }

        [RuntimeProperty("MyRuntimeProperty", RecalculateValue.OnEachRequest)]
        public void RuntimePropBase(IEnumerable<Tag> objectTags, out
IReadOnlyCollection<(Tag, string)> values)
        {
            Console.WriteLine($"Hello from {CurrentMethod.Name()}");
            var strings = "a lazy brown fox jumped over the quick dog".Split(new[] { ' '
});
            values = objectTags.Select((tag, index) => (tag, value: strings[index %
strings.Length])).ToArray();
        }
        [PropertyGetterPostAction("MyRuntimeProperty")]
        public void GetMyRuntimePropertPostAction(Tag objectTag, ref string objectName,
ref bool isNull)
        {
            Console.WriteLine($"{CurrentMethod.Name()}({objectTag.Value},
{objectName})");
            objectName = objectName + " " + "Post-action-for:MyRuntimeProperty";
            isNull = false;
        }
        // Property setter pre-action example
        [PropertySetterPreAction("object_desc")]
        public void OnStartSetObjectDesc(Tag obj, string value, bool isNull)
        {
            ItemRevision.Interface(_metamodel).fnd0isOwningUser(obj, "admin", out var
isOwnedByAdmin);
            if (isOwnedByAdmin)
                _sysLog.Log($"Setting object_desc to {(value is null ? "<null>" :
$"'{value}'")}");
        }

        // helper method
        private static class CurrentMethod
        {
            public static string Name()
            {
                return new StackTrace().GetFrame(1)?.GetMethod()?.Name ?? "<anon>";
            }
        }
    }
}
```

How to add a custom runtime property of other single-valued types

When you write the C# method for a runtime property, you determine the property's type simply by using the appropriate type for the method's *values* parameter:

Metamodel property type	Type of *values* parameter in the C# method
char	out IReadOnlyCollection<(Tag, char?)>
date_t	out IReadOnlyCollection <(Tag, TcDate?)>
double	out IReadOnlyCollection <(Tag, double?)>
integer	out IReadOnlyCollection <(Tag, int?)>
logical	out IReadOnlyCollection <(Tag, bool?)>
string	out IReadOnlyCollection <(Tag, string)>
tag_t	out IReadOnlyCollection <(Tag, Tag?)>

To see how this works, here's class from the previous example with a new runtime property that holds the length of the object's object_name:

```
[MetamodelTypeExtension("Item")]
public class MyItemExtensions
{
    private readonly IBulkPropertyReader<ItemRuntimePropertySource> _propertyReader;
    private readonly IMetamodel _metamodel;

    public MyItemExtensions(IBulkPropertyReaderFactory propertyReaderFactory, IMetamodel
metamodel)
    {
        _propertyReader = propertyReaderFactory.For(Item.Properties);
        _metamodel = metamodel;
    }

    [RuntimeProperty("test_property", RecalculateValue.OnEachRequest)]
    public void CalculateTestPropertyValue(IEnumerable<Tag> objects, out
IReadOnlyCollection<(Tag, string)> values)
    {
        var tagsToRead = objects.ToList();
        values = _propertyReader
            .AddItem1(item => item.object_name)
            .AddItem2(item => item.object_desc)
            .Read(tagsToRead)
            .Select(item => (item.Tag, $"{item.Item1}_{item.Item2}"))
            .ToArray();
        values = tagsToRead.Select(tag => (tag, (string)null)).ToArray();
    }

    [RuntimeProperty("another_test_property", RecalculateValue.OnEachRequest)]
    public void CalculateAnotherTestPropertyValue(IEnumerable<Tag> objects, out
IReadOnlyCollection<(Tag, int)> values)
    {
        var objectsAsList = objects.ToList();
        var itemInterface = Item.Interface(_metamodel);
        var lengths = itemInterface.get_object_name(objectsAsList)
            .Select(name => (int?)name.Length);
        values = objectsAsList.Zip(lengths).ToArray();
    }
}
```

Note that you may not *change* the type of a runtime property while the Tc Server is running; make a new property with a new name and type, or stop the server and start it again.

How to add custom extensions to a string property

In this example, we will continue to use the same C# class to add extensions to the runtime string property, *test_property*, which we wrote in the earlier section.

Note that you can add extensions to any string property on any type this way, not just to runtime properties. You can also add to a subtype an extension to a property defined on its supertype.

Property getter preactions and preconditions

Preactions and preconditions don't have access to the property value; the C# method is passed only the object's Tag. A preaction returns void, and carries a [PropertyGetterPreAction] attribute which holds the property's name and its type, which must match the type of the property.

Here's a skeleton preaction defined on test_property.

```
[PropertyGetterPreAction("test_property", TcPropertyType.String)]
public void OnStartGetTestProperty(Tag obj)
{
}
```

A precondition method is the same, except it has a [PropertyGetterPreCondition] attribute and returns an int: returning 0 allows access to the property, while a non-zero value is used to show why access was denied. Here's a skeleton precondition:

```
[PropertyGetterPreCondition("test_property", TcPropertyType.String)]
public int IsAccessToTestPropertyAllowed(Tag obj)
{
    return 0;
}
```

And here's an example of the whole class, where the precondition denies access to *test_property* if the object is owned by user *admin*, and the preaction logs attempts to read *test_property* on an Item whose object_desc is null.

```
using System.Collections.Generic;
using System.Linq;
using System.Diagnostics;
using Teamcenter.TcServer.MetamodelWrappers.foundation;
using Teamcenter.TcServer.Api;
using Teamcenter.TcServer.Api.Metamodel;
using Teamcenter.TcServer.Api.PropertyReader;

[MetamodelTypeExtension("Item")]
public class MyItemExtensions
{
    private readonly IBulkPropertyReader<ItemRuntimePropertySource> _propertyReader;
    private readonly IMetamodel _metamodel;
    private readonly ISysLog _sysLog;

    public MyItemExtensions(IBulkPropertyReaderFactory propertyReaderFactory, IMetamodel
metamodel, ISysLog sysLog)
    {
        _propertyReader = propertyReaderFactory.For(Item.Properties);
        _metamodel = metamodel;
        _sysLog = sysLog;
    }

    [PropertyGetterPreCondition("test_property", TcPropertyType.String)]
    public int IsAccessToTestPropertyAllowed(Tag obj)
    {
        Item.Interface(_metamodel).fnd0isOwningUser(obj, "admin", out var
isOwnedByAdmin);
        var allowAccess = !isOwnedByAdmin;
        return allowAccess ? 0 : 1;
    }

    [PropertyGetterPreAction("test_property", TcPropertyType.String)]
    public void OnStartGetTestProperty(Tag obj)
    {
        var desc = Item.Interface(_metamodel).get_object_desc(new[] { obj }).First();
        if (desc is null)
            _sysLog.Log("About to get a test_property value with a null object_desc");
    }

    [RuntimeProperty("test_property", RecalculateValue.OnEachRequest)]
    public void CalculateTestPropertyValue(IEnumerable<Tag> objects, out
IEnumerable<(Tag, string)> values)
    {
        values = _propertyReader
                    .AddItem1(item => item.object_name)
                    .AddItem2(item => item.object_desc)
                    .Read(objects)
                    .Select(item => (tag: item.Tag, name: item.Item1, desc: item.Item2))
                    .Select(item => (item.tag, $"{item.name}_{item.desc}"));
    }

    [RuntimeProperty("another_test_property", RecalculateValue.OnEachRequest)]
    public void CalculateAnotherTestPropertyValue(IEnumerable<Tag> objects, out
IReadOnlyCollection<(Tag, int)> values)
    {
        var objectsAsList = objects.ToList();
        var itemInterface = Item.Interface(_metamodel);
        var lengths = itemInterface.get_object_name(objectsAsList)
            .Select(name => (int?)name.Length);
        values = objectsAsList.Zip(lengths).ToArray();
    }
}
```

Property getter postactions

A property getter postaction is passed a reference to the property's value, so it can can modify the vanilla. It is decorated with a [PropertyGetterPostAction] attribute. Here's an example of a post action that truncates the value to a maximum length.

```
[PropertyGetterPostAction("test_property")]
public void ModifyTestPropertyValue(Tag tag, ref string value, ref bool isNull)
{
    if (isNull) return;

    const string ellipsis = "...";
    var maximumLength = 10;
    if (value.Length > maximumLength)
        value = $"{value.Substring(0, maximumLength - ellipsis.Length)}{ellipsis}";
}
```

At the start of the method, *value* and *isNull* are set to property's current value; you can modify them if you wish. Note that if isNull is true, you must set isNull to true *and* the value itself in order to set a non=-null value. If isNull is false, it is sufficient to update isNull to true to make it null.

Property setter extensions

These are all similar to the property getter postaction, except the value is passed without ref as can't modify the property value. Here are the method signatures for the string property object_desc[6].

```
[PropertySetterPreCondition("object_desc")]
public int AllowSetObjectDesc(Tag obj, string value, bool isNull)
{
    return 0;
}

[PropertySetterPreAction("object_desc")]
public void OnStartSetObjectDesc(Tag obj, string value, bool isNull)
{
}

[PropertySetterPostAction ("object_desc")]
public void OnEndSetObjectDesc(Tag obj, string value, bool isNull)
{
}
```

And here's an example of how a preaction might be used to log interesting events.

```
[PropertySetterPreAction("object_desc")]
public void OnStartSetObjectDesc(Tag obj, string value, bool isNull)
{
    Item.Interface(_metamodel).fnd0isOwningUser(obj, "admin", out var isOwnedByAdmin);
    if (isOwnedByAdmin)
        _sysLog.Log($"Setting object_desc to {(value is null ? "<null>" :
$"'{value}'")}");
}
```

Complete code to add a preaction on a string property:

[6] We don't use test_property in this example, as runtime properties can't be set.

You can add pre/post in the same file where property was added (in this case RuntimeProperty.cs). This code adds a post-action on "MyRuntimeProperty" property was was added earlier.

// this post-action GetMyRuntimePropertPostAction appends " Post-action-for:MyRuntimeProperty" value with the value

// from Base Action(see RuntimePropBase(…)).

```
       [PropertyGetterPostAction("MyRuntimeProperty")]
        public void GetMyRuntimePropertPostAction(Tag objectTag, ref string
objectName, ref bool isNull)
        {
            Console.WriteLine($"{CurrentMethod.Name()}({objectTag.Value},
{objectName})");
            objectName = objectName + " " + "Post-action-for:MyRuntimeProperty";
            isNull = false;
        }
```

  // Property setter pre-action example

```
       [PropertySetterPreAction("object_desc")]
        public void OnStartSetObjectDesc(Tag obj, string value, bool isNull)
        {
            ItemRevision.Interface(_metamodel).fnd0isOwningUser(obj, "admin", out
var isOwnedByAdmin);
            if (isOwnedByAdmin)
                _sysLog.Log($"Setting object_desc to {(value is null ? "<null>" :
$"'{value}'")}");
        }
```

How to add custom extensions to single-valued properties of other types

Property getter preactions and preconditions

These are written in the same way preactions and preconditions on a string property, except the type enum you specify on the method attribute must match the Metamodel property's type, as shown in the following table:

Metamodel property type	Enum used in the method attribute
char	TcPropertyType.Char
date_t	TcPropertyType.Date
double	TcPropertyType.Double
integer	TcPropertyType.Int
logical	TcPropertyType.Logical
string	TcPropertyType.String
tag_t	TcPropertyType.Tag

Property getter postactions and property setter extensions

These are the same as the extensions for a string property, except the type of the C# *value* parameter must match the Metamodel property's type, as in the following table:

Metamodel property type	Type of *value* parameter in the C# method
char	char
date_t	TcDate
double	double
integer	int
logical	bool
string	string
tag_t	Tag

Here's an example of a property getter postaction on Item.is_vi; it changes all null values to false:

```
[PropertyGetterPostAction("is_vi")]
public void AfterGettingIsVi(Tag obj, ref bool value, ref bool isNull)
{
    if (isNull)
    {
        value = false;
        isNull = true;
    }
}
```

Note that since you can't set a runtime property, setter extensions on runtime properties can't be run.

Complete example code to add a post-action to a custom property:

You can add pre/post in the same file where property was added (in this case RuntimeProperty.cs). This code adds a post-action on "MyRuntimeProperty" property was added in earlier example.

// this post-action GetMyRuntimePropertPostAction appends " Post-action-for:MyRuntimeProperty" value with the value

// from Base Action(see RuntimePropBase(…)).

```
       [PropertyGetterPostAction("MyRuntimeProperty")]
        public void GetMyRuntimePropertPostAction(Tag objectTag, ref string
objectName, ref bool isNull)
        {
            Console.WriteLine($"{CurrentMethod.Name()}({objectTag.Value},
{objectName})");
            objectName = objectName + " " + "Post-action-for:MyRuntimeProperty";
            isNull = false;
        }
```

// Property setter pre-action example

```
       [PropertySetterPreAction("object_desc")]
        public void OnStartSetObjectDesc(Tag obj, string value, bool isNull)
        {
            ItemRevision.Interface(_metamodel).fnd0isOwningUser(obj, "admin", out
var isOwnedByAdmin);
            if (isOwnedByAdmin)
                _sysLog.Log($"Setting object_desc to {(value is null ? "<null>" :
$"'{value}'")}");
        }
```

How to add a custom runtime array property

This is done in the same way as the custom runtime string property (see *How to add a custom runtime string property*), but giving the *values* parameter one of the following array types:

Metamodel property type	**Type of the *values* parameter type in the C# method**
char	`out IReadOnlyCollection<(Tag, char?[])>`
date_t	`out IReadOnlyCollection<(Tag, TcDate?[])>`
double	`out IReadOnlyCollection<(Tag, double?[])>`
integer	`out IReadOnlyCollection<(Tag, int?[])>`
logical	`out IReadOnlyCollection<(Tag, bool?[])>`
string	`out IReadOnlyCollection<(Tag, string[])>`
tag_t	`out IReadOnlyCollection<(Tag, Tag?[])>`

As an example, here's a int runtime array property

```
[RuntimeProperty("test_array_property", RecalculateValue.OnEachRequest)]
public void CalculateTestPropertyValue(IEnumerable<Tag> objects, out
IReadOnlyCollection<(Tag, int?[7][])> values)
{
    values = _propertyReader
        .AddItem1(item => item.object_name)
        .Read(objects)
        .Select(entry =>
             (entry.Tag, entry.Item1?.Select(c => (int?) Convert.ToInt32(c)).ToArray()))
        .ToArray();
}
```

How to add custom extensions to array properties

Array property getter preactions and preconditions

[7] The array type, int?[], allows you set array elements to null. But note that even though you can do this, the AOM ITK always returns null array elements as default values (i.e. 0, 0.0, "", false, etc.), and hides that fact that they are null. In similar vein it considers a null array to be identical to an empty array. IPropertyReader and IBulkProperyReader are built on AOM ITK and so have the same behaviour. The only two areas in Tc server that have access to actual null values for array elements are property getter postactions and SOA response Service Data.

These written in exactly the same way as preactions and preconditions for single-valued property getters; just use one of the following type enums in the method's attribute:

Metamodel property type	Enum used in the method attribute
char	TcPropertyType.CharArray
date_t	TcPropertyType.DateArray
double	TcPropertyType.DoubleArray
integer	TcPropertyType.IntArray
logical	TcPropertyType.LogicalArray
string	TcPropertyType.StringArray
tag_t	TcPropertyType.TagArray

As examples, here are some skeleton methods for the *test_array_property* we defined earlier.

```
[PropertyGetterPreCondition("test_array_property", TcPropertyType.IntArray)]
public int AllowGetTestArrayProperty(Tag obj)
{
    return 0;
}


[PropertyGetterPreAction("test_array_property", TcPropertyType.IntArray)]
public void OnStartGetTestArrayProperty(Tag obj)
{
}
```

Array property getter postactions

The array property getter postaction is written in the same way as the one for a single-valued property, except the *value* parameter has the type IArrayPropertyValue<T>. The method is decorated with a [PropertyGetterPostAction] attribute.

When the method runs, the *value* parameter holds the current value of the array, and allows you to modify the its contents if you wish. Here's an example of a post action that modifes the value of an integer array, based on its current contents.

```
[PropertyGetterPostAction("test_array_property")]
public void ModifyTestArrayPropertyValue(Tag tag , IArrayPropertyValue<int> value)
{
    if (value.Count > 1 && value[1].isNull) value.Set(1, 5);
    value.RemoveAt(0);
}
```

Extensions to array property element getters

When Tc server is asked for the value of an individual element from an array, the array property getter extensions aren't called, but you can write specific array property *element* extensions that are called in this case.

For the precondition and preaction, these methods are exactly like the extensions for the entire array property, except the attribute carries one of the following TcProperty types:

Metamodel property type	Type of *value* parameter in the C# method
char	TcPropertyType.CharArrayElement
date_t	TcPropertyType.DateArrayElement
double	TcPropertyType.DoubleArrayElement
integer	TcPropertyType.IntArrayElement
logical	TcPropertyType.LogicalArrayElement
string	TcPropertyType.StringArrayElement
tag_t	TcPropertyType.TagArrayElement

For example

```
[PropertyGetterPreAction("test_array_property", TcPropertyType.IntArrayElement)]
public void OnStartGetTestArrayPropertyElement(Tag obj)
{
}

[PropertyGetterPreCondition("test_array_property", TcPropertyType.IntArrayElement)]
public int AllowGetTestArrayPropertyElement(Tag obj)
{
    return 0;
}
```

The post action is very like the post action for a single-valued property, with an additional parameter to hold the element index. Here's an example of a post-action that sets an null element to the element's index.

```
[PropertyGetterPostAction("test_array_property")]
public void GetRuntimeIntArrayElementPostAction(Tag objectTag, int index, ref int value,
ref bool isNull)
{
    if (isNull)
    {
        value = index;
        isNull = false;
    }
}
```

Extensions to array property setters

An array property setter's precondition, preaction and postaction are all written by in the same way, and take the same set of parameters. Here are the basic method skeletons for Item's *bom_view_tags* property:

```
[PropertySetterPreCondition("bom_view_tags")]
public int AllowSetBvt(Tag objectTag, IReadOnlyArrayPropertyValue<Tag> value)
{
    return 0;
}

[PropertySetterPreAction("bom_view_tags")]
public void OnStartSetBvt(Tag objectTag, IReadOnlyArrayPropertyValue<Tag> value)
{
}
```

```
[PropertySetterPostAction("bom_view_tags")]
public void OnEndSetBvt(Tag objectTag, IReadOnlyArrayPropertyValue<Tag> value)
{
}
```

IReadOnlyArrayPropertyValue<T> is simply an alias for IReadOnlyList<(T value, bool isNull)>.

Extensions to array property element setters

You can write extensions to an array property *element* setter by writing the appropriate method signature. For example:

```
[PropertySetterPreCondition("bom_view_tags")]
public int AllowSetBvtElement(Tag objectTag, int index, Tag value, bool isNull)
{
    return 0;
}

[PropertySetterPreAction("bom_view_tags")]
public void OnStartSetBvtElement(Tag objectTag, int index, Tag value, bool isNull)
{
}

[PropertySetterPostAction("bom_view_tags")]
public void OnEndSetBvtElement(Tag objectTag, int index, Tag value, bool isNull)
{
}
```

How to write C# code that reads the value of a C# runtime property

All the ways of reading a property that we've looked at up to now – using IMetamodel, or IBulkPropertyReader – have relied on the property metadata being available in a generated dll. But if you've just written the property and you want to use the values in your code, how do you do it? This section shows you how[8].

Include a property in the SOA response Service Data

If you need to return the property value to a client in the SOA service data, simply include it in the AddResponse call when creating the response, e.g.

```
[SoaServiceMethod(Name = "op")]
public object MySoaOperation()
{
    var tags = _pomQuery.FromType(Item.QueryRoot)
        .Where(query => query.root.item_id.Length() == 6)
        .LoadInstancesFromDatabase();
```

[8] Eventually custom C# properties will be included in a template and thus found in the C# dll generated from the template. When this feature is available, you will be able to read them in the same way as OOTB properties.

```
    // return test_property values in the service data
    var serviceData = _serviceDataFactory.Create().AddResponse(tags, "test_property");
    return new { serviceData };
}
```

You can add as many property names as you like:

```
    var serviceData = _serviceDataFactory.Create().AddResponse(tags, "test_property",
"test_array_property");
```

Use a property value in business logic

If you need the property value in your own business logic, you can acquire it from an IPropertyReader, passing it the property name and its type, e.g. if *_propertyReader* is an IPropertyReader, you can write

```
var testArray = _propertyReader.GetIntegerArrayProperty (new [] {tag},
"test_array_property", tagIndex)
```

or

```
var element0 = _propertyReader.GetIntegerArrayPropertyElement(new [] {tag},
"test_array_property", 0)
```

and if *_itemReader* is an IBulkPropertyReader<ItemRuntimePropertySource> then you can write a lambda that returns property metadata object that matches the property's type, for example:

```
_itemReader.AddItem1(_ => new
IntegerArrayProperty<ItemRuntimePropertySource>("test_array_property")
```

And then you can read the resulting values from the Read() call.

How to add a custom operation

As before, start with a C# class that carries a MetamodelTypeExtension attribute:

```
using Teamcenter.TcServer.Metamodel;

namespace MyCustomizations
{
    [MetamodelTypeExtension("Item")]
    public class MoreItemExtensions
    {
    }
}
```

Now, you add a custom operation by writing a method with an Operation attribute. The method must return void, and it first parameter must have type Tag; this is the object's tag. Any further C# parameters you add correspond to the Metamodel operation's parameters. Use *out* parameters to return results to the caller. You should use unique prefix (similar to BMIDE) to avoid any name collision with other template(s)

Here's an example of a custom operation that takes an array of integers and returns their sum:

```
[Operation("sum")]
public void Sum(Tag objectTag, IReadOnlyList<int> values, out int sum)
{
    sum = values.Sum();
}
```

As before, you can acquire Teamcenter services through the class constructor and use them in your implementation (see *How to access Teamcenter services from a custom SOA operation*).

Defining the operation's parameters

When deciding on your operation's parameters, you are limited to using a fixed set of parameter types, as in BMIDE. These are

Input parameter types

- The supported primitive types: bool, char, double, float, int, long, string, TcDate, TcTag
- IReadOnlyList<T>
- IReadOnlySet<T>
- IReadOnlyDictionary<TKey, TValue>

Output parameter types

- An *out* parameter of any of the supported primitive types
- out ICollection<T>
- out ISet<T>
- out IDictionary<TKey, TValue>

Input / output parameter types

- A *ref* parameter of any of the supported primitive types
- IList<T>
- ISet<T>
- IDictionary<TKey, TValue>

where T, TKey, TValue are one of the supported primitive types.

Complete example code for adding an operation:

```
using System.Diagnostics;
using Teamcenter.TcServer.Api;
using Teamcenter.TcServer.Api.Metamodel;



namespace myCompanyProject
{
    [MetamodelTypeExtension("Item")]
    public class Operation
    {
        static DateTime thisDate = DateTime.Now;
        [Operation("CalculateCostOperation")]
        //  you can think CalculateSum as base action
        public void CalculateSum(Tag objectTag, IReadOnlyList<int> values, out int sum)
        {
            sum = values.Sum();
            Console.WriteLine("{0:g}", thisDate);
            Console.WriteLine("Sum is : " + sum);
            Console.WriteLine($"{CurrentMethod.Name()}");
        }
        [OperationPostAction("CalculateCostOperation")]
        public void AddSalesTax(Tag objectTag, IReadOnlyList<int> values, ref int sum)
        {
            Console.WriteLine("{0:g}", thisDate);
            Console.WriteLine($"{CurrentMethod.Name()}");
            sum *= 2;
        }

        private static class CurrentMethod
        {
            public static string Name()
            {
                return new StackTrace().GetFrame(1)?.GetMethod()?.Name ?? "<anon>";
            }
        }
    }
}
```

How to write code that calls a custom operation

There are a few scenarios where you need to know how to call a C# custom operation.

- You've written a Metamodel operation in C#, and you want to tell others how to call it from C#
- You have a custom template that someone else has written
- You've written a Metamodel operation in C#, and now you want to call it from the same assembly
- You've written a Metamodel operation in C# and now someone wants to call it from C++

We'll deal with each of these in turn.

Telling others how to call a custom Metamodel operation you've written in C#

I am assuming that you've produced a custom assembly that contains a custom operation, you have released the assembly to your production or staging site, and now you want to let people know how they can call the operation.

During the release, the process will also have created another C# assembly containing the metadata for your custom types and operations. You make this assembly available to people who want to call your operation (e.g. through NuGet). When they include your assembly in their project and add the appropriate *using* statement, they will see the new C# operation appear on the type's Metamodel interface.

Calling a Metamodel operation defined in another template

The template could have been produced by Siemens, by yourselves, or by some third party, but the important thing is that is has been released and deployed on your system. You acquire the associated metadata assembly (e.g. via NuGet) and use the operation definition that it adds to the type's Metamodel interface.

There is an example of this later on, in the section on *Setting up an operation on a mock object*.

Calling a Metamodel operation from the assembly where it's defined

But what happens if you want to write code that calls an operation in an assembly that hasn't yet been released – perhaps it's defined and called from code in the same assembly?

In this case, you don't want to call the C# method directly, as this would by-pass the any custom extensions to the operation. Instead, you request Teamcenter to dispatch the operation using the Teamcenter service to do this, IRegisteredTcOperationDispatcher.

Acquire the service through the constructor in the usual way, and then call its *Execute()* method passing in

- The object Tag
- The Metamodel operation's BMIDE name
- An object[] containing the operation's parameters, missing out the first parameter, the object Tag

As as example, we can write the following SOA operation that calls the operation *sum* defined above. Recall that we defined *sum* as follows

```
public void Sum(Tag objectTag, IReadOnlyList<int> values, out int sum)
```

The object[] of parameters we pass to *Execute()* are therefore going to contain two elements, one for *values* and one for the result, *sum*. *values* can be an int[] containing the values to be added to together (since int[] implements IReadOnlyList<int>) and *sum* can be any integer, as it will be set by the operation. Our code must read the value of *sum* when the operation returns.

Here, then, is a SOA service that uses an IRegisteredTcOperationDispatcher to execute our custom operation, *sum*, on some Item objects

Go to the file SoaService.cs (where you added SOA service method "HelloSoaOperation"), Add another SOA service method, named "callCalculateCostSoaOperation". The code _tcOperationDispatcher.Execute call will trigger execution of "CalculateCostOperation"

```
[SoaService(Name = "SoaService", Namespace = "siemens.service.2023")]
    public class SoaService
    {
        private readonly IBulkPropertyReaderFactory _propertyReaderFactory;
        private readonly IPomEnq _pomEnquiry;
        private readonly IMetamodel _metamodel;
        private readonly IServiceDataFactory _serviceDataFactory;
        private readonly IRegisteredTcOperationDispatcher _tcOperationDispatcher;

        public SoaService(IPomEnq pomQuery, IBulkPropertyReaderFactory
propertyReaderFactory, IPomEnq pomEnquiry, IRegisteredTcOperationDispatcher
tcOperationDispatcher, IMetamodel metamodel, IServiceDataFactory serviceDataFactory)
```

```
        {
            _propertyReaderFactory = propertyReaderFactory;
            _pomEnquiry = pomEnquiry;
            _metamodel = metamodel;
            _tcOperationDispatcher = tcOperationDispatcher;
            _serviceDataFactory = serviceDataFactory;
        }

        [SoaServiceMethod(Name = "HelloSoaOperation", ResponseTypeName = "result")]
        public object Hello()
        {
            return "Hello";
        }

        [SoaServiceMethod(Name = "callCalculateCostSoaOperation", ResponseTypeName =
"result")]
        public object CalculateCost()
        {
            var objectTag =
_pomEnquiry.FromType(Item.QueryRoot).LoadInstancesFromDatabase().Take(1).Single();
            var ints = new[] { 100, 200, 400, 500 };
            var parameters = new object[] { ints, default(int) };
            // "CalculateCostOperation#const,int,*$const,int,*$int,*" is similar to BMIDE
operation signature
            _tcOperationDispatcher.Execute(objectTag,
"CalculateCostOperation#const,int,*$const,int,*$int,*", parameters);
            var result = new
            {
                sum = new { uiValues = new[] { (int)parameters.Last() } }
            };
            return result;
        }
    }
```

Where does this name, *sum#const,int,*$const,int,*$int,**, come from? This is the name that BMIDE gives the operation in order to allow operation overloading. But how do you know what your custom operation's BMIDE name is? The simplest way is to write the C# operation, start a Tc server instance (or an ITK program) and allow it to intialize, and then search in the syslog for a 'xxx#' where 'xxx' is the operation's name. You should find something like this

```
2022/05/03-10:41:25.469 UTC - Successfully registered Item.sum#const,int,*$const,int,*$int,*()
```

Which contains the full BMIDE name.

Calling a Metamodel operation from C++ code

Use the ITK call METHOD_execute() to call the method, passing the in the object's tag and the method id, which you can get by passing the operation's BMIDE name to METHOD_get_message_id(). You also need to pass in the operation's parameters as C values[9]. The easiest way of finding the C types you need to pass is by reading its BMIDE name[10].

[9] C# arrays are passed as a C array followed by an int length. C# dictionaries are passed as a C array of keys and another of values. Out and ref parameters are passed as pointers.
[10] This isn't so easy; we may need to provide some tooling to make this easier to do.

How to add a custom extension to a Metamodel operation

You can add preconditions, preactions and postactions to any Metamodel operation, irrespective of whether it has been authored in C++ or C#.

Add a custom extension to a C# Metamodel operation

Write a C# with the same parameters as the original Metamodel operation. For a postaction, it often makes sense to change the operation's *out* parameters to *ref* parameters, as this gives you the option of inspecting the value set by the base operation. For preconditions and preactions, using *ref* instead of *out* means the method doesn't have to set the parameter value, which C# semantics would otherwise require.

A precondition returns an int (0 for allow access, non-zero to hold the reason for denying access). The preaction and postaction return void.

Here, then, are skeleton extensions for the *sum* operation above.

```
[MetamodelTypeExtension("Item")]
public class MoreItemExtensions
{
    [Operation("sum")]
    public void Sum(Tag objectTag, IReadOnlyList<int> values, out int sum)
    {
        sum = values.Sum();
    }

    [OperationPreCondition("sum#const,int,*$const,int,*$int,*")]
    public int IsSumAllowed(Tag objectTag, IReadOnlyList<int> values, ref int sum)
    {
        return 0;
    }

    [OperationPreAction("sum#const,int,*$const,int,*$int,*")]
    public void OnStartSum(Tag objectTag, IReadOnlyList<int> values, ref int sum)
    {
    }

    [OperationPreAction("sum#const,int,*$const,int,*$int,*")]
    public void OnEndSum(Tag objectTag, IReadOnlyList<int> values, ref int sum)
    {
    }
}
```

Complete example code for adding post-action:

In this example we will add a post-action on the operation("CalculateCostOperation") added earlier.

From project, open Operation.cs, add the following:

```
[OperationPostAction("CalculateCostOperation")]
        public void AddSalesTax(Tag objectTag, IReadOnlyList<int> values, ref int
sum)
        {
            Console.WriteLine("{0:g}", thisDate);
            Console.WriteLine($"{CurrentMethod.Name()}");
            sum *= 2;
        }
```

That’s all you need to do. This post-action will be called when `CalculateCostOperation` gets executed.

Add a custom extension to a Metamodel operation defined in a template

If the operation you want to extension is defined in a template, use the TcOperationNames object to acquire the object name. For example, to write a postaction on Item.isMature, you would write

```
[OperationPostAction(FoundationTemplate.TcOperationNames.isMature)]
public void OnEndIsMature(Tag objectTag, ref bool isMature)
{
}
```

How to access Teamcenter services from a custom SOA operation

In order to have the SOA operation do anything interesting with Teamcenter data, it's going to need access to Teamcenter services. In this section we're going to see how to access those services; we'll look at how to use these services in more detail later on.

The following services are currently available:

C# interface	Service
IAom IAomProp	Provide access to low-level ITK methods. But it's recommended to use the easier and safer IMetamodel or IBulkPropertyReaderFactory interfaces in preference to these.
IBulkPropertyReaderFactory	Provides an efficient way of reading sets of property values from homogenous collections of Metamodel objects. Use this interface in preference to IMetamodel when reading more than one property value.
IMetamodel	Provides C# interfaces that give acces to operations and property setters and getters defined on Teamcenter Metamodel types.
IPomApi	Provides access to a few low-level POM ITK calls
IPomQuery	Provides access to POM ENQ queries
IPropertyReader	Provides access to the values of Metamodel properties that aren't defined in Teamcenter templates[11]. But if the property *is* defined in a template, it's recommended to use the easier and safer interfaces available from IMetamodel.
IRegisteredTcOperationDispatcher	Provides low-level access to Metamodel operations. Use this only for calling Metamodel operations that aren't defined in a template. Otherwise, use the interfaces available from IMetamodel.
IServiceDataFactory	Provides access to ServiceData objects that are used to produce Teamcenter service data in SOA responses.

You acquire these services by adding a parameter of the service's type to the public constructor of the SOA service class. You can add as many different services as you need to the constructor. Store the parameter values in fields and use them in the service operations. There are some examples of how this is done in the following sections.

This remainder of the section gives some simple examples of how to use some of the more common services from a SOA operation. The services themselves are documented in full later on.

IPomQuery

IPomQuery executes POM queries. For example, here's the service class we created earlier, but this time with a constructor that acquire the IPomQuery service, a and private field that is used to store it:

[11] That is, use this ro read properties that are defined only in C# code. See *How to write C# code that reads the value of a C# runtime property*.

```
[SoaService(Name = "example.service", Namespace = "example.service.2022")]
public class MySoaService
{
    private readonly IPomQuery _pomQuery;

    public MySoaService(IPomQuery pomQuery)
    {
        _pomQuery = pomQuery;
    }

    [SoaServiceMethod(Name = "op", ResponseTypeName = "result")]
    public object MySoaOperation()
    {
        return new { };
    }
}
```

And here's how to use use the IPomQuery service in the operation itself to load the objects that meet a certain condition and return their Tags:

```
[SoaServiceMethod(Name = "op")]
public object MySoaOperation()
{
    return _pomQuery.FromType(Item.QueryRoot)
        .Where(query => query.root.item_id.Length() == 6)
        .LoadInstancesFromDatabase();
}
```

IServiceDataFactory, IServiceData

Up to this point, the operation has only returned data that has an arbitrary format; that is, the format of the response is peculiar to the request. But what if you want to return standard Tc SOS service data, that has the standard SOA format, so that an an existing SOA client can consume it? With IServiceDataFactory and IServiceData, you can do this easily.

There are three parts to this: first acquire an IServiceDataFactory service by including in the constructor's list of services; then, when your operation runs, use the factory to create an IServiceData. Finally, before your operation returns, populate the IServiceData object with the tags of the objects you want to return in your response and include the object in your response.

This is how it is used:

```
using System.Linq;
using FoundationTemplate;
using Teamcenter.TcServer.Metamodel;
using Teamcenter.TcServer.Query;
using Teamcenter.TcServer.Soa;

[SoaService(Name = "example.service", Namespace = "example.service.2022")]
public class MySoaService
{
    private readonly IPomQuery _pomQuery;
    private readonly IServiceDataFactory _serviceDataFactory;

    public MySoaService(IPomQuery pomQuery, IServiceDataFactory serviceDataFactory)
    {
        _pomQuery = pomQuery;
        _serviceDataFactory = serviceDataFactory;
    }

    [SoaServiceMethod(Name = "op")]
    public object MySoaOperation()
    {
        var tags = _pomQuery.FromType(Item.QueryRoot)
            .Where(query => query.root.item_id.Length() == 6)
            .LoadInstancesFromDatabase();
        var serviceData = _serviceDataFactory.Create().AddResponse(tags);
        return new { serviceData };
    }
}
```

When you return tags in a ServiceData object like this, the SOA response contains the complete ServiceData json populated with the objects' property values according to the current property policy. IServiceData provides other methods that allow you to build a variety of SOA responses; it is documented fully later on.

IMetamodel

Given a collection of Tags, IMetamodel affords you access to a typed interface to the Metamodel object it represents. You acquire an IMetamodel by adding it to the constructor's list of services, and use it to obtain an interface specific to the Metamodel type you request. The interface gives you access to all the object's properties and operations[12].

Here's an example that returns a Tc SOA service data response for each Item whose item_id has six characters; Items with duplicate item_ids are weeded out from the response.

```
using System.Linq;
using FoundationTemplate;
using Teamcenter.TcServer.Metamodel;
using Teamcenter.TcServer.Query;
using Teamcenter.TcServer.Soa;

[SoaService(Name = "example.service", Namespace = "example.service.2022")]
public class MySoaService
{
    private readonly IPomQuery _pomQuery;
    private readonly IServiceDataFactory _serviceDataFactory;
    private readonly IItem _itemInterface;

    public MySoaService(IPomQuery pomQuery, IServiceDataFactory serviceDataFactory,
IMetamodel metamodel)
    {
        _pomQuery = pomQuery;
        _serviceDataFactory = serviceDataFactory;
        _itemInterface = Item.Interface(metamodel);
    }

    [SoaServiceMethod(Name = "op")]
```

[12] Except operations that are missing from the template, such as those you've recently written in C#.

```
    public object MySoaOperation()
    {
        // read the item tags from the database
        var tags = _pomQuery.FromType(Item.QueryRoot)
            .Where(query => query.root.item_id.Length() == 6)
            .LoadInstancesFromDatabase();

        // read item_id property value for each object
        var itemIds = tags.Zip(_itemInterface.get_item_id(tags),
            (tag, itemId) => (tag, itemId));

        var selected = itemIds
            // group by item_id
            .ToLookup(item => item.itemId, item => item)
            // weed out duplicates (by discarding arbitrarily)
            .Select(item => item.First().tag);

        // return the selected objects in the service data
        var serviceData = _serviceDataFactory.Create().AddResponse(selected);
        return new { serviceData };
    }
}
```

IBulkPropertyReaderFactory, IBulkPropertyReader

If your business logic depends on the values of several properties, consider using these bulk services where performance is important. Although they don't provide any features that aren't already available through IMetamodel, they do provide a more efficient way of reading multiple properties.

You acquire an IBulkPropertyReaderFactory through the constructor in the normal way, and use it to obtain the IBulkPropertyReader specific to the Metamodel type. You use the reader by calling *Add()* for each property you want to read, concluding with a single call to *Read()*. It returns a collection of entities that have the objects' Tags paired with their property values.

In this example, we've reimplemented the operation above to use the bulk services instead of IMetamodel.

```
using System.Linq;
using FoundationTemplate;
using Teamcenter.TcServer.Metamodel;
using Teamcenter.TcServer.Query;
using Teamcenter.TcServer.Soa;

[SoaService(Name = "example.service", Namespace = "example.service.2022")]
public class MySoaService
{
    private readonly IPomQuery _pomQuery;
    private readonly IServiceDataFactory _serviceDataFactory;
    private readonly IBulkPropertyReader<ItemRuntimePropertySource> _reader;

    public MySoaService(IPomQuery pomQuery, IServiceDataFactory serviceDataFactory,
IBulkPropertyReaderFactory readerFactory)
    {
        _pomQuery = pomQuery;
        _serviceDataFactory = serviceDataFactory;
        _reader = readerFactory.For(Item.Properties);
    }

    [SoaServiceMethod(Name = "op")]
    public object MySoaOperation()
    {
        // read the item tags from the database
        var tags = _pomQuery.FromType(Item.QueryRoot)
            .Where(query => query.root.item_id.Length() == 6)
            .LoadInstancesFromDatabase();

        // read item_id and is_vi for each object
        var properties = reader.AddItem1(item => item.item_id)
            .AddItem2(item => item.is_vi)
            .Read(tags);

        var selected = properties
            // find objects where is_vi is null or false,
            .Where(item => !item.Item2 ?? true)
            // group by item_id
            .ToLookup(item => item.Item1, item => item)
            // weed out duplicates (by discarding arbitrarily)
            .Select(item => item.First().Tag);

        // return the selected objects in the service data
        var serviceData = _serviceDataFactory.Create().AddResponse(selected);
        return new { serviceData };
    }
}
```

ISysLog

ISysLog allows you to write entries the to the Tc server log file. Acquire the service through the constructor. Call

- Log(string message) to write a formatted message to the file
- LogRaw(string message) to write an unformatted message to the file
- ReportSeriousError(string message) to write a fatal error message to the file and terminate the Tc server instance

There is no support currently for writing entries conditionally on the Tc server debugging level, since the ITK interfaces don't support this.

How to write POM queries

The customization framework provides a new interface, IPomQuery, that wraps POM Enquiry ITK to provide a fluent, type-safe way of constructing and running POM queries.

The style of the interface is similar like C#'s Linq, and particularly like Linq-to-sql. Like Linq, it makes extensive use of lambdas. Like Linq-to-sql, the lambdas aren't actually executed, but they are used to form the underlying query. The query is built up with successive calls to POM Enquiry ITK, so it carries the same feature limitations that the POM Enquiry has.

Queries

1. Start off with a root type
2. Then have optional joins to other types
3. May then have a *where* expression to filter the values returned
4. And finish either by selecting attributes (and optionally grouping and / or ordering the results) or by loading objects from the database

Here are a couple of examples. The first one loads all the Items owned by user *admin*. The second fetches item_id and object_name attributes from Items whose item_ids start with '00' and whose object_name isn't null:

```
IEnumerable<Tag> tags = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_owning_user)
    .Where(query => query.join1.user_name == "admin")
    .SelectInstances(query => query.root)
    .LoadInstancesFromDatabase();

IEnumerable<(string itemId, string objectName)> results =
_pomQuery.FromType(Item.QueryRoot)
    .Where(query => query.root.item_id.IsLike("00%") && query.root.object_name != null)
    .SelectColumn(query => query.root.item_id)
    .SelectColumn(query => query.root.object_name)
    .ExecuteOnDatabase();
```

(These examples, and all the examples in this section, assume we have access to a field, _pomQuery, of type IPomQuery.)

Basic queries

Start a query with a root type

You start a query by calling IPomQuery.FromType(). Pass in the POM type you want to be the query's root type. You can find all the C# type definitions in the template metadata assemblies. In the example above were are using Item.QueryRoot from FoundationTemplate. You can use any QueryRoot object.

Join in the other types your query needs

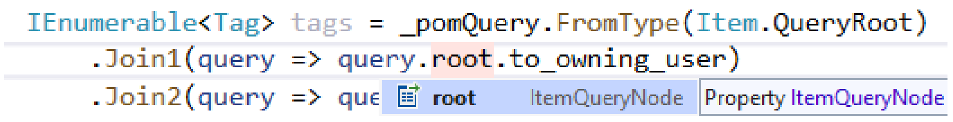
If you want to include other types in the query, you must do this immediately after the call to IPomQuery.FromType() by calling Joinx(). The first call is Join1(), the second Join2(), etc. So for example, we can write

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_owning_user)
    .Join2(query => query.root.to_itemrevision_items_tag)
    ...
```

Each call to Join() changes the query's type by adding in the type being joined.

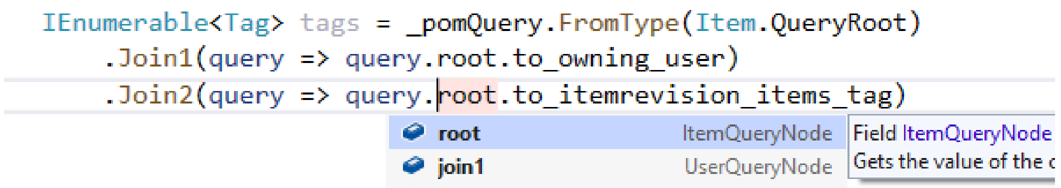
In the example above, calling IPomQuery.FromType(Item.QueryRoot) returns a query based on (root: Item). The call to Join1() returns a query based on (root: Item, join1: User). The call to Join2() returns a query based on (root: Item, join1: User, join2: ItemRevision).

Each call to Joinx() takes a lambda that operates on the query's current type. So, in the call to Join1(), intellisense shows the *query* parameter with a member called *root*; *root* is the query's root type, namely Item:



```
IEnumerable<Tag> tags = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_owning_user)
    .Join2(query => que
```

and in the Join2 call, intellisense shows *query* having two members, *root* and *join1*



```
IEnumerable<Tag> tags = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_owning_user)
    .Join2(query => query.root.to_itemrevision_items_tag)
```

So, in these lambdas we are writing, the *query* parameter has the query's type and its members *root*, *join1*, *join2*, etc. hold the POM types the query is based on.

There are several overloads of Joinx():

- Join via a POM reference on the object to the referenced object
  Takes a single lambda that selects the one of the type's POM references. In the example above, *query.root.to_owning_user* uses the POM reference from Item to its owning User to make the join.

  This overload also accepts an inverse reference, going from the an object's uid to an object that references it. *query.root.to_itermrevision_items_tag* is an example of this, joining from the Item's uid to the ItemRevision's *items_tag* reference.

- Join via a POM reference to the referenced object, specifying the kind of join
  As above, but this overload also allows you specify the kind of join: (left) outer join, full outer join, right outer join, or inner join.

  ```
  var result = _pomQuery.FromType(Item.QueryRoot)
      .Join1(query => query.root.to_owning_user, JoinKind.Outer)
      ...
  ```

- Join to a VLA
  Takes a single lambda that selects one of the type's VLAs, e.g.

  ```
  var result = _pomQuery.FromType(Item.QueryRoot)
      .Join1(query => query.root.ead_paragraph)
      ...
  ```

  For more information on VLAs, see the later section on *Querying with VLAs*.

- Join to an arbitrary POM type or CTE
  Takes a QueryRoot object, e.g.

  ```
  var result = _pomQuery.FromType(Item.QueryRoot)
      .Join1(ItemRevision.QueryRoot)
      ...
  ```
  You can add further constraints to the query in the Where clause or include a constraint using the next overload:

- Join with a constraint to an arbitrary POM type or CTE
  Takes a query root and a lambda that defines the constraint. In this overload, the lambda's parameter is the query's type *after* the join. For example, in this call to Join1, the lambda's query parameter has a *join1* member, allowing you to write the join constraint, as here:

  ```
  var result = _pomQuery.FromType(Item.QueryRoot)
      .Join1(ItemRevision.QueryRoot,
             query => query.root.item_id == query.join1.item_revision_id)
      ...
  ```

  The query joins Items to Item revisions based on matched the item_id with the item_revision_id.

Note that you can safely join in the same type twice, each time you join the type in, it is given a different name, allowing you to write a query keeps the two distinct, e.g.

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(ItemRevision.QueryRoot)
    .Join2(ItemRevision.QueryRoot)
    .Where(query => query.root.item_id == query.join1.object_name &&
        query.root.object_name == query.join2.item_revision_id)
    ...
```

Add a where clause

Add a where clause by calling Where(). Where() takes a lambda that operates on the query's type; it returns a logical expression, e.g.

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
```

```
    .Join2(query => query.join1.to_owning_user)
    .Where(query => query.join2.user_name == "admin")
    ...
```

Although you can only call Where() once, you can make the expression as complex as you like, using && and || operators:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .Where(query => query.join2.user_name == "admin" ||
        (query.root.item_id.Length() > 5 || query.root.object_name.Length() <= 6)
        && query.root.is_vi == null)
    ...
```

You may use any of the following methods, which are defined on POM attributes according to their type. They all translate to their corresponding POM Enquiry functions.

Attribute type: char	
Method	**Return type**
ToAscii()[13]	int

Attribute type: string	
Method	**Return type**
IsLike(string pattern)	bool
IsNotLike(string pattern)	bool
Substring(int startIndex, int length)	string
ToLower()	string
ToUpper()	string
ToNumberOrNull()[14]	double
TrimStart()	string
TrimEnd()	string

Attribute type: Tag	
Method	**Return type**
Uid()	string

Attribute type: POM reference	
Method	**Return type**
Cpid()	int

Attribute type: VLA

[13] Although this function is provided, it the implementation in POM Enquiry is faulty, and so this function always throws.
[14] Null is returned if the conversion to a number fails

Method	Return type
Length()	int

All attribute types	
Method	**Return type**
In(val1, val2, val3, …)	bool
In(subquery)	bool
NotIn(val1, val2, val3, …)	bool
NotIn(subquery)	bool
OrIfNull[15](val)	attribute type

Using subqueries in the Where() clause

Subqueries with In() and NotIn()

In() and NotIn() can be used with statically-defined lists of values, e.g.

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_owning_user)
    .Where(query => query.join1.user_name.In("admin", "adm", "superuser"))
    ...
```

but they can also take a subquery.

You write a subquery by using the StartSubQuery() method on the *query* parameter passed to the Where() method's lambda. For example, look at this query which returns items whose owning users are based in the US:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_owning_user)
    .Where(query => query.root.owning_user.In(
        query.StartSubQuery.FromType(User.QueryRoot)
        .Where(subq => subq.root.geography == "US")
        .SelectColumn(subq => subq.root.uid))
    )
    ...
```

The subquery is a different object from the main query, but it is built in the same way, calling Join(), Where() and so on, just like the main query. It must finish with a single call to SelectColumn, selecting a column of the matching type.

You can correlate the subquery to the outer query by using both query lambda parameters in the subquery's where clause, e.g.

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_owning_user)
    .Where(query => query.root.owning_user.In(
        query.StartSubQuery.FromType(User.QueryRoot)
        .Where(subq => subq.root.nationality == query.root.object_name)
```

[15] ie, coalesce

```
        .SelectColumn(subq => subq.root.uid))
    )
    ...
```

Subqueries with Exists() and DoesNotExist()

In a similar vein, you can add EXISTS and NOT EXISTS subqueries to your where clause by writing the subquery and callint Exists() or DoesNotExist() on it:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_owning_user)
    .Where(query =>
        query.StartSubQuery.FromType(User.QueryRoot)
        .Where(subq => subq.root.nationality == query.root.object_name).Exists()
    )
```

Using a query to load objects

Queries with no joins

If a query hasn't any joins, you can load objects the objects it returns by calling LoadInstancesFromDatabase(), e.g.

```
var tags = _pomQuery.FromType(Item.QueryRoot)
    .Where(query => query.root.is_vi == true)
    .LoadInstancesFromDatabase();
```

Note that you don't select the object's uids, since it's clear from the query which objects are to be loaded. POM loads the instances, checking them for access, and returns the objects that were loaded successfully.

The value returned is an IReadOnlyList<Tag>.

Queries with joins

If the query has one or more joins, you can opt either to load objects from the query's root type or one of its joined types, or to load the entire graph of joined objects.

To load objects from one of the query's types, call SelectInstances() to select the joined type whose instances to be loaded, and then call LoadInstancesFromDatabase() as above, e.g.

```
var tags = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_owning_user)
    .Where(query => query.join1.user_name == "admin")
    .SelectInstances(query => query.root)
    .LoadInstancesFromDatabase();
```

Alternatively, calling LoadInstanceGraphFromDatabase() loads all the objects spanned by the query and returns them as a directed graph. Here's an example:

```
var graph = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Where(query => query.root.is_vi == true)
    .LoadInstanceGraphFromDatabase();
```

Graph queries are useful where your query returns a collection of related objects, and where you want information on the relationships between the objects as well as the objects themselves.

The shape of the resulting graph – which nodes are connected to which – is determined by the shape of the Join() relationships. Consider the following examples that join in types via POM references:

Query	Graph

`var graph = _pomQuery.FromType(Item.QueryRoot)`
`    .Join1(query => query.root.to_itemrevision_items_tag)`
`    .LoadInstanceGraphFromDatabase();`
	`root -> join1`

`var graph = _pomQuery.FromType(Item.QueryRoot)`
`    .Join1(query => query.root.to_itemrevision_items_tag)`
`    .Join2(query => query.root.to_owning_user)`
`    .LoadInstanceGraphFromDatabase();`
	`root -> join1`
	`     -> join2`

`var graph = _pomQuery.FromType(Item.QueryRoot)`
`    .Join1(query => query.root.to_itemrevision_items_tag)`
`    .Join2(query => query.join1.to_owning_user)`
`    .LoadInstanceGraphFromDatabase();`
	`root -> join1 -> join2`

`var graph = _pomQuery.FromType(Item.QueryRoot)`
`    .Join1(query => query.root.to_itemrevision_items_tag)`
`    .Join2(query => query.root.to_uom_tag)`
`    .Join3(query => query.join1.to_owning_user)`
`    .LoadInstanceGraphFromDatabase();`
	`root -> join1 -> join3`
	`     -> join2`

If you use the Join() overload that joins to an arbitrary POM class, the joined node is made a child of root. If you use the overload that joins to an arbitrary POM class with a constraint and the constraint expresses a join relationship, then the resulting graph shows that relationship; this doesn't happen, though, if some other kind of constraint is used.

As an example, consider the following queries with three types: Item as root, ItemRevision at join1 and User at join2. The first query joins to User without a constraint, so join2 is made a child of root. The second joins to User with a constraint that expresses a relationship between ItemRevision and User, so join2 is a child of join1. The third shows the same query, but where the constraint doesn't show any relationship, so join2 is a child of root.

Query	Graph

`var graph = _pomQuery.FromType(Item.QueryRoot)`
`    .Join1(query => query.root.to_itemrevision_items_tag)`
`    .Join2(User.QueryRoot)`
`    .Where(query =>`
`        query.join1.owning_user == query.join2.uid)`
`    .LoadInstanceGraphFromDatabase();`
	`root -> join1`
	`     -> join2`

`var graph = _pomQuery.FromType(Item.QueryRoot)`
`    .Join1(query => query.root.to_itemrevision_items_tag)`
`    .Join2(User.QueryRoot,`
`        query => query.join1.owning_user == query.join2.uid)`
`    .LoadInstanceGraphFromDatabase();`
	`root -> join1 -> join2`

`var graph = _pomQuery.FromType(Item.QueryRoot)`
`    .Join1(query => query.root.to_itemrevision_items_tag)`
`    .Join2(User.QueryRoot,`
`        query => query.join2.user_name == "infodba")`
`    .LoadInstanceGraphFromDatabase();`
	`root -> join1`
	`     -> join2`

The returned value is a graph where each node in the graph represents an object that was loaded successfully. The node's children represent the objects that were joined to it through to the node's object. Each node also holds the node's name, *root*, *join1*, etc.

Here's an example of a simple function that recurses depth-first over a graph:

```
var graph = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.root.to_uom_tag)
    .Join3(query => query.join1.to_owning_user)
    .LoadInstanceGraphFromDatabase();
WriteGraphNode(graph);

static void WriteGraphNode(IReadOnlyCollection<IObjectGraphNode> graphNodes)
{
    foreach (var node in graphNodes)
    {
        Console.WriteLine($"At node {node.Name} with tag {node.Tag}");
        WriteGraphNode(node.Children);
    }
}
```

Note that the output is the graph of *loaded* objects. If an object fails to load, for example because access is denied, then its child objects are not included in the graph, even though it might have been possible to reach them through the query and to load them successfully.

Using a query to read POM attributes

It is important to understand that writing queries to read attribute values by-passes all access rules, making the attribute values available irrespective of whether the logged in user is authorised to read them or not. When you write queries that read attributes directly, you become responsible for ensuring that the security of your system isn't comprised.

Reading simple values

To read attribute values from a query directly, without loading the objects, you call SelectColumn() after the joins and the where clause, passing in a lambda that selects the attribute you want to retrieve. You call SelectColumn once for each attribute you want and then execute the query by calling ExecuteOnDatabase(). Here's an example:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .SelectColumn(query => query.root.item_id)
    .SelectColumn(query => query.join1.item_revision_id)
    .SelectColumn(query => query.join2.user_name)
    .ExecuteOnDatabase();
```

The resulting type is an IReadOnlyList<(T1, T2, T3)> where T1, T2, T3 correspond to the types of the selected attributes[16].

You can select an expression, for example

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .SelectColumn(query => query.root.item_id.Substring(0, 6) + "_A")
    .ExecuteOnDatabase();
```

You can also select a literal value, such an int, TcDate, string, etc.:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .SelectColumn(_ => SelectedLiteral.Value(2))
    .ExecuteOnDatabase();
```

Reading aggregated values

The following aggregate functions are available to use in *SelectColumn()* calls on attributes, according to their type:

Function	Available on attribute of type	Resulting expression type
.Count()	All types	int
.CountAll()	All types	int
.CountDistinct()	All types	int
.Avg()	double, int	double
.Sum()	double, int	Same type as the attribute
.Max()	All types	Same type as the attribute
.Min()	All types	Same type as the attribute

Here's an example that returns the three different counts of the Items' object_desc attributes for Items belonging to user *admin*:

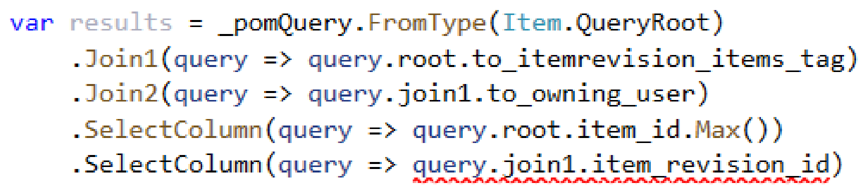
```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .Where(query => query.join2.user_name == "admin")
    .SelectColumn(query => query.root.object_desc.Count())
    .SelectColumn(query => query.root.object_desc.CountAll())
    .SelectColumn(query => query.root.object_desc.CountDistinct())
    .ExecuteOnDatabase();
```

And, as with simple values, you can select expressions based on aggregated values:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .Where(query => query.join2.user_name == "admin")
    .SelectColumn(query => query.root.object_desc.Count() + 1)
    .ExecuteOnDatabase();
```

[16] In the general case, the result is an IReadOnlyList<(T1, T2, …)>. Where only one column is selected, the returned type is IReadOnlyList<ValueTuple<T1>>.

In this kind of query where there is no grouping, you must select either all aggregated expressions, or none; you can't select a mixture of aggregated and non-aggregated values. Should you try to mix them, you'll get a compilation error like this



```
var results = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .SelectColumn(query => query.root.item_id.Max())
    .SelectColumn(query => query.join1.item_revision_id)
```

Reading attributes from grouped objects

Organise your objects into groups by calling GroupByn() after the joins and the where clause. You can call GroupBy() once for each group key you want to add. The first call is to GroupBy1(), then GroupBy2() etc. Pass the GroupByn() method a lambda that selects the group key.

Here's an example that groups each Item/ItemRevision/User by the user's name.

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .GroupBy1(query => query.join2.user_name)
    ...
```

And here's an example where we're grouping on an expression:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .GroupBy1(query => query.join2.user_name + "+" + query.join2.os_username)
    ...
```

although it would be more natural in most cases to use two calls to GroupBy:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .GroupBy1(query => query.join2.user_name)
    .GroupBy2(query => query.join2.os_username)
    ...
```

Once you've added all the group keys, you can call SelectCoumn() and ExecuteOnDatabase() as above. After a call to GroupBy(), the SelectColumn() calls accept only aggregated expressions.

You will see an additional method alonside SelectColumn(): SelectGroupKey(). This method takes a lambda that operates on a *keys* parameter. *keys* has members *key1*, *key2*, etc. that reference the keys specified in the preceding calls to GroupBy1(), GroupBy2() etc.

Here is the previous example with an additional call to SelectGroupKeys() that selects the owning user's name:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .GroupBy1(query => query.join2.user_name)
    .SelectGroupKey(keys => keys.key1) // Select the owning user name
    ...
```

And here the same query but grouped this time by the user's nationality and IP clearance and selecting both the keys:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .GroupBy1(query => query.join2.nationality)
    .GroupBy2(query => query.join2.ip_clearance)
    .SelectGroupKey(keys => keys.key1) // Select the nationality
    .SelectGroupKey(keys => keys.key2) // and the IP clearance
    ...
```

You may mix SelectGroupKey and SelectColumn calls, for example:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .GroupBy1(query => query.join2.nationality)
    .GroupBy2(query => query.join2.ip_clearance)
    .SelectGroupKey(keys => keys.key1)
    .SelectColumn(query => query.root.object_desc.Count())
    .SelectGroupKey(keys => keys.key2)
    ...
```

And of course, you may select expressions too.

And at the end, you execute the query by calling ExecuteOnDatabase, as with any attributes query.

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .GroupBy1(query => query.join2.nationality)
    .GroupBy2(query => query.join2.ip_clearance)
    .SelectGroupKey(keys => keys.key1)
    .SelectColumn(query => query.root.object_desc.Count())
    .SelectGroupKey(keys => keys.key2)
    .SelectColumn(query => query.root.object_desc.Count())
    .ExecuteOnDatabase();
```

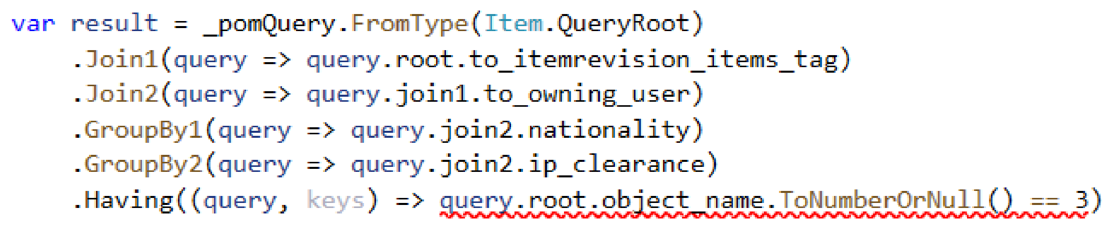
Using Having() to filter groups

After the calls to GroupByn(), you may add a single call to Having() in order to filter the groups the query returns. The call is similar to the Where() call we've seen above. It takes a lambda that operates on the query type *and* the groups' keys, and returns a logical expression; the expression may be built from the key values and from any aggregated expressions based on the query type.

In this example, query.keys.key1 is the first group key, i.e. the user's nationality.

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .GroupBy1(query => query.join2.nationality)
    .GroupBy2(query => query.join2.ip_clearance)
    .Having((query, keys) => query.root.object_name.Max().ToNumberOrNull() == 3
        && keys.key1 == "US")
    ...
```

If you try to use unaggregated values from the query type attribute, you'll get a compilation error, as shown here:



```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .GroupBy1(query => query.join2.nationality)
    .GroupBy2(query => query.join2.ip_clearance)
    .Having((query, keys) => query.root.object_name.ToNumberOrNull() == 3)
```

Querying with VLAs

Using the VLA length

A POM VLA has a .Length() method that gives you access to the VLA's length while avoiding a join to the VLA itself. Here's an example where we're selecting the length of the VLA *ead_paragraph*, and also using the length in the Where() clause.

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Where(query => query.root.ead_paragraph.Length() > 10)
    .SelectColumn(query => query.root.item_id)
    .SelectColumn(query => query.root.ead_paragraph.Length())
    .ExecuteOnDatabase();
```

And of course you can use the length in grouped queries too, for example in this query that groups the Items by the VLA length and selects the results:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .GroupBy1(query => query.root.ead_paragraph.Length())
    .Having((_, keys) => keys.key1 > 10)
    .SelectGroupKey(keys => keys.key1)
    .SelectColumn(query => query.root.uid.Count())
    .ExecuteOnDatabase();
```

Joining to a VLA

Joining to a VLA is straighforward. In this example, we join from Item to its ead_paragraph VLA:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.ead_paragraph)
    ...
```

and now we can select the *pval* and *pseq* columns from the VLA:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.ead_paragraph)
    .SelectColumn(query => query.root.item_id)
    .SelectColumn(query => query.join1.pval)
    .SelectColumn(query => query.join1.pseq)
    .ExecuteOnDatabase();
```

or we can simply use the values elsewhere in the query, as here, where we use a subquery to select instances whose VLAs contain strings that match a particular pattern:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .Where(query => query.StartSubQuery.FromType(Item.QueryRoot)
        .Join1(subq => subq.root.ead_paragraph)
        .Where(subq => subq.root.uid == query.root.uid
            && subq.join1.pval.Substring(2, 4) == "0000").Exists())
    .LoadInstancesFromDatabase();
```

Joining through a Tag VLA to the referenced objects

This requires two joins: one to the VLA, and a second one from the VLA to the objects it references. We use the joined VLA's *to_pval_reference* member to make the second join.

Here's an example where we joining via the Item's project_list VLA to the Project objects:

```
var graph = _pomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.project_list)
    .Join2(query => query.join1.to_pval_reference)
    .LoadInstanceGraphFromDatabase();
```

Note that the query graph produced by this query skips over the VLA; it considers the Project objects at *join2* to be direct children of the Items at *root*.

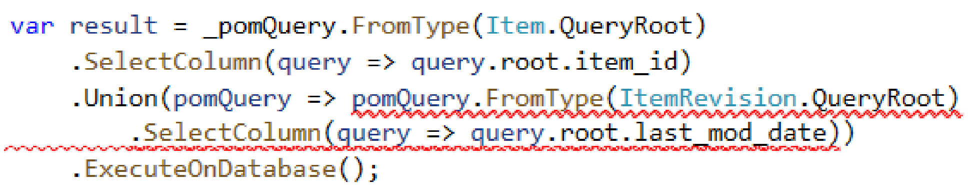
Querying with UNION, UNION ALL, INTERSECTION and DIFFERENCE

The query API allows you to join attribute queries together using UNION, UNION ALL, INTERSECTION and DIFFERENCE.

Write the first query, but instead of calling ExecuteOnDatabase(), call the Union(), UnionAll(), Intersection() or Difference() method. The methods take a lambda that operates on an IStartQueryForSetOperation; this is very IPomQuery, and allows you to write the second query in the set:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .SelectColumn(query => query.root.item_id)
    .Union(pomQuery => pomQuery.FromType(ItemRevision.QueryRoot)
        .SelectColumn(query => query.root.item_revision_id))
    .ExecuteOnDatabase();
```

You can join any two queries together in this way, as long as they produce the same column types. If the column types don't agree, you will get a compiler error:



```
var result = _pomQuery.FromType(Item.QueryRoot)
    .SelectColumn(query => query.root.item_id)
    .Union(pomQuery => pomQuery.FromType(ItemRevision.QueryRoot)
        .SelectColumn(query => query.root.last_mod_date))
    .ExecuteOnDatabase();
```

With Union(), UnionAll() and Intersection(), you can chain any number of queries together in this way, as long as you use the same set operation throughout the whole chain. For example:

```
var result = _pomQuery.FromType(Item.QueryRoot).SelectColumn(query =>
query.root.object_name)
    .UnionAll(pomQuery => pomQuery.FromType(ExcelTemplate.QueryRoot)
        .SelectColumn(query => query.root.item_id))
    .UnionAll (pomQuery2 => pomQuery2.FromType(ItemRevision.QueryRoot)
        .SelectColumn(query => query.root.item_revision_id))
    .ExecuteOnDatabase();
```

However, you may call Difference() only once. If you add a second Difference() call, you will get an runtime exception.

Querying with CTEs

Simple CTEs

Writing a CTE with a single column

To use a CTE, you must define it first, before writing the main query. The CTE query must select one or more columns. After the column have been selected you call AsCte() to create the CTE. The AsCte() method requires you to specify the types of the CTE's columns explicitly.

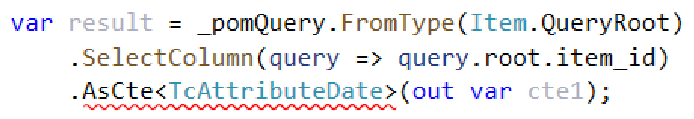
Here's an example of a CTE that has a single string column:

```
_pomQuery.FromType(Item.QueryRoot)
    .SelectColumn(query => query.root.item_id)
    .AsCte<TcAttributeString>(out var cte1)
    ...
```

Here are the column types for the AsCte call:

Attribute type	**Column type used in AsCte<>**
bool / logical	TcAttributePomLogical
char	TcAttributeChar
double	TcAttributeDouble
int	TcAttributeInteger
string	TcAttributeString
TcDate	TcAttrbuteDate
Tag	TcAttributeTag

If you get the type wrong, you will see a compiler error:



```
var result = _pomQuery.FromType(Item.QueryRoot)
    .SelectColumn(query => query.root.item_id)
    .AsCte<TcAttributeDate>(out var cte1);
```

The out variable, *cte1,* is used where we want to include the CTE into the main query.

Using the CTE

You can either use the CTE as the root of your main query, or you can use it in a join. In either case, you carry on writing the query in the same fluent statement, following on from .AsCte() with a second call to .FromType().

To use the CTE as the root type, call .FromType(cte1), like this

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .SelectColumn(query => query.root.item_id)
    .AsCte<TcAttributeString>(out var cte1)
    .FromType(cte1)
    ...
```

cte1 has a single string column, *col1*, and you can use col1 as you continue writing the query, as in this where clause:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .SelectColumn(query => query.root.item_id)
    .AsCte<TcAttributeString>(out var cte1)
    .FromType(cte1)
    .Where(query => query.root.col1.IsLike("A%"))
    ...
```

Alternatively, you can start your root query with another type and join the CTE in later, like this:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .SelectColumn(query => query.root.item_id)
    .AsCte<TcAttributeString>(out var cte1)
    .FromType(ItemRevision.QueryRoot)
    .Join1(cte1, query => query.root.item_revision_id == query.join1.col1)
    ...
```

And in either case, you continue writing the query as normal, for example, loading the object graph:

```
var result = _pomQuery.FromType(Item.QueryRoot)
    .SelectColumn(query => query.root.item_id)
    .AsCte<TcAttributeString>(out var cte1)
    .FromType(ItemRevision.QueryRoot)
    .Join1(cte1, query => query.root.item_revision_id == query.join1.col1)
    .LoadInstanceGraphFromDatabase();
```

If you join to a CTE and then call LoadInstanceGraphFromDatabase in this way, the resulting graph will miss out the CTE nodes.

Writing a CTE with multiple columns

If you need more columns in your CTE, select them and then define each column type in the AsCte() call. Here's an example of a CTE that has three columns:

```
_pomQuery.FromType(Item.QueryRoot)
    .SelectColumn(query => query.root.uid)
    .SelectColumn(query => query.root.item_id)
    .SelectColumn(query => query.root.object_name)
    .AsCte<TcTagAttribute, TcAttributeString, TcAttributeString>(out var cte1)
    ...
```

cte1 has three columns, *col1*, *col2* and *col3*, which you can use in the main query. For example:

```
var graph = _pomQuery.FromType(Item.QueryRoot)
    .SelectColumn(query => query.root.uid)
    .SelectColumn(query => query.root.item_id)
    .SelectColumn(query => query.root.object_name)
    .AsCte<TcTagAttribute, TcAttributeString, TcAttributeString>(out var cte1)
    .FromType(Item.QueryRoot)
    .Join1(cte1, query => query.root.uid == query.join1.col1)
    .Join2(ItemRevision.QueryRoot, query => query.join1.col1 == query.join2.items_tag)
    .Where(query => query.join1.col2 == "myid" && query .join1.col3 == "myname")
    .LoadInstanceGraphFromDatabase();
```

Using multiple CTEs

You may define as many CTEs as you need for your query. Here's an example of a query that has two CTEs; both are used in the main query:

```
_pomQuery.FromType(Item.QueryRoot)
    .SelectColumn(query => query.root.item_id)
    .AsCte<TcAttributeString>(out var cte1)
    .FromType(ItemRevision.QueryRoot)
    .SelectColumn(query => query.root.item_revision_id)
    .AsCte<TcAttributeString>(out var cte2)
    .FromType(Item.QueryRoot)
    .Join1(cte1, query => query.root.item_id == query.join1.col1)
    .Join2(cte2, query => query.join1.col1 == query.join2.col1)
    .SelectColumn(query => query.join2.col1)
    .ExecuteOnDatabase();
```

Recursive CTEs

Recursives CTEs are written using a CTE that is UNION ALLed to itself, in the same way as in SQL.

Here's an example of a query that uses a recursive CTE to return the parent hierarchy of the POM group "group1"

```
var result = _pomQuery.FromType(Pom_group.QueryRoot)
    .Where(query => query.root.name == "group1")
    .SelectColumn(query => query.root.uid)
    .SelectColumn(query => query.root.name)
    .SelectColumn(query => query.root.parent)
    .SelectColumn(_ => SelectedLiteral.Value(1))
    .AsCte<TcAttributeTag, TcAttributeString, TcAttributeTag, TcAttributeInteger>(out var
cte1)
    .UnionAll(q => q.FromType(Pom_group.QueryRoot)
        .Join1(cte1, query => query.root.parent == query.join1.col1)
        .SelectColumn(query => query.root.uid)
        .SelectColumn(query => query.root.name)
        .SelectColumn(query => query.root.parent)
        .SelectColumn(query => query.join1.col4 + 1))
    .FromType(cte1)
    .SelectColumn(query => query.root.col2)
    .SelectColumn(query => query.root.col3)
    .SelectColumn(query => query.root.col4)
    .OrderBy(query => query.root.col4)



    .ExecuteOnDatabase();
```

Running the same query more than once

In all the examples above, we've created the query and executed it in a single statement. Each time the statement is executed, a new query is created, executed and then deleted. But what if you want to create the query just once and execute it many times?

The answer is to call ToReusableQuery() in place of ExecuteOnDatabase(), LoadInstancesFromDatabase() or LoadInstanceGraphFromDatabase(). You can later call these methods on the reusable query object returned as many times as you wish.

Here's an example of a SOA service that uses this approach:

```
[SoaService(Name = "newtest", Namespace = "test.newtest.service")]
public class MySoaService
{
    private readonly IReusableValuesQuery<ValueTuple<string>> _query1;

    public MySoaService(IPomQuery pomQuery)
    {
        _query1 = pomQuery.FromType(ExcelTemplate.QueryRoot)
            .SelectColumn(query1 => query1.root.item_id)
            .ToResuableQuery();
    }

    [SoaServiceMethod(Name = "op")]
    public object Op()
    {
        var ids = _query1.ExecuteOnDatabase();
        return ids.Select(id => id.Item1);
    }
}
```

The executable query object implements IDispose; you should call its Dispose() method if at any point you no longer need the query[17].

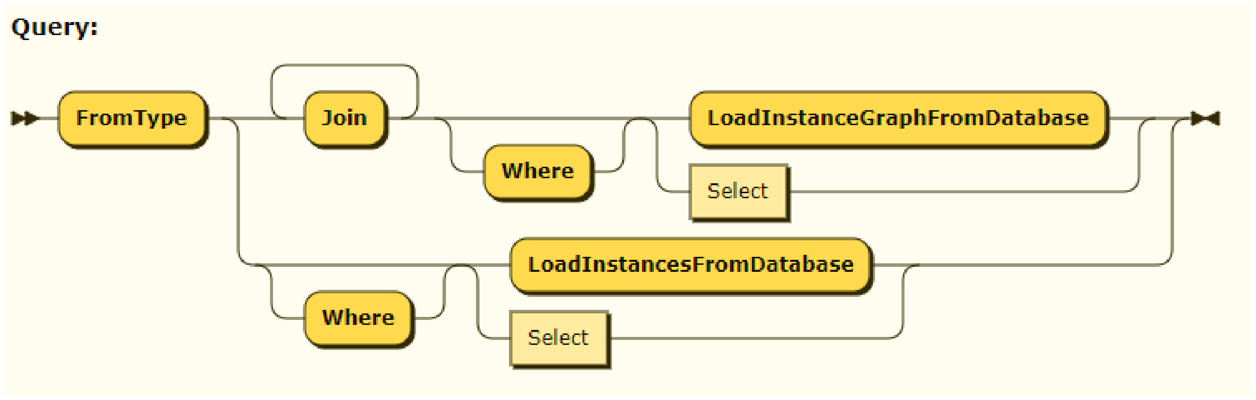
Note that the query isn't actually created until the first time it is executed[18].

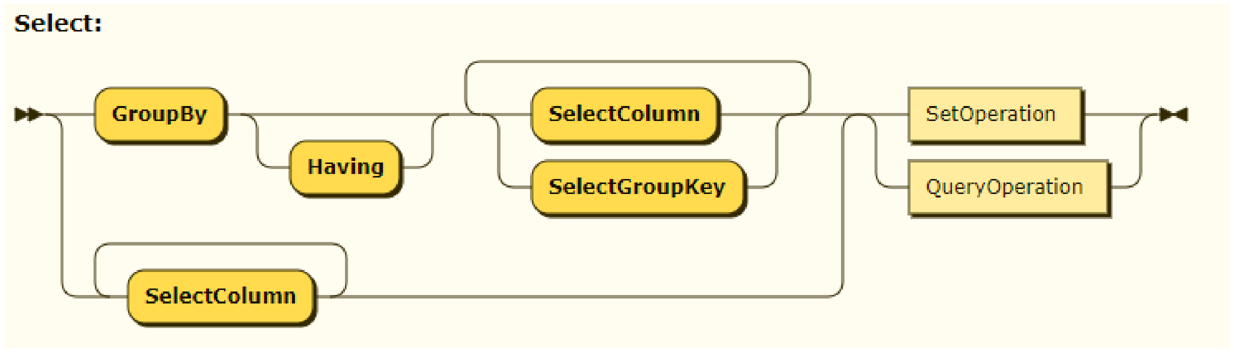
[17] There's no need to call Dispose() when the server is shutting down.
[18] The upshot is that it's safe to create the query in a class constructor, since the statement doesn't throw.

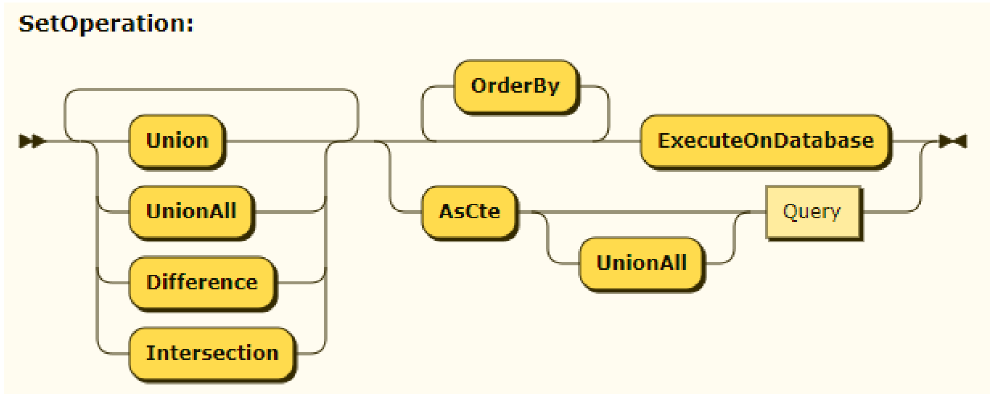
API Overview

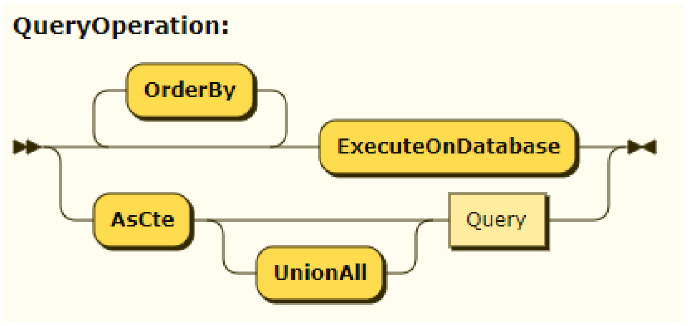
IPomQuery API that guides you through the process of writing a query in a structured way. For example, the API ensures that all your joins are done before the where clause. It effectively presents a query grammar.

The following railroad diagrams give an overview of the API's grammar. For the sake of clarity, it is somewhat simplified: for example, all the Join calls are grouped together, and it doesn't show the ToResusableQuery calls.









Limitations of queries

The API is designed to make it easy to write queries that that execute correctly on the database. But it isn't perfect. Some queries, though they are correctly formed, will result in a runtime exception nonetheless – this is usually because of an existing limitation in POM Enquiry.

If you use the mock framework to validate your tests (see *How to write unit tests using mocks*) you will trap these errors in your unit tests and correct them before ever running them on a real system.

You may also notice certain limitations in the API grammar, for example you can't

- Request the length of a string expression (nor take a substring of it, trim it, nor change its case)
- Use a subquery in a Having expression
- Use literals in a Having expression
- Take the length of a string attribute key in a having expression
- Write a parameterized query

How to write unit tests using mocks

It is good practice to write units tests to validate your business logic in the absence of a running Tc server. Such tests execute quickly and reliably, and can be run from within your development environment. They can give you - and those who change the code after you - immediate feedback on errors as they introduce them. You are strongly encouraged to adopt this practice for your Tc customization code.

You can use any test framework (we use NUnit in the examples that follow) and use mocks provided by a suitable mocking framework if required. You will also need a suitable test runner (our preferred one is ReSharper's runner that is embedded in Visual Studio).

However, there are three aspects of mocking the customization API that are difficult or impossible to achieve with using a standard mocking framework

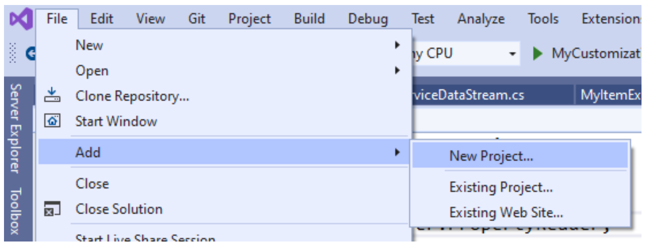
- Mocking fluent interfaces that use a number of different interface types to achieve a single outcome, such as IPomQuery
- Mocking two or more interfaces that semantically share common state, such as IPropertyReader, IBulkPropertyReader and IMetamodel
- Mocking interfaces that have methods defined internally, such as IServiceData

The Tc customization framework solves these issues by providing its own framework for mocking POM queries and Metamodel objects. These allow you to

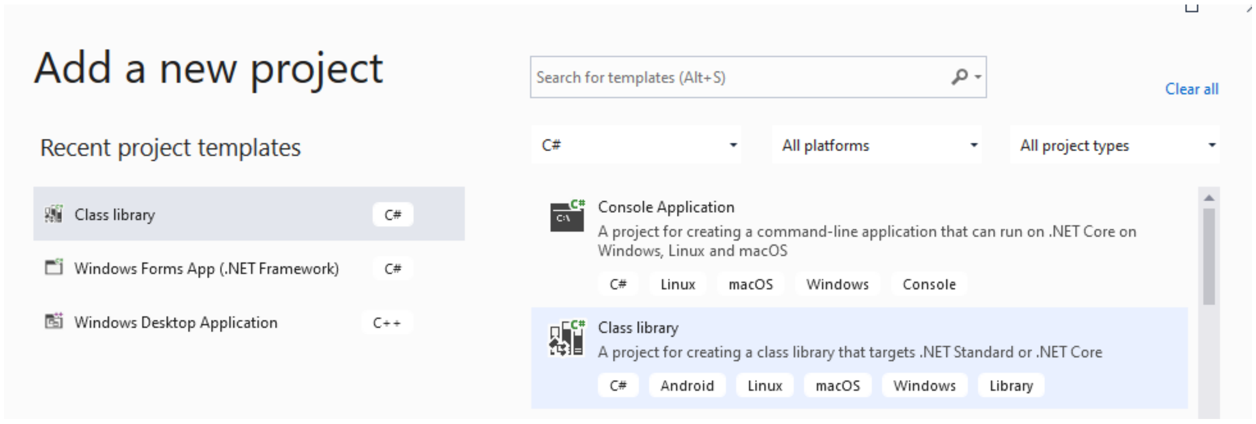
- Mock the entire collection of interfaces used by IPomQuery by mocking only the final outcome
- Register a set of mock Metamodel objecs that are used to serve up consistent set of results from IPropertyReader, IBulkPropertyReader<T>, IMetamodel, IRegisteredTcOperationDispatcher and IAomProp. It also allows you to validate the state of the mock objects at the end of a test.
- Use prepackaged mocked versions of interfaces that have methods defined with internal scope

How to set up an environment for writing unit tests

If you don't already have a separate test project set up, you will need to create one. In Visual Studio, select File / Add / New project…



choose C# class library for .NET Core



Add a new project

Search for templates (Alt+S)

Clear all

Recent project templates

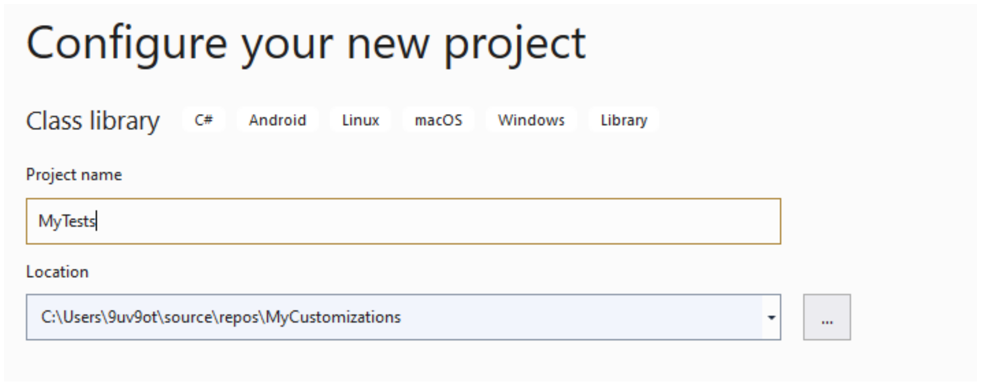
Class library	C#
Windows Forms App (.NET Framework)	C#
Windows Desktop Application	C++

C# · All platforms · All project types

Console Application
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS
C# Linux macOS Windows Console

Class library
A project for creating a class library that targets .NET Standard or .NET Core
C# Android Linux macOS Windows Library

give it a suitable name



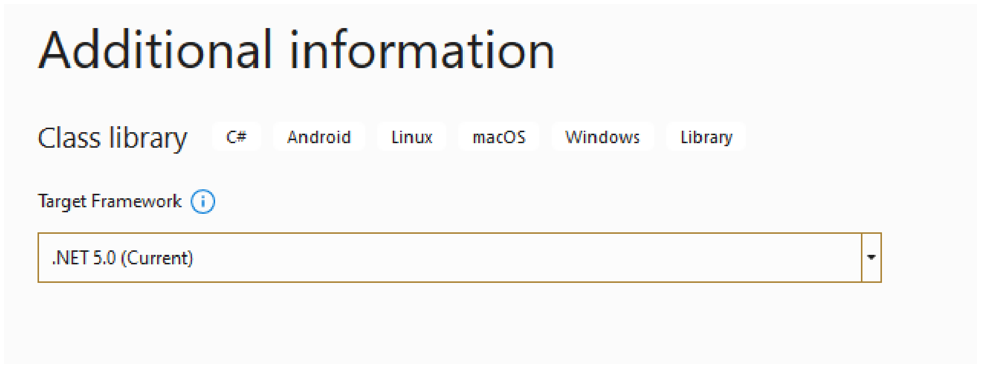
Configure your new project

Class library C# Android Linux macOS Windows Library

Project name: MyTests

Location: C:\Users\9uv9ot\source\repos\MyCustomizations

and set the .NET version to 5.0



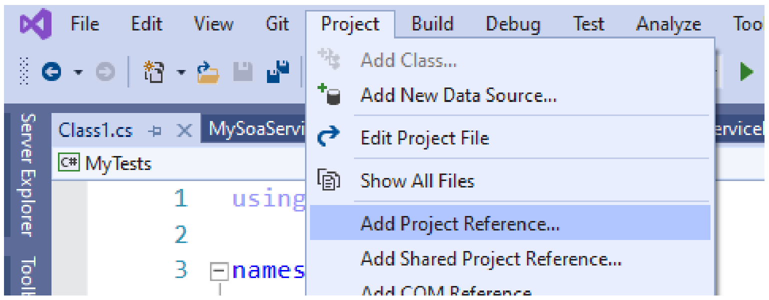
Additional information

Class library C# Android Linux macOS Windows Library

Target Framework: .NET 5.0 (Current)

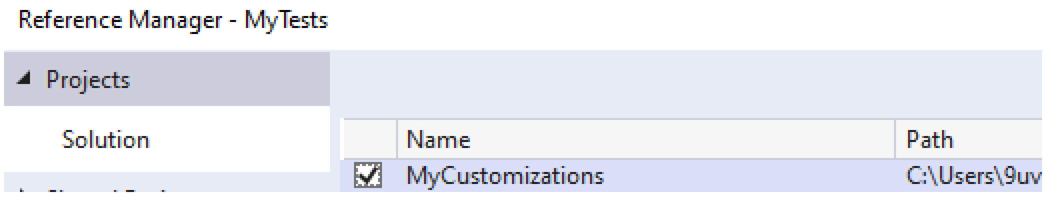
Next, add a reference to your customization project, so you can write test code that uses your customization classes.

Select File / Project / Add project reference…



```
using
names
```

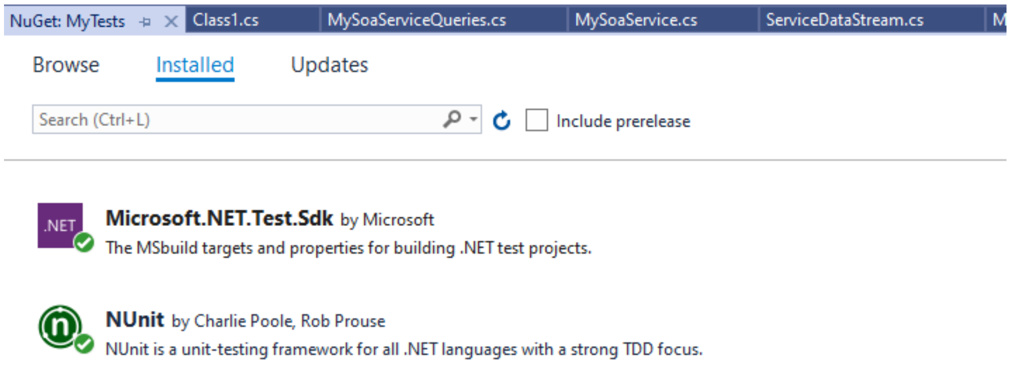
check the box for the customization project



	Name	Path
☑	MyCustomizations	C:\Users\9uv

and click OK.

Finally, use NuGet to include the test library you want to use. Here we add NUnit and also add Microsoft.NET.Test.Sdk, which is required to run .NET core tests.



Microsoft.NET.Test.Sdk by Microsoft
The MSbuild targets and properties for building .NET test projects.

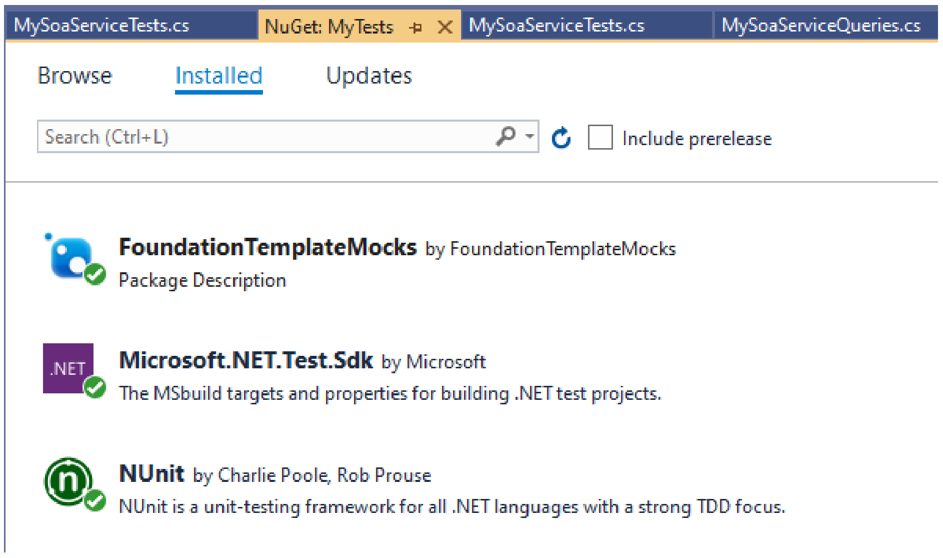
NUnit by Charlie Poole, Rob Prouse
NUnit is a unit-testing framework for all .NET languages with a strong TDD focus.

At this point, you can writing unit tests for your own classes.

A note of caution: only ever add these packages to your test project; never add them to the project with your production code.

How to set up an environment for writing tests with mocked Teamcenter services

To include the the Tc customization mocking framework, you will need to add the a mock assembly for each of the template assemblies you are using. In these examples we are only using FoundationTemplate, so we add FoundationTemplateMocks.



FoundationTemplateMocks by FoundationTemplateMocks
Package Description

Microsoft.NET.Test.Sdk by Microsoft
The MSbuild targets and properties for building .NET test projects.

NUnit by Charlie Poole, Rob Prouse
NUnit is a unit-testing framework for all .NET languages with a strong TDD focus.

At this point, the set up is complete.

How to write unit tests with mocked queries

To make the examples that follow more useful, I'm going to base them on a SOA service and a workflow handler that need testing. However, the approach shown here works equally well for workflow handler and Metamodel extensions, and so should be read by anyone interested in unit testing Tc customizations.

The examples don't necessarily show the best way to test the code in question, but they are designed to show good practice in using the mocking framework.

Here's the SOA service we're going to test. The developer has pulled out the query it uses into a separate class.

```
using System.Collections.Generic;
using System.Linq;
using FoundationTemplate;
using Teamcenter.TcServer.PropertyReader;
using Teamcenter.TcServer.Query;
using Teamcenter.TcServer.Soa;

namespace MyCustomizations
{
    public class MyMockedSoaQueries
    {
        // The resusable query for Item objects
        public IReusableInstancesQuery QueryItems { get; }

        public MyMockedSoaQueries(IPomQuery pomQuery)
        {
            QueryItems = pomQuery.FromType(Item.QueryRoot)
                .Where(query => query.root.item_id.In(
                    query.StartSubQuery.FromType(ItemRevision.QueryRoot)
```

```
                        .SelectColumn(subq => subq.root.item_revision_id)))
                .ToResuableQuery();
        }
    }


    [SoaService(Name = "mocked")]
    public class MyMockedSoaService
    {
        private readonly IBulkPropertyReader<ItemRuntimePropertySource> _propertyReader;
        private readonly IServiceDataFactory _serviceDataFactory;
        private readonly MyMockedSoaQueries _myMockedSoaQueries;

        public MyMockedSoaService(IPomQuery pomQuery, IBulkPropertyReaderFactory
propertyReaderFactory, IServiceDataFactory serviceDataFactory)
        {
            _propertyReader = propertyReaderFactory.For(Item.Properties);
            _serviceDataFactory = serviceDataFactory;
            _myMockedSoaQueries = new MyMockedSoaQueries(pomQuery);
        }

        [SoaServiceMethod(Name = "op")]
        public OpResponse ExampleOp()
        {
            // query for Item objects
            var tags = _myMockedSoaQueries.QueryItems
                .LoadInstancesFromDatabase()
                .ToList();
            // concatenate their names and item_ids
            var namesAndDescriptions = _propertyReader
                .AddItem1(item => item.object_name)
                .AddItem2(item => item.item_id)
                .Read(tags)
                .Select(NameAndId)
                .ToList();
            var serviceData = _serviceDataFactory.Create().AddResponse(tags);
            return new OpResponse(serviceData, namesAndDescriptions);
        }

        // return first ans second values joined with a "'"
        private static string NameAndId(TagAndValue<string, string> row)
        {
            var (_, name, id) = row;
            return string.IsNullOrEmpty(name) ? id ?? ""
                : string.IsNullOrEmpty(id) ? name
                : $"{name}.{id}";
        }

        // the response from operation op
        public class OpResponse
        {
            public IServiceData ServiceData { get; }
            public IReadOnlyList<string> NamesAndIds { get; }

            public OpResponse(IServiceData serviceData, IReadOnlyList<string>
namesAndIds)
            {
                ServiceData = serviceData;
                NamesAndIds = namesAndIds;
            }
        }
    }
}
```

You'll notice that the developer has pulled out the query the service uses into a separate class. They have also defined a specific class for the SOA operation's return type. These practices are both strongly recommended, particularly when writing unit tests.

Using TcServerMock to test a SOA operation

First, in your test project, create a new C# class to contain the tests and a test method. Let's start with a test that validates that, if no objects come back from the query, then NameAndId is empty[19]:

```
using NUnit.Framework;

namespace MyTests
{
    public class MyMockedSoaServiceTests
    {
        [Test]
        public void GivenQueryReturnsNoObjects_ExampleOpNameAndId_ShouldBeEmpty()
        {
        }
    }
}
```

We want to create a new instance of the service class, but to do so, we need to pass in an IPomQuery, an IBulkPropertyReaderFactory and an IServiceDataFactory. We're going to get these from an instance of TcServerMock.

We create a TcServerMock object using the static Create() factory method, use its properies to get hold of the mocked services, and pass them into the service class:

```
[Test]
public void GivenQueryReturnsNoObjects_ExampleOpServiceDataResponse_ShouldBeEmpty()
{
    var mock = TcServerMock.Create();
    var service = new MyMockedSoaService(mock.PomQuery, mock.BulkPropertyReaderFactory,
mock.ServiceDataFactory);
}
```

Now we can call ExampleOp() and validate the returned value:

```
[Test]
public void GivenQueryReturnsNoObjects_ExampleOpServiceDataResponse _ShouldBeEmpty()
{
    var mock = TcServerMock.Create();
```

[19] Please excuse the long tests names. Most people write shorter names, but I find these help me remember what the purpose of the test is.

```
    var service = new MyMockedSoaService(mock.PomQuery, mock.BulkPropertyReaderFactory,
mock.ServiceDataFactory);
    var result = service.ExampleOp();
    Assert.That(result.ServiceData.Response, Is.Empty);
}
```

The test runs and passes, because, by default, all mocked queries return empty result sets.

Setting up a mock query result

Mock results from LoadInstancesFromDatabase

But what if you want to write a test that validates what happens when the query returns some data? Simply set up the data you want the query to return before calling the operation under test.

You set up a query by defining it and setting its results. So, for our next test, we could write

```
var mock = TcServerMock.Create();
mock.PomQuery.FromType(Item.QueryRoot)
    .Where(query => query.root.item_id.In(
        query.StartSubQuery.FromType(ItemRevision.QueryRoot)
            .SelectColumn(subq => subq.root.item_revision_id)))
    .LoadInstancesFromDatabase()
    .ReturnsTags(1, 2, 3);
```

The call to *ReturnsTags* sets the result returned from any query with this definition; in this case we are saying that the query returns Tags 1, 2 and 3.

With this feature, we can now write the complete test:

```
[Test]
public void
GivenQueryReturnsThreeObjects_ExampleOpServiceDataResponse_ShouldHaveThreeEntries()
{
    var mock = TcServerMock.Create();
    mock.PomQuery.FromType(Item.QueryRoot)
        .Where(query => query.root.item_id.In(
            query.StartSubQuery.FromType(ItemRevision.QueryRoot)
                .SelectColumn(subq => subq.root.item_revision_id)))
        .LoadInstancesFromDatabase()
        .ReturnsTags(1, 2, 3);

    var service = new MyMockedSoaService(mock.PomQuery, mock.BulkPropertyReaderFactory,
mock.ServiceDataFactory);
    var result = service.ExampleOp();
    Assert.That(result.ServiceData.Response, Has.Count.EqualTo(3));
}
```

However, given that the developer has provided the test definitions in a separate class, we can take advantage of this and use their definition when setting up the mock to avoid duplicating the definition:

```
[Test]
public void
GivenQueryReturnsThreeObjects_ExampleOpServiceDataResponse_ShouldHaveThreeEntries()
{
    var mock = TcServerMock.Create();
    new MyMockedSoaQueries(mock.PomQuery).QueryItems
```

```
        .LoadInstancesFromDatabase()
        .ReturnsTags(1, 2, 3);

    var service = new MyMockedSoaService(mock.PomQuery, mock.BulkPropertyReaderFactory,
mock.ServiceDataFactory);
    var result = service.ExampleOp();
    Assert.That(result.ServiceData.Response, Has.Count.EqualTo(3));
}
```

The great advantage of this approach is that, if a developer later changes the definition of the query, the test will automatically adapt to use the new definition. Better still, if the query changes the type of the results it yields, the test will fail to compile until the developer modifies it to accept the new result type. For these reasons, you are strongly recommended to follow a similar pattern that makes the same query definitions available to both your test and your production code.

Note that instead of *ReturnsTags()* you can call *Returns()* and pass in actual Tags rather than raw numbers. The two methods do the same thing.

Mock results from ExecuteOnDatabase

If your query calls *ExecuteOnDatabase()* to return rows of values, you can mock it in the same way, passing the set of tuples to return when the query is executed, like this:

```
mock.PomQuery.FromType(Item.QueryRoot)
    .SelectColumn(query => query.root.object_name)
    .SelectColumn(query => query.root.is_vi)
    .SelectColumn(query => query.root.last_mod_date)
    .ExecuteOnDatabase()
    .Returns(
        ("name1", null, firstJanuary2000),
        ("name2", true, null));
```

As you can see from the example, you can provide null values too.

Mock results from LoadInstanceGraphFromDatabase

Graph queries work in a similar way, but here you provide a formatted string that describes the entire graph to return, like this:

```
var mock = TcServerMock.Create();
mock.PomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .LoadInstanceGraphFromDatabase()
    .Returns("{1->[2,3,4]}");
```

Each object returned has a number that is the object’s Tag. If the object no children is described by its Tag alone (e.g. “2”, “3” and “4” in the example above). An object that has children is described between {} braces. Inside the braces are its tag, a “->” arrow, followed collections of siblings each of which is surrounded by surrounded by [] brackets and represents the children arising from one join.

If a node in the graph has two joins, you can have two such collections:

```
var mock = TcServerMock.Create();
mock.PomQuery.FromType(Item.QueryRoot)
```

```
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.root.to_owning_user)
    .LoadInstanceGraphFromDatabase()
    .Returns("{1->[2,3],[4,5,6]}");
```

with the one joined second appearing second in the graph string.

If a child node has its own children, the pattern between the braces is be repeated, like this:

```
mock.PomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .LoadInstanceGraphFromDatabase()
    .Returns("{1->[{2->[3,4]}]}");
```

Note that in the query results, a joined node need not have any children[20] in the results, so if object 2 has no children we would write:

```
var mock = TcServerMock.Create();
mock.PomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .LoadInstanceGraphFromDatabase()
    .Returns("{1->[2]}");
```

and if object 1 has none:

```
var mock = TcServerMock.Create();
mock.PomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .LoadInstanceGraphFromDatabase()
    .Returns("1");
```

All the examples here have shown a single root node, but can specify as many root nodes as you wish. This example shows two:

```
var mock = TcServerMock.Create();
mock.PomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.join1.to_owning_user)
    .LoadInstanceGraphFromDatabase()
    .Returns("{1->[2]},{3->[4,{5->[6,7]}]}");
```

To help you write graph strings, the mock framwork provides a method *GetExampleGraphStringForMock()*, which you can call to return an example graph string that has two loaded instances at each graph node. It may be helpful to call this method, write the result out like this:

```
var mock = TcServerMock.Create();
Console.WriteLine(mock.PomQuery.FromType(Item.QueryRoot)
    .Join1(query => query.root.to_itemrevision_items_tag)
    .Join2(query => query.root.to_owning_user)
    .LoadInstanceGraphFromDatabase()
```

[20] This is true even if you specify an inner join, since the children may have failed to load.

```
    .GetExampleGraphStringForMock());
```

and then edit the string to suit the purposes of your test.

The full grammar for the graph string is

```
graph ::= (node ("," node)*)?
node ::= tag | complex_node
tag ::= [1-9] ([0-9])*
complex_node ::= "{" tag "->" joined_nodes ("," joined_nodes)* "}"
joined_nodes ::= "[" graph "]"
```

You can play around with graph strings at the grammar's BNF playground site here.

Executing the mock query multiple times

During the scope of the TcServerMock object, each time the query is executed, it returns the registered mock result. If your code executes the same query multiple times and you want to specify different results on each execution, you can chain the results together, like this

```
var mock = TcServerMock.Create();
new MyMockedSoaQueries(mock.PomQuery).QueryItems
    .LoadInstancesFromDatabase()
    .ReturnsTags(1, 2, 3)       // result from the first execution
    .ReturnsTags(1, 2, 3, 4)    // result from the second execution
    .ReturnsTags();             // result from the third and all subsequent executions
```

How to write unit tests with mock Metamodel objects

The TcServerMock object mock also allows you to set up mock Metamodel objects that are used by the mock services, IMetamodel, IBulkPropertyReader<T>, IProperty, IAomProp and IRegisteredOperationDispatcher.

You create mock objects using the Builder classes defined in the mock template assembly. You create a Builder class by calling *WithTag* on the Mock object class, e.g.

```
var builder = MockItem.WithTag(1);
```

and create the mock object itself by calling *Create()*

```
var mockItem = MockItem.WithTag(1).Create();
```

Before calling *Create()*, you may set as many properties as you wish:

```
var mockItem = MockItem.WithTag(1).object_name("name1").item_id("id1").Create();
```

Those you don't set explicitly are assigned the default value for the C# property type (0, false, null etc).

Finally, register the objects with the mock by calling RegisterObjects(). Here's an example that registers three Item objects:

```
var mock = TcServerMock.Create()
    .RegisterObjects(
        MockItem.WithTag(1).object_name("name1").item_id("id1").Create(),
        MockItem.WithTag(2).object_name("name2").item_id("id2").Create(),
        MockItem.WithTag(3).object_name("name3").item_id("id3").Create());
```

As an example, here's a test that uses them to validate the output from the service operation:

```
[Test]
public void
GivenOneObjectTagReturnedFromQuery_ExampleOpNamesAndIds_ShouldContainTheObjectNameAndId()
{
    // set up objects with tags 1, 2, and 3
    var mock = TcServerMock.Create()
        .RegisterObjects(
            MockItem.WithTag(1).object_name("name1").item_id("id1").Create(),
            MockItem.WithTag(2).object_name("name2").item_id("id2").Create(),
            MockItem.WithTag(3).object_name("name3").item_id("id3").Create());
    // if the query returns tag 2
    new MyMockedSoaQueries(mock.PomQuery)
        .QueryItems
        .LoadInstancesFromDatabase()
        .ReturnsTags(2);

    // when we call ExampleOp()
    var soaService = new MyMockedSoaService(mock.PomQuery,
mock.BulkPropertyReaderFactory, mock.ServiceDataFactory);
    var result = soaService.ExampleOp();

    // it should return the concatenation of name and id from object with Tag 2
    var nameAndId = result.NamesAndIds.FirstOrDefault();
    Assert.That(nameAndId, Is.EqualTo("name2.id2").NoClip);
}
```

Once you've set up your mock objects, they are used by IPropertyReader, IBulkPropertyReader, IMetamodel *and* IAomProp. If you reimplement your service, for example, replacing IMetamodel with IBulkPropertyReader, you can leave your test set ups unchanged.

Using TcServer mock to test a workflow handler

This approach works equally well when it comes to testing workflow handlers, as we will see in the following examples. These examples are all based on the workflow action handler we developed in *How to write a custom Workflow Handler*. Here's the handler code that the examples will be testing:

```
using System;
using System.Linq;
using FoundationTemplate;
using Teamcenter.TcServer.Itk.Epm;
using Teamcenter.TcServer.Metamodel;
using Teamcenter.TcServer.Workflow;

namespace MyCustomizations
{
    [WorkflowExtension]
    public class MyWorkflowHandlers
    {
        private readonly IEpmTask _epmTaskInterface;

        public MyWorkflowHandlers(IMetamodel metamodel)
        {
            _epmTaskInterface = EpmTask.Interface(metamodel);
        }

        [ActionHandler("myhandler", "used in example code")]
        public void MyActionHandler(EpmActionMessage actionMessage)
        {
            if (actionMessage.Task.HasNullValue) return;

            // find the task's start date
            var startDate = _epmTaskInterface.get_fnd0StartDate(new[] {
actionMessage.Task }).First()?.ToDateTime();

            // if it was started more than seven days ago, set its priority to 1
            if (startDate is not null && DateTime.UtcNow - startDate >
TimeSpan.FromDays(7))
                _epmTaskInterface.set_priority(actionMessage.Task, 1);
        }
    }
}
```

To write a test on the handle, we create a new class in the test project and add a test method. We'll start with a test that validates that, if the Task tag in the action message is null, the handler doesn't throw.

Here's the test skeleton:

```
using MyCustomizations;
using NUnit.Framework;
using Teamcenter.TcServer.Mock;

namespace MyTests
{
    public class MyWorkflowHandlerTests
    {
        [Test]
        public void GivenANullTaskTag_MyActionHandler_ShouldNotThrow()
        {
        }
    }
}
```

We create a TcServerMock object in the way we've already seen, and use it to provide services that are passed to the workflow handler class constructor:

```
[Test]
public void GivenANullTaskTag_MyActionHandler_ShouldNotThrow()
{
    var mock = TcServerMock.Create();
    var handler = new MyWorkflowHandlers(mock.Metamodel);
}
```

Then we can create the EpmActionMessage and pass it into the handler itself.

```
[Test]
public void GivenANullTaskTag_MyActionHandler_ShouldNotThrow()
{
    var mock = TcServerMock.Create();
    var handler = new MyWorkflowHandlers(mock.Metamodel);
    var epmActionMessage = new EpmActionMessage();
    Assert.DoesNotThrow(() => { handler.MyActionHandler(epmActionMessage); });
}
```

Setting up an EpmActionMessage

In the test above, we relied on using a default EpmActionMessage that set the handler's Task to Tag.NullValue. To a set our own Tag value, we simply use the EpmActionHandler's initializer:

```
var epmActionMessage = new EpmActionMessage { Task = Tag.CreateFromRaw(3) };
```

We can use this to write a test that validates what happens if the task's start date is null:

```
[Test]
public void GivenATaskWithANullStartDate_MyActionHandler_ShouldNotSetThrow()
{
    var task = MockEpmTask.WithTag(1).Create();
    var mock = TcServerMock.Create().RegisterObjects(task);
    var handler = new MyWorkflowHandlers(mock.Metamodel);
    var message = new EpmActionMessage { Task = Tag.CreateFromRawValue(3) };
    Assert.DoesNotThrow(() => { handler.MyActionHandler(message); });
}
```

You can set other values in the EpmActionMessage to suit the needs of your test, for example

```
var message = new EpmActionMessage
{
    Task = Tag.CreateFromRawValue(3),
    Action = EpmAction.Fail,
    Arguments = new EpmHandlerArg[] { new EpmHandlerStringArg("arg1", "value1")},
    ProposedState = EpmState.EpmFailed,
};
```

Setting up an EpmRuleMessage for a rule handler is done in exactly the same way.

Validating a property value at the end of a test

Next, let's write a test that validates that the handler sets the priority if its start date is over seven days ago. First set up a mock task object with an old start date:

```
[Test]
public void
GivenATaskWithAStartDateMoreThanSevenDaysAgo_MyActionHandler_ShouldSetThePriorityTo1()
{
    var dateTime = DateTime.UtcNow - TimeSpan.FromDays(7) - TimeSpan.FromSeconds(1);
    var over7DaysAgo = TcDate.CreateFrom(dateTime);

    // set up a task with priority 2 and start date over 7 days ago
    var task = MockEpmTask.WithTag(1).priority(2).fnd0StartDate(over7DaysAgo).Create();
    var mock = TcServerMock.Create().RegisterObjects(task);
}
```

Next call the handler, passing the task object in with the EpmActionMessage:

```
[Test]
public void
GivenATaskWithAStartDateMoreThanSevenDaysAgo_MyActionHandler_ShouldSetThePriorityTo1()
{
    var dateTime = DateTime.UtcNow - TimeSpan.FromDays(7) - TimeSpan.FromSeconds(1);
    var over7DaysAgo = TcDate.CreateFrom(dateTime);

    // set up a task with priority 2 and start date over 7 days ago
    var task = MockEpmTask.WithTag(1).priority(2).fnd0StartDate(over7DaysAgo).Create();
    var mock = TcServerMock.Create().RegisterObjects(task);

    // call the handler
    var handler = new MyWorkflowHandlers(mock.Metamodel);
    handler.MyActionHandler(new EpmActionMessage { Task = task.Tag });
}
```

And finally, validate the task's priority is now 1:

```
[Test]
public void
GivenATaskWithAStartDateMoreThanSevenDaysAgo_MyActionHandler_ShouldSetThePriorityTo1()
{
    var dateTime = DateTime.UtcNow - TimeSpan.FromDays(7) - TimeSpan.FromSeconds(1);
    var overSevenDaysAgo = TcDate.CreateFrom(dateTime);

    // set up a task with priority 2 and start date over 7 days ago
    var task =
MockEpmTask.WithTag(1).priority(2).fnd0StartDate(overSevenDaysAgo).Create();
    var mock = TcServerMock.Create().RegisterObjects(task);

    // call the handler
    var handler = new MyWorkflowHandlers(mock.Metamodel);
    handler.MyActionHandler(new EpmActionMessage { Task = task.Tag });

    // validate that the priority is now 1
    Assert.That(task.priority(), Is.EqualTo(1));
}
```

You can validate the value of any property on any mock object in this way. If the property is nullable, you can validate that it's null like this:

```
Assert.That(task.priority(), Is.Null);
```

Validating whether a property was set

What if we want to validate that the priority wasn't set at all? This is what we do in the following test which validates that, if the start date is under seven days old, the priority isn't set:

First we set up the test and call the handler:

```
[Test]
public void
GivenATaskWithAStartDateLessThan7DaysAgo_MyActionHandler_ShouldNotSetTheTaskPriority()
{
    var dateTime = DateTime.UtcNow - TimeSpan.FromDays(7) + TimeSpan.FromSeconds(1);
    var underSevenDaysAgo = TcDate.CreateFrom(dateTime);
    // set up a task with start date under 7 days ago
    var task = MockEpmTask.WithTag(1).fnd0StartDate(underSevenDaysAgo).Create();
    var mock = TcServerMock.Create().RegisterObjects(task);

    // call the handler
    var handler = new MyWorkflowHandlers(mock.Metamodel);
    handler.MyActionHandler(new EpmActionMessage { Task = task.Tag });
}
```

Then we validate the priority wasn't set, by calling *VerifyReceived(0)* on the mock object's property setter to verify it has been called zero times. VerifyReceived returns a Verified object which returns Verified.Ok if the assertion is true. So we can write:

```
[Test]
public void
GivenATaskWithAStartDateLessThan7DaysAgo_MyActionHandler_ShouldNotSetTheTaskPriority()
{
    var dateTime = DateTime.UtcNow - TimeSpan.FromDays(7) + TimeSpan.FromSeconds(1);
    var underSevenDaysAgo = TcDate.CreateFrom(dateTime);

    // set up a task with start date under 7 days ago
    var task = MockEpmTask.WithTag(1).fnd0StartDate(underSevenDaysAgo).Create();
    var mock = TcServerMock.Create().RegisterObjects(task);

    // call the handler
    var handler = new MyWorkflowHandlers(mock.Metamodel);
    handler.MyActionHandler(new EpmActionMessage { Task = task.Tag });

    // validate that the priorty hasn't been set to any value
    Assert.That(task.priority(Arg.Any<int?>()).VerifyReceived(0),
Is.EqualTo(Verified.Ok));
}
```

If the test fails, you will see something like this message

```
  Expected: <Ok>
  But was:  <Verification failed: Expected 0 calls to set property priority(<any int?>)
but found 1>
```

Of you course you can call *VerifyReceived* with any number of times, and any valid value, for example:

```
Assert.That(task.priority(3).VerifyReceived(1), Is.EqualTo(Verified.Ok));
```

which validates the property has been set to the value 3. But a word of caution: this doesn't validate that the priority ended up as 3, since it could have been set to a different value afterwards. If you want to validate the value of a property at the end of a call, do this directly, as shown in the earlier example.

Setting up an operation on a mock object

For this example, we're going to test a variation on the workflow rule handler we created earlier. It blocks all tasks that are running later (i.e. whose start date is more than 30 days ago) unless they are owned by the user admin.

Here's the rule handler code

```
[RuleHandler("myrulehandler", "used in example code")]
public EpmDecisionType MyRuleHandler(EpmRuleMessage ruleMessage)
{
    if (ruleMessage.Task.HasNullValue) return EpmDecisionType.Undecided;

    _epmTaskInterface.fnd0isOwningUser(ruleMessage.Task, "admin", out var
isOwnedByAdmin);

    // find the task's start date
    var startDate = _epmTaskInterface.get_fnd0StartDate(new[] { ruleMessage.Task
}).First()?.ToDateTime();
    var isRunningLate = startDate != null && DateTime.UtcNow - startDate >
TimeSpan.FromDays(30);

    // if it was started more than thirty days ago, block it, unless it is owned by admin
    var shouldBeBlocked = isRunningLate && !isOwnedByAdmin;

    return shouldBeBlocked ? EpmDecisionType.NoGo : EpmDecisionType.Go;
}
```

Here's our first attempt at the test, which sets up the object with a start date more than 30 days old:

```
[Test]
public void
GivenAnAdminTaskWithAStartDateMoreThan30DaysAgo_MyRuleHandler_ShouldReturnGo()
{
    var dateTime = DateTime.UtcNow - TimeSpan.FromDays(31);
    var over30DaysAgo = TcDate.CreateFrom(dateTime);

    // set up a task with a start date over 30 days ago
    var task = MockEpmTask.WithTag(1).fnd0StartDate(over30DaysAgo).Create();
    var mock = TcServerMock.Create().RegisterObjects(task);

    // call the handler
    var handler = new MyWorkflowHandlers(mock.Metamodel);
    var result = handler.MyRuleHandler(new EpmRuleMessage { Task = task.Tag });

    // validate that the task is allowed
    Assert.That(result, Is.EqualTo(EpmDecisionType.Go));
}
```

But this test fails because it hasn't set up the outcome of calling *fnd0isOwningUser*. Here the corrected test, with the operation set up included:

```
[Test]
public void
GivenAnAdminTaskWithAStartDateMoreThan30DaysAgo_MyRuleHandler_ShouldReturnGo()
{
    var dateTime = DateTime.UtcNow - TimeSpan.FromDays(31);
    var over30DaysAgo = TcDate.CreateFrom(dateTime);

    // set up a task with a start date over 30 days ago, whose owner is admin
    var task = MockEpmTask.WithTag(1).fnd0StartDate(over30DaysAgo).Create();
    task.fnd0isOwningUser("admin", Arg.Out(true)).Returns(0);
    var mock = TcServerMock.Create().RegisterObjects(task);

    // call the handler
    var handler = new MyWorkflowHandlers(mock.Metamodel);
    var result = handler.MyRuleHandler(new EpmRuleMessage { Task = task.Tag });

    // validate that the task is allowed
    Assert.That(result, Is.EqualTo(EpmDecisionType.Go));
}
```

The set up call, *task.fnd0isOwningUser*, is like the real call, except the *out* parameters is passed using *Arg.Out()*,allowing you to pass in the value that is passed out when the real call is made. So this call says, when *fnd0isOwningUser* is called on this object, passing in *admin*, set the result to *true* and return 0.

Of course you can use wildcard arguments in your set up, too, like this:

```
task.fnd0isOwningUser(Arg.Any<string>(), Arg.Out(true)).Returns(0);
```

Validating whether an operation was called

Sometimes, you're interested in whether an operation was called or not. To do this, you can use the same mock operation, but call *VerifyReceived()* instead of *Returns()*, .e.g.

```
// verify that there was one call to fndisOwningUser that received "admin"
Assert.That(task.fnd0isOwningUser("admin", Arg.Any<bool>()).VerifyReceived(1),
Is.EqualTo(Verified.Ok));

// verify that there was one call to fnd0isOwningUser that returned true
Assert.That(task.fnd0isOwningUser(Arg.Any<string>(), Arg.Out(true)).VerifyReceived(1),
Is.EqualTo(Verified.Ok));
```

Using TcServerMock to tests Metamodel extensions

The same approach we've used with SOA services and workflow handlers also work for Metamodel extensions: create the TcServerMock, use the services it provides to instantiate the MetamodelTypeExtension object, and call the method under test.

The mocking framework makes no attempt to simulate Metamodel behaviors: if you write a test for a runtime property, the runtime property code is run, but any extensions on the runtime property are omitted. This is a deliberate decision, allowing you to write unit tests for each unit of your business logic.

The framework provides a set of static functions that create implementations of IArrayPropertyValue<T> and IReadOnlyArrayPropertyValue<T>, which you can pass into your array property extensions.

The methods are TestArrayPropertyValue.Create() and TestReadOnlyArrayPropertyValue.Create(). You pass in the array element you want the array to have as its initial value.

For example, here's a test that creates an int array with the value { 1, 2, null } and validates that the getter postaction has removed the null element:

```
[Test]
public void
GivenAnArrayWithOneNullElement_AfterGetExampleProperty_ShouldReturnTheArrayWithoutTheNull
Element()
{
    var mock = TcServerMock.Create();
    var extension = new MyItemExtensions(mock.BulkPropertyReaderFactory);
    var value = TestArrayPropertyValue.Create(1, 2, null);
    extension.AfterGetExampleProperty(Tag.CreateFromRawValue(1), value);
    Assert.That(value, Is.EqualTo(TestArrayPropertyValue.Create(1, 2)));
}
```

Limitations of the mocking framework

It’s important to understand that tests built using the mocking framework are design to test your business logic. They don’t validate that your code integrates correctly with the Teamcenter environment. To do this, you must also test your code on a running Teamcenter server.

Rapid code-and-test

One of the features of Teamcenter server C# customizations is that ability to hot-deploy new custom code on a running Teamcenter server. This opens the possibility of writing code using very rapid code-test cycles, by modifying the code that is executing on the server, testing, modifying it again, and so on, all while the server is running.

See **TC_DOTNET_EXTENSION_SOURCE** env var in *How to set up the development environment***.**

When the above variable is set and file is saved in Visual Studio project, code will get automatically compiled and corresponding C# assemblies (you will see compilation error is the syslog, you have any syntax error in your code) will be deployed in running Teamcenter server. So, the process of “Rapid code-and-test” boils down to: Write code -> save->test -> Write code.

It’s important to grasp that what is being described here is a *development* methodology, not a testing strategy. It neither replaces nor competes with unit testing, TDD, integration testing or whatever CI/CD approach you currently use.

Basic Types

- **Basic types provided by the Tc customization framework**

The Tc customization code provides two basic types that are used throughout custom code

- Tag
  A C# representation of the ITK C type, tag_t
  Represents a Teamcenter Metamodel object.
- TcDate
  A C# representation of the ITK C type, date_t.

Those behave as primitive types

See *Localization and* Constants for Runtime Property

To generate default localization and constants for runtime property use TemplateGen utility.

Command For generating localization and constants file:

TemplateGen --action AssemblyToTemplate --output c:\workdir\metamodel --template <path of the installed templates> --assembly <path to customization assembly>

Description of the arguments:

--action AssemblyToTemplate indicates to generate deployable template from customization assembly.

-- output indicates folder for auto generated files

--template indicates the directory where all the installed templates reside

--assembly indicates the path of the customization assembly. Note that TemplateGen will generate a json file in the assembly location if there is no existing json file. The file will have the following entry. You can change the Guid and Prefix. Here “SoaCoreTestExtensions” is the assembly name where customization exists.

{"Name":"SoaCoreTestExtensions","DisplayName":"SoaCoreTestExtensions","Prefix":"SoaA","Guid":"00000000-0000-0000-0000-000000000000","Version":"tc2312.0000"}

The above TemplateGen command will generate 4 xml files:

<customization_assembly_name>_dependency.xml – dependency template file

<customization_assembly_name>_deployable_template.xml – which has the constant definition and can be deployed.

<customization_assembly_name>_template.xml – Which is a BMIDE loadable file, don’t deploy this

<customization_assembly_name>_template_en_US.xml – which has the localization and can be deployed

Help Contents for details.

- **Types generated from Teamcenter templates**

All the POM and Metamodel types that Teamcenter uses: Item, Part, etc., are defined in BMIDE templates. Each template is used to generate a C# assembly that contains C# definitions of the POM and Metamodel types.

In the examples that follow, we’ll use the Item type defined in Foundation template. Item has the following properties and methods. Item is defined in Teamcenter.TcServer.MetamodelWrappers.foundation namespace.

- Item.QueryRoot
  A C# property that contains definitions of the Item’s POM attributes. Used when building queries.
- Item.Properties
  A C# property that contains definitions of the Item’s Metamodel properties. Used when reading properties with an IBulkPropertyReader.
- Item.Interface()
  A C# method that returns an implementation of IItem, which gives access to the Item’s operations and its property getters and setters.

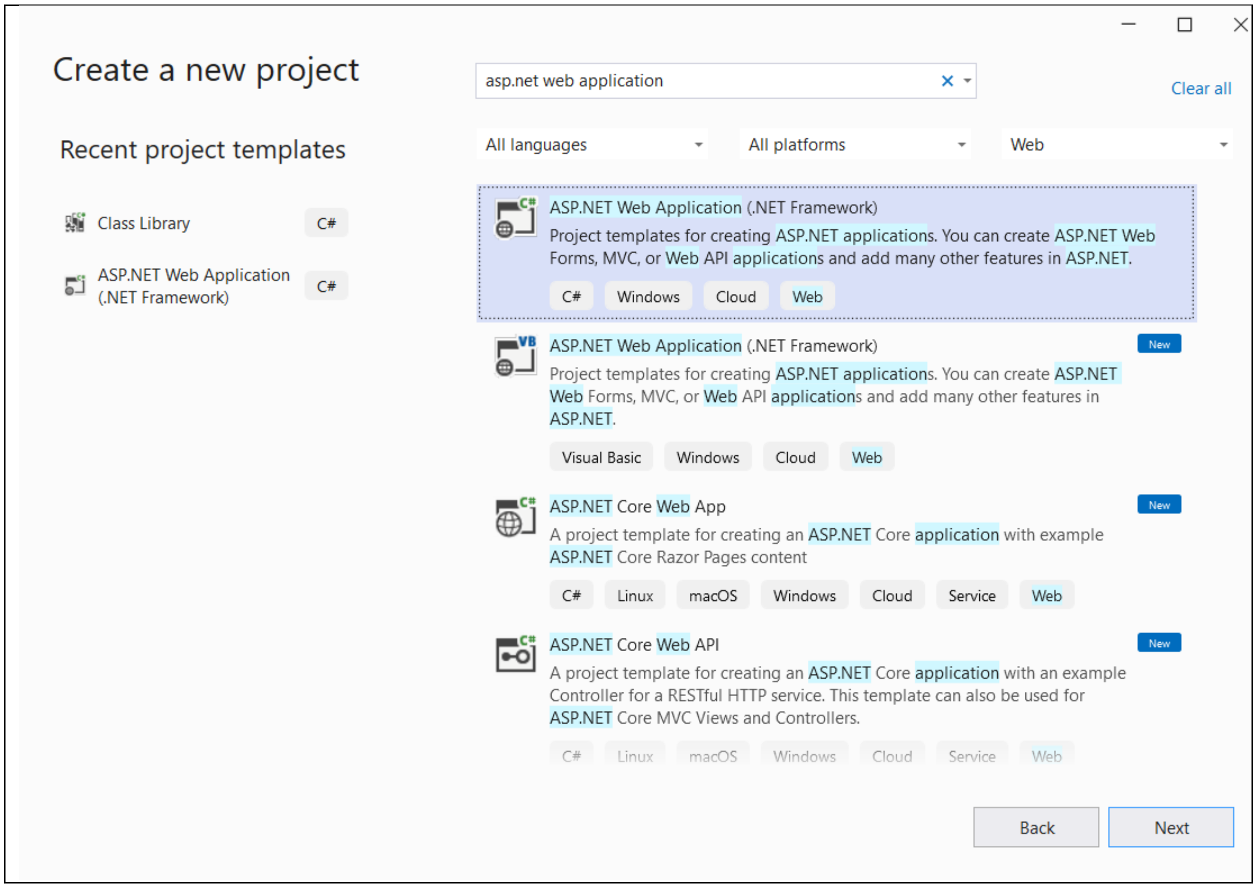
And each type generated follows the same pattern.

Setting up NuGet Server

Earlier setup used local dir to setup NuGet Packages. Recommended way is to set up a NuGet server service which is a mechanism through which developers can create, share, and consume useful code. Often such code is bundled into "packages" that contain compiled code (as DLLs) along with other content needed in the projects that consume these packages.

Setting up NuGet server consists of the following steps:

1. Create and deploy an ASP.NET Web application with NuGet.Server
    a) In Visual Studio, select File->New-Project and search for "ASP.NET Web Application (.NET Framework)", select the matching template for C#. If you don't see "ASP.NET Web Application (.NET Framework)", you need to install the "ASP.NET and web development" component.

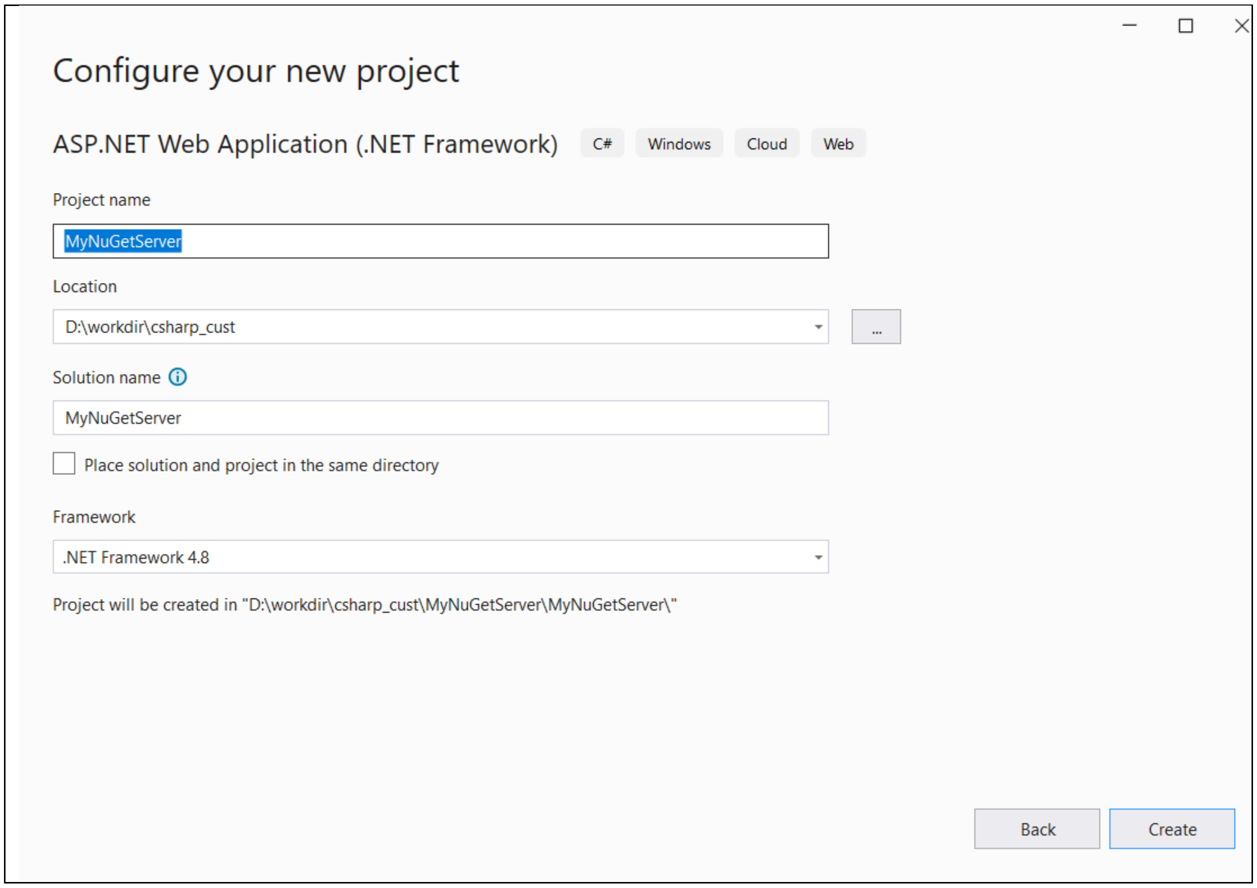


ASP.NET Web Application (.NET Framework)
Project templates for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.
C# Windows Cloud Web

ASP.NET Web Application (.NET Framework) New
Project templates for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.
Visual Basic Windows Cloud Web

ASP.NET Core Web App New
A project template for creating an ASP.NET Core application with example ASP.NET Core Razor Pages content
C# Linux macOS Windows Cloud Service Web

ASP.NET Core Web API New
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.
C# Linux macOS Windows Cloud Service Web



Configure your new project

ASP.NET Web Application (.NET Framework)

Project name

MyNuGetServer

Location

D:\workdir\csharp_cust

Solution name

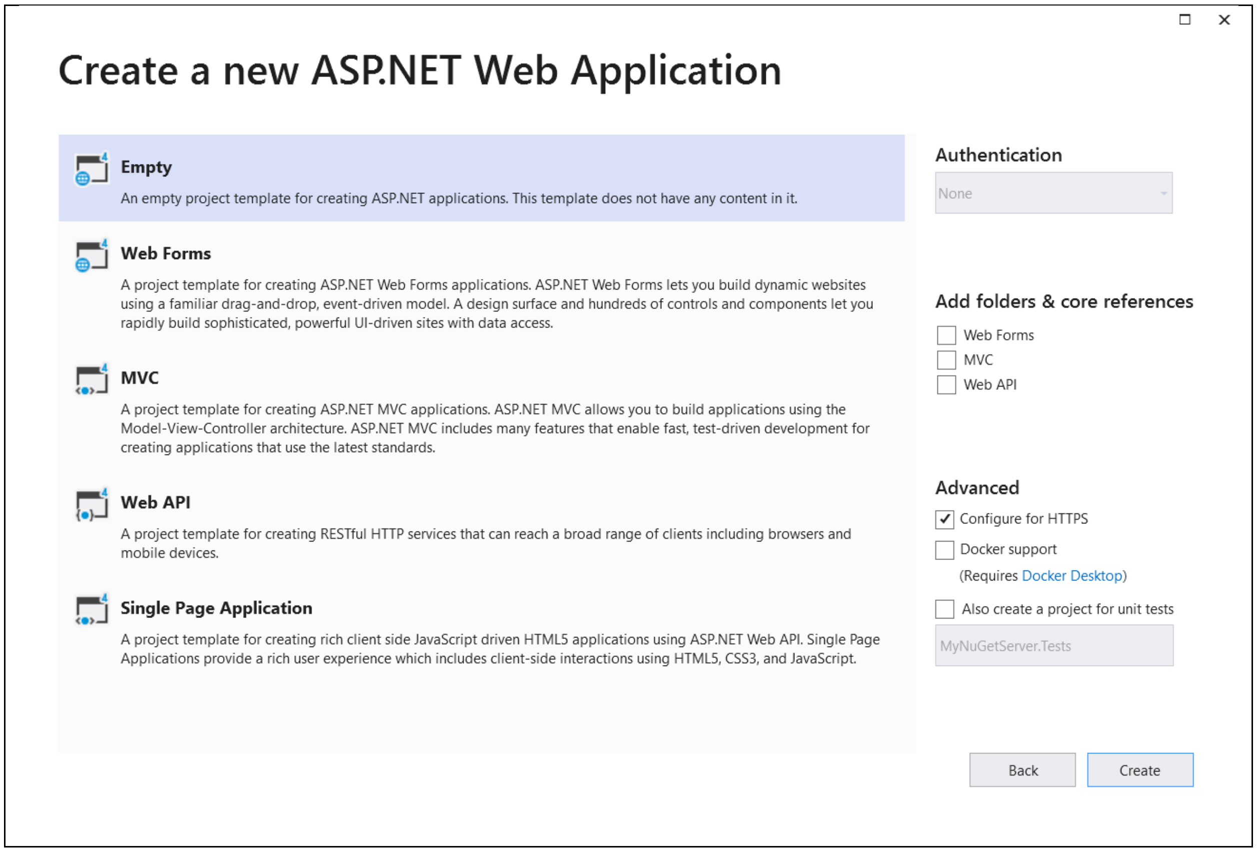
MyNuGetServer

Place solution and project in the same directory

Framework

.NET Framework 4.8

Project will be created in "D:\workdir\csharp_cust\MyNuGetServer\MyNuGetServer\"



Create a new ASP.NET Web Application

Empty
An empty project template for creating ASP.NET applications. This template does not have any content in it.

Web Forms
A project template for creating ASP.NET Web Forms applications. ASP.NET Web Forms lets you build dynamic websites using a familiar drag-and-drop, event-driven model. A design surface and hundreds of controls and components let you rapidly build sophisticated, powerful UI-driven sites with data access.

MVC
A project template for creating ASP.NET MVC applications. ASP.NET MVC allows you to build applications using the Model-View-Controller architecture. ASP.NET MVC includes many features that enable fast, test-driven development for creating applications that use the latest standards.

Web API
A project template for creating RESTful HTTP services that can reach a broad range of clients including browsers and mobile devices.

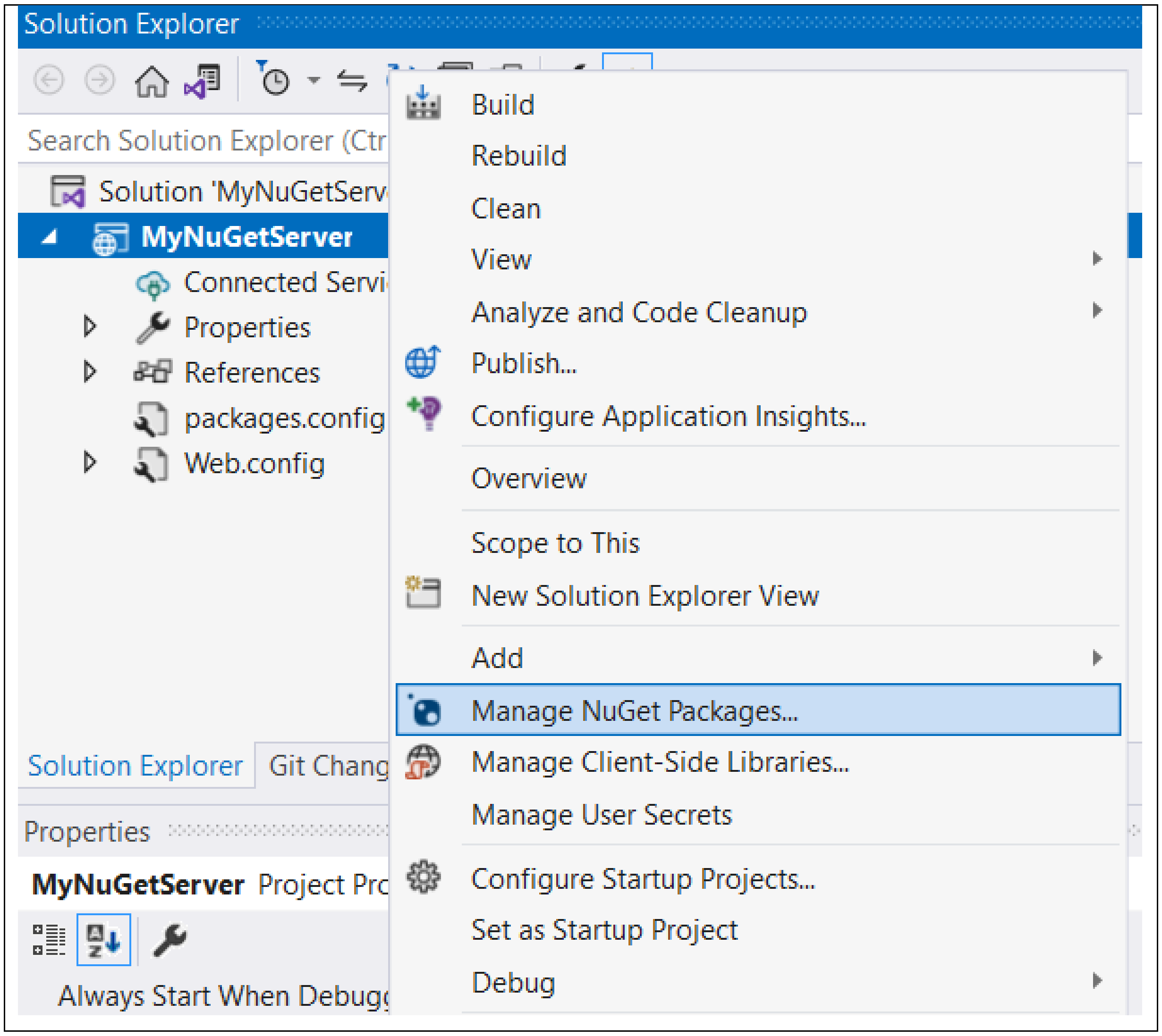
Single Page Application
A project template for creating rich client side JavaScript driven HTML5 applications using ASP.NET Web API. Single Page Applications provide a rich user experience which includes client-side interactions using HTML5, CSS3, and JavaScript.

Authentication
None

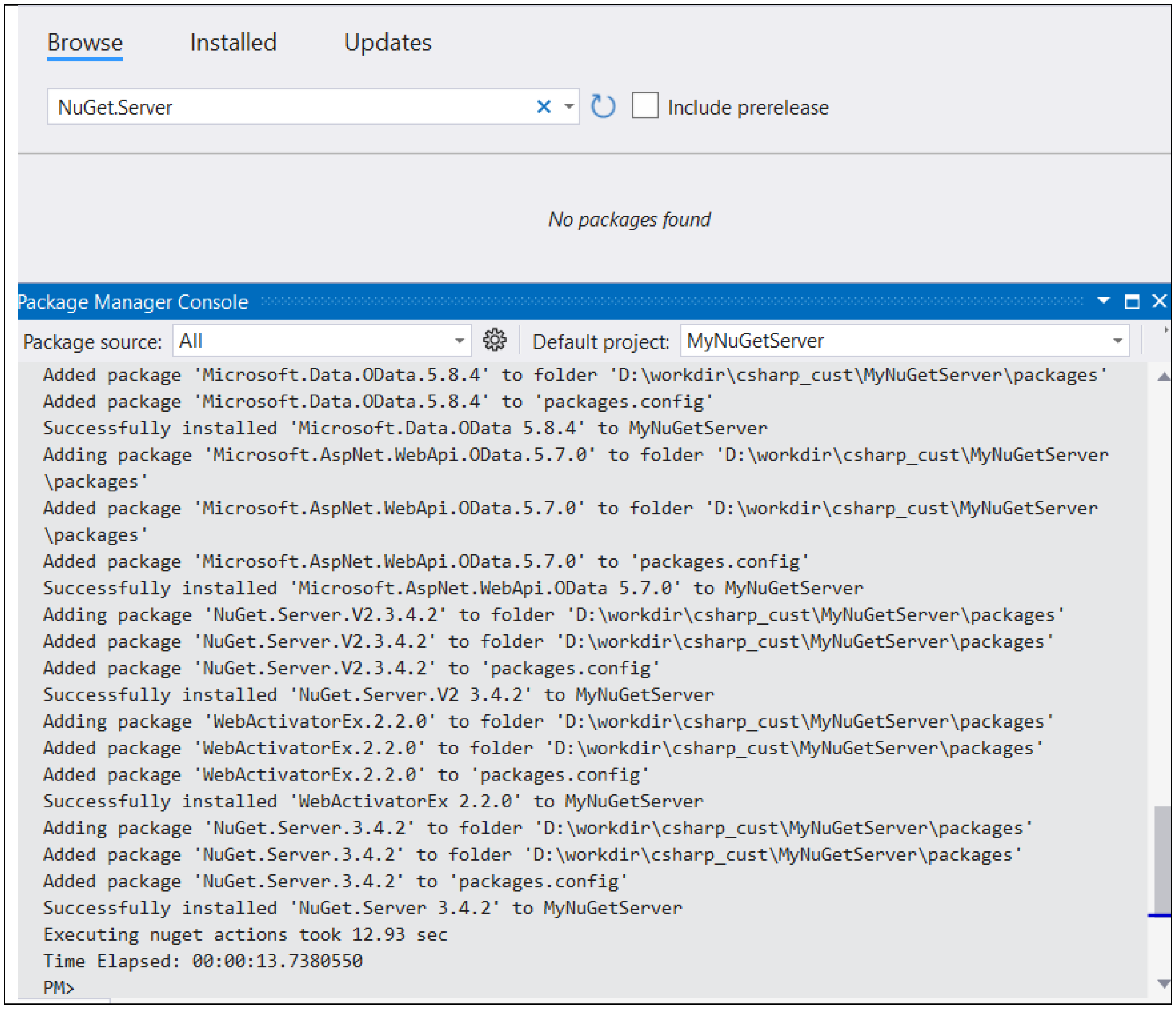
Add folders & core references
☐ Web Forms
☐ MVC
☐ Web API

Advanced
☑ Configure for HTTPS
☐ Docker support
(Requires Docker Desktop)
☐ Also create a project for unit tests
MyNuGetServer.Tests

b) Right click on the project, select “Manage Nuget Packages”

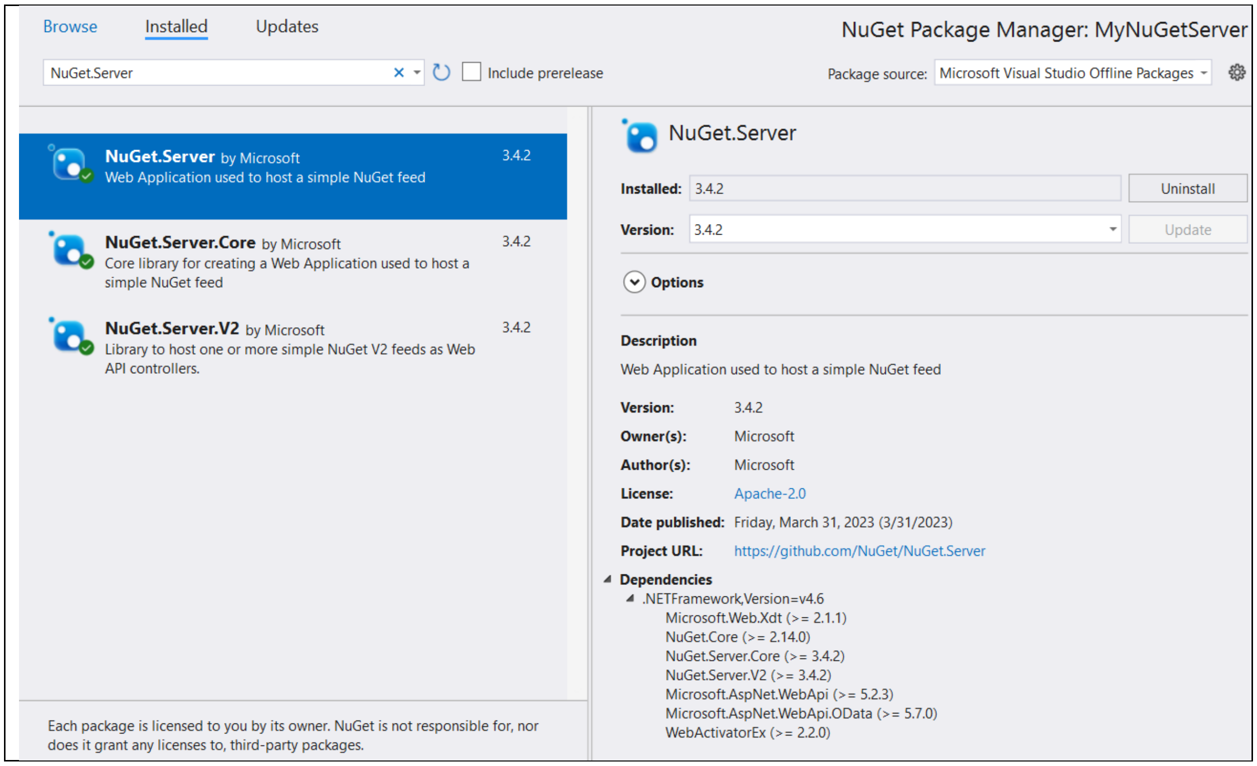


c) In the Package Manager UI, type “NuGet.Server” in search field and select the Browse tab. If search does not find any packages, install the latest version of the NuGet.Server package with Package Manager Console with “Install-Package NuGet.Server” command. Accept the license terms if prompted.



```
Added package 'Microsoft.Data.OData.5.8.4' to folder 'D:\workdir\csharp_cust\MyNuGetServer\packages'
Added package 'Microsoft.Data.OData.5.8.4' to 'packages.config'
Successfully installed 'Microsoft.Data.OData 5.8.4' to MyNuGetServer
Adding package 'Microsoft.AspNet.WebApi.OData.5.7.0' to folder 'D:\workdir\csharp_cust\MyNuGetServer
\packages'
Added package 'Microsoft.AspNet.WebApi.OData.5.7.0' to folder 'D:\workdir\csharp_cust\MyNuGetServer
\packages'
Added package 'Microsoft.AspNet.WebApi.OData.5.7.0' to 'packages.config'
Successfully installed 'Microsoft.AspNet.WebApi.OData 5.7.0' to MyNuGetServer
Adding package 'NuGet.Server.V2.3.4.2' to folder 'D:\workdir\csharp_cust\MyNuGetServer\packages'
Added package 'NuGet.Server.V2.3.4.2' to folder 'D:\workdir\csharp_cust\MyNuGetServer\packages'
Added package 'NuGet.Server.V2.3.4.2' to 'packages.config'
Successfully installed 'NuGet.Server.V2 3.4.2' to MyNuGetServer
Adding package 'WebActivatorEx.2.2.0' to folder 'D:\workdir\csharp_cust\MyNuGetServer\packages'
Added package 'WebActivatorEx.2.2.0' to folder 'D:\workdir\csharp_cust\MyNuGetServer\packages'
Added package 'WebActivatorEx.2.2.0' to 'packages.config'
Successfully installed 'WebActivatorEx 2.2.0' to MyNuGetServer
Adding package 'NuGet.Server.3.4.2' to folder 'D:\workdir\csharp_cust\MyNuGetServer\packages'
Added package 'NuGet.Server.3.4.2' to folder 'D:\workdir\csharp_cust\MyNuGetServer\packages'
Added package 'NuGet.Server.3.4.2' to 'packages.config'
Successfully installed 'NuGet.Server 3.4.2' to MyNuGetServer
Executing nuget actions took 12.93 sec
Time Elapsed: 00:00:13.7380550
PM>
```

d) Now NuGet.Sever will be available after install. Select "Installed" tab, and select NuGet.Server


Browse
Installed
Updates
NuGet Package Manager: MyNuGetServer
Include prerelease
Package source: Microsoft Visual Studio Offline Packages
Uninstall
Update
Options

NuGet.Server

NuGet.Server by Microsoft — 3.4.2
Web Application used to host a simple NuGet feed

NuGet.Server.Core by Microsoft — 3.4.2
Core library for creating a Web Application used to host a simple NuGet feed

NuGet.Server.V2 by Microsoft — 3.4.2
Library to host one or more simple NuGet V2 feeds as Web API controllers.

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

NuGet.Server

Installed: 3.4.2

Version: 3.4.2

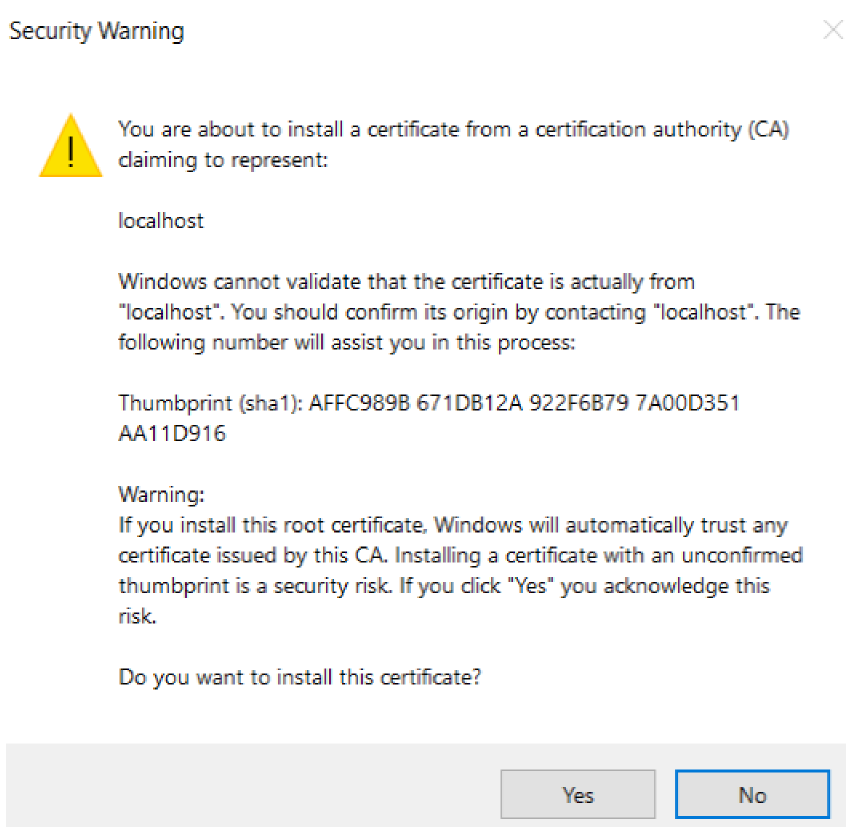
Description
Web Application used to host a simple NuGet feed

Version:	3.4.2
Owner(s):	Microsoft
Author(s):	Microsoft
License:	Apache-2.0
Date published:	Friday, March 31, 2023 (3/31/2023)
Project URL:	https://github.com/NuGet/NuGet.Server

Dependencies
.NETFramework,Version=v4.6
Microsoft.Web.Xdt (>= 2.1.1)
NuGet.Core (>= 2.14.0)
NuGet.Server.Core (>= 3.4.2)
NuGet.Server.V2 (>= 3.4.2)
Microsoft.AspNet.WebApi (>= 5.2.3)
Microsoft.AspNet.WebApi.OData (>= 5.7.0)
WebActivatorEx (>= 2.2.0)

e) Installing NuGet.Server converts the empty Web application into a package source. It installs a variety of other packages, creates a *Packages* folder in the application, and modifies *web.config* to include additional settings (see the comments in that file for details).

f) You might need to install a certificate from certification authority (CA) when pushing NuGet packages, such as deploying Scripted Business Logic Customization feature from Deployment Center (DC) or pushing your own NuGet packages. Press “Yes” if you see the following dialog (this is unrelated to Scripted Business Logic Customization)


Security Warning

You are about to install a certificate from a certification authority (CA) claiming to represent:

localhost

Windows cannot validate that the certificate is actually from "localhost". You should confirm its origin by contacting "localhost". The following number will assist you in this process:

Thumbprint (sha1): AFFC989B 671DB12A 922F6B79 7A00D351 AA11D916

Warning:
If you install this root certificate, Windows will automatically trust any certificate issued by this CA. Installing a certificate with an unconfirmed thumbprint is a security risk. If you click "Yes" you acknowledge this risk.

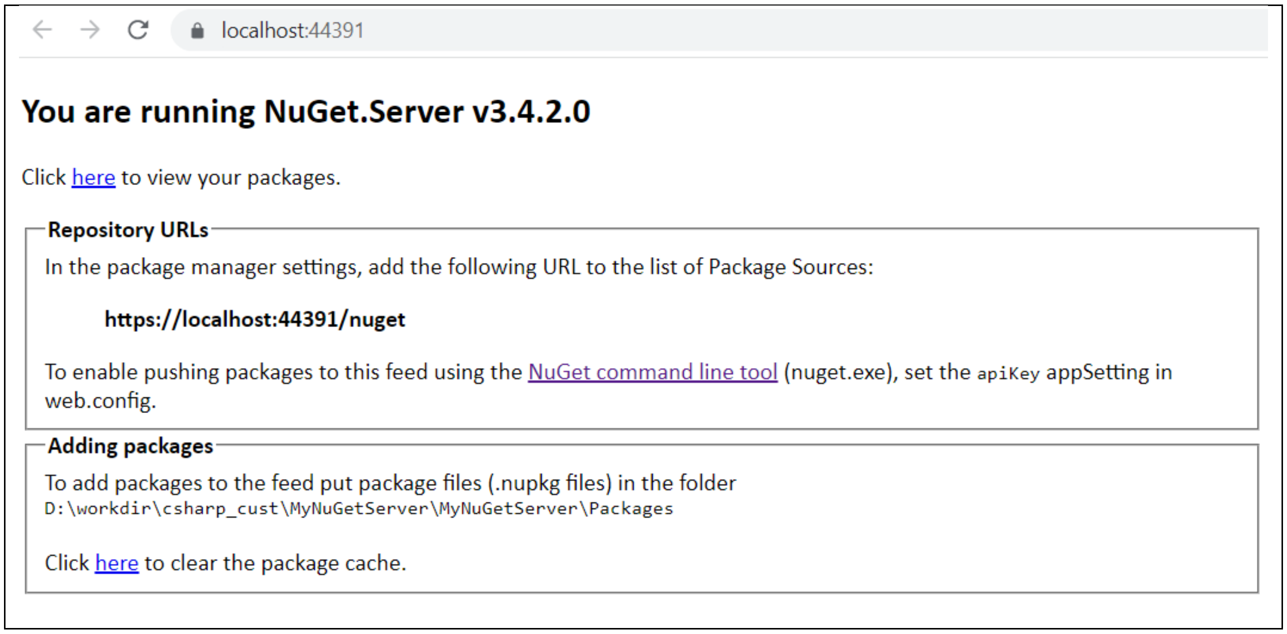
Do you want to install this certificate?

g) Note the following notice from Microsoft

ⓘ **Important**

Carefully inspect `web.config` after the NuGet.Server package has completed its modifications to that file. NuGet.Server may not overwrite existing elements but instead create duplicate elements. Those duplicates will cause an "Internal Server Error" when you later try to run the project. For example, if your `web.config` contains `<compilation debug="true" targetFramework="4.5.2" />` before installing NuGet.Server, the package doesn't overwrite it but inserts a second `<compilation debug="true" targetFramework="4.6" />`. In that case, delete the element with the older framework version.

h) Run the site locally in Visual Studio using Debug > Start Without Debugging or Ctrl+F5. The home page provides the package feed URLs as shown below. If you see errors, carefully inspect your *web.config* for duplicate elements as noted earlier.



You are running NuGet.Server v3.4.2.0

Click here to view your packages.

Repository URLs

In the package manager settings, add the following URL to the list of Package Sources:

https://localhost:44391/nuget

To enable pushing packages to this feed using the NuGet command line tool (nuget.exe), set the `apiKey` appSetting in web.config.

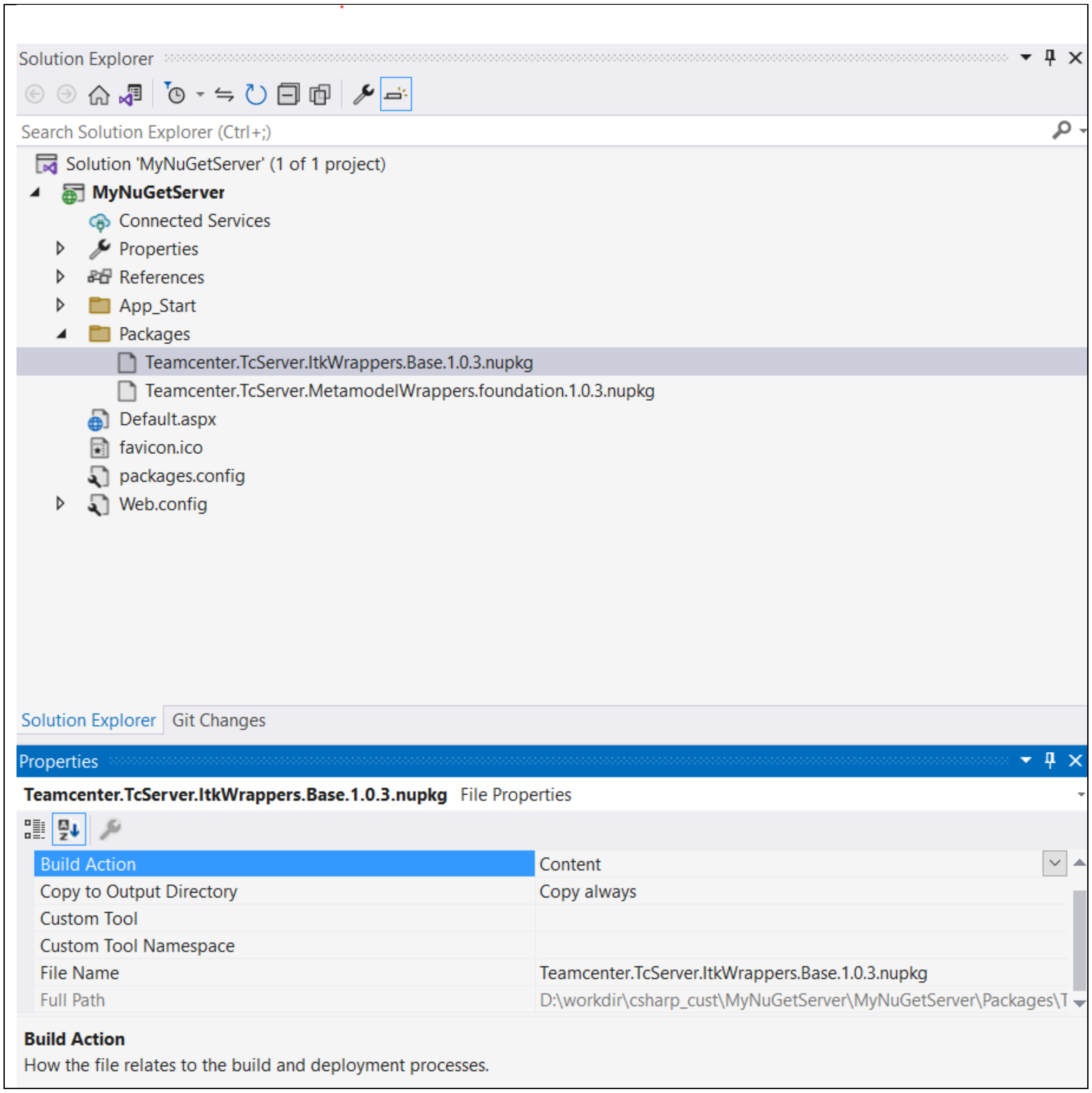
Adding packages

To add packages to the feed put package files (.nupkg files) in the folder
`D:\workdir\csharp_cust\MyNuGetServer\MyNuGetServer\Packages`

Click here to clear the package cache.

i) the URL that you use for the package source will be http://<domain>/NuGet.
You can set
TC_NUGET_FEEDS=**`http://<domain>/NuGet;`** https://api.NuGet.org/v3/index.json

2. Adding packages to the feed externally
Once a NuGet.Server site is running, you can add packages using NuGet push provided that you set an API key value in web.config. see https://learn.microsoft.com/en-us/NuGet/hosting-packages/NuGet-server for details
3. Making packages available when you publish the web app
To make packages available in the feed when you publish the application to a server, add each .nupkg files to the Packages folder in Visual Studio, then set each one's Build Action to Content and Copy to Output Directory to Copy always:


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'MyNuGetServer' (1 of 1 project)
MyNuGetServer
Connected Services
Properties
References
App_Start
Packages
Teamcenter.TcServer.ItkWrappers.Base.1.0.3.nupkg
Teamcenter.TcServer.MetamodelWrappers.foundation.1.0.3.nupkg
Default.aspx
favicon.ico
packages.config
Web.config
Solution Explorer
Git Changes
Properties
Teamcenter.TcServer.ItkWrappers.Base.1.0.3.nupkg File Properties

Build Action	Content
Copy to Output Directory	Copy always
Custom Tool	
Custom Tool Namespace	
File Name	Teamcenter.TcServer.ItkWrappers.Base.1.0.3.nupkg
Full Path	D:\workdir\csharp_cust\MyNuGetServer\MyNuGetServer\Packages\T

Build Action
How the file relates to the build and deployment processes.

For details: visit https://learn.microsoft.com/en-us/NuGet/hosting-packages/NuGet-server

Directory based setup

Although we recommend NuGet server based setup, for quick testing of the customization code directory based setup can be used. To use directory based setup, put your customization NuGet package path in TC_NUGET_FEEDS. Note that this variable has been discussed earlier in this document.

For example:

```
set TC_NUGET_FEEDS=%TC_BIN%\csharpcust\test\nugetrepo;https://api.nuget.org/v3/index.json;
```

In the above, it is assumed that customization NuGet package is in %TC_BIN%\csharpcust\test\nugetrepo folder

How to run TemplateGen.exe to generate metamodel wrapper for templates

Templategen.exe is a convenient utility that can be used to generate C# wrappers for any ITKs and metamodel operations. These generated wrappers provide a way to call ITKs/Metamodel operations from C# code. It can also be used to generate constants and localization for runtime properties

This utility can be found in %TC_BIN%/csharpcust

C# wrapper for all ITKs(Teamcenter.TcServer.ItkWrappers.Base.1.0.14.nupkg) and wrapper for foundation template (Teamcenter.TcServer.MetamodelWrappers.foundation.1.0.14.nupkg) are provided in the %TC_BIN%/csharpcust directory.

<u>Command For generating Metamodel wrapper from templates:</u>

<u>The following only needed if you use template(s) other than foundation template. For foundation template</u> Teamcenter.TcServer.MetamodelWrappers.foundation.1.0.14.nupkg is supplied with the Teamcenter Kit.

TemplateGen.exe --action TemplateToCSharp --version 1.0.14 --output c:\vssource\scratch --template <path of the installed templates> --NuGet c:\NuGet --writefiles

<u>Command For generating ITK wrapper :</u>

The following is only needed if you have custom ITKs for which C# ITK wrapper needed. If you only use ITKs from Teamcenter, Teamcenter.TcServer.ItkWrappers.Base.1.0.14.nupkg is supplied with the Teamcenter kit.

TemplateGen.exe --action DoxygenToCSharp --version 1.0.14 --output c:\vssource\scratch --NuGet c:\NuGet --writefiles –itksource <path where all Doxygen ITKs exists in xml format >

TemplateGen.exe /? Or –help or help will display the available options.

Description of the arguments:

--**action** indicates if Metamodel or ITK NuGet package should be generated. This option values can be TemplateToCSharp or DoxygenToCSharp

--action TemplateToCSharp

TemplateToCSharp action generates C# meta model definition from template.

For each template this option will generate a NuGet package.

--action DoxygenToCSharp

DoxygenToCSharp action generates ITK wrapper from Doxygen xml files.

.

--version indicates version of the NuGet Package, for Tc2312 use 1.0.14

-- output indicates folder for auto generated C# wrapper code and other build generated files. The output for this –action is in the form of NuGet package.

--template indicates the directory where all the installed templates reside, for --action "DoxygenToCSharp ", this option is not needed.

--NuGet indicates the folder where NuGet packages are being generated.

--writefiles indicates if we want auto generated C# wrapper files to be written out.

--itksource indicates the folder where Doxygen   xml files exists for the ITKs.. For –action "TemplateToCSharp", this option is not needed

See *How to set up the development environment* to see how to use these NuGet packages.

Using soacore_test.exe

This section assumes you have access to an installed Teamcenter test environment of some kind. If you can run your own test Teamcenter server against a test database, that will be sufficient.

Alongside tcserver.exe, the environment should include a number of test executables. The one we are going to use is soacore_test.exe, as it allows us to start up a complete Tc server-like executable that has a command-line interface that accepts SOA requests and writes out their responses. The executable supports all the Teamcenter customization features outlined in this document.

Setting up the environment

First, set the following environment variables

set TC_DOTNET_EXTENSION_MANIFEST=[21]
set TC_DOTNET_EXTENSION_SOURCE=<*path to directory containing C# customization code*>[22]
set TC_NUGET_FEEDS= https://api.NuGet.org/v3/index.json;<*path to local TC NuGet feed or URL of remote feed*>[23]

Running the executable

Next, start soacore_test with the console line command[24]:

```
soacore_test dotnet interactive
```

to which you may append the -u=*user_name* and -p=*password* switches for login.

You will see the output

```
>urun soacore_test dotnet interactive
In ITK_user_main...
Initial Exception
Testing dotnet interactive
>
```

Running SOA requests

At this prompt you can write SOA requests in the format

service-name operation-name {body}

where *{body}* is the json body that contains the operation's serialized parameters.

For example, if you have a service called newtest with an operation op that takes no parameters, you can call it like this:

```
> newtest op {}
```

or if it takes (int count, string message), you call it like this:

```
> newtest op {"count":1, "message": "hello"}
```

The SOA result is written on the output console, for example:

```
Got response with status 200 and headers
{
".QName":"test.newtest.service.object",
```

[21] Or you can set this to the name of a NuGet package that contains other custom code that you need during your test.
[22] The hot-deployed code must all be in this folder, or in its subfolders. If you have a .csproj file, it must be in this folder.
[23] This is the NuGet feed(s) that has previous customization NuGet Packages
[24] Use the same environment you would use for starting a test Teamcenter server instance.

```
"ids":["000003","000014","000021","000015","000022","000018","000016","000017
","000019","000020","GCS_CP_Socket","000004","GCS_CP_Plug","000001","000007",
"000005","000006","MDS_default_styles_template","000013","000002","000009","0
00010","000011","000012"],
"totalResults":24
}
```

While testing, you can add to the console output using by calling Console.WriteLine() from the C# custom code.

To terminate the program, enter and empty input at the prompt.

Running custom code

When the soacore_test starts, it looks in the directory pointed to by the environment variable TC_DOTNET_EXTENSION_SOURCE, looks for all the .cs files it finds, including in any subdirectories, compiles them into an assembly called DynamicExtensions.dll, and registers all the customizations the assembly contains with the Tc server.

If the directory includes .csproj file, it uses the file's NuGet and project dependencies to resolve any dependencies in the code it finds. If you write your custom code using Visual Studio or Visual Studio code, you will find it should load and run automatically.

Hot deploying code

You can update the code in TC_DOTNET_EXTENSION_SOURCE while the program is running[25]. Simply edit the code, and save it. Each time the .cs or .csproj files are changed on disk, a new version of the DynamicExtensions.dll assembly is compiled, and any changed customizations are updated.

Note that there are certain things you can't do. In general, you can't unregister things, so if you write a runtime property, say, and then rename it, the old property will still be there. You can, however, unregister SOA operations. Another important limitation is that you can't change the a runtime property's type. As a workaround, you can new property with a new name and type as a temporary measure until you can restart the program.

When things don't work

When things go wrong you may get enough information on the console to work out what has happened. But usually, the information you need to understand the issue is buried in the syslog. Using the syslog, consider the following

- Did the DynamicExtensions assembly compile successfully?
  If there are errors compiling the code, they are written to the syslog, and the old version of the code is left running.

- Has the compilation completed?

[25] This works in both tcserver.exe and soacore_test.exe.

There is a noticeable time lag of a second or two between saving the source file to disk and the code being deployed. Each time the assembly is deployed successfully, you will see a note in the syslog like this:

```
2022/05/17-17:04:14.040 UTC - Loaded dynamic library
C:\Users\9uv9ot\AppData\Local\Temp\dotnetdynamicextensions\DynamicExten
sions\0.0.3\DynamicExtensions.dll
```

The version (here, 0.0.3) is incremented on each compilation.

- Were the SOA services, Metamodel extensions or Workflow handler methods registered correctly? Any errors in the registration, for example an incorrect method signature – are reported clearly in the syslog, e.g. `2022/05/17-17:14:27.835 UTC - Extension method TemplateExtension.GetIsViPreCondition is ignored and will not be attached to property is_vi) on type Template. Method signature should be int (Tag objectTag), not void (Tag objectTag, bool value)`

  Each time a SOA service or workflow handler is created or updated, this is confirmed by an entry in the syslog that starts, `2022/05/17-16:40:14.208 UTC - Successfully registered` … followed by a description of what has been registered. Metamodel extensions are also logged in this way, but you don't see a separated log entry when an existing extension is updated.

- Was there an error running the SOA request?
  Look for the string *service-name:operation-name* in the log file, as this will be found at the start and the end of the operation. Somewhere between these two, you should find the cause of the error.

In the event of an error, you will probably see the follwing in the SOA response: "Teamcenter has detected a serious internal error; To maintain data integrity log out and restart Teamcenter, otherwise data corruption could occur." Assuming this is a test system using a test database, you can ignore this advice and continue. The process will nearly always continue to run without any problems.

Debugging

You can attach a Visual Studio or Visual Studio Code debugger to a running tcserver.exe or soacore_test instance.

Attach the debugger for Managed (.NET Core, .NET 6+) code. You can set breakpoints in your C# code, and step through as it executes.

Namespace Consideration

When defining SOA service, runtime property, workflow handlers and customizing metamodel BO, use same prefix as in your BMIDE template to eliminate name collision with other templates.

For property name, prefix should be added to the name of the property.

NOTE: Null or empty prefix could cause property name collision. In this example “**acm0**” is the prefix used for a property named **“acm0**runtime_string_with_precondition”. You should use your company specific BMIDE prefix.

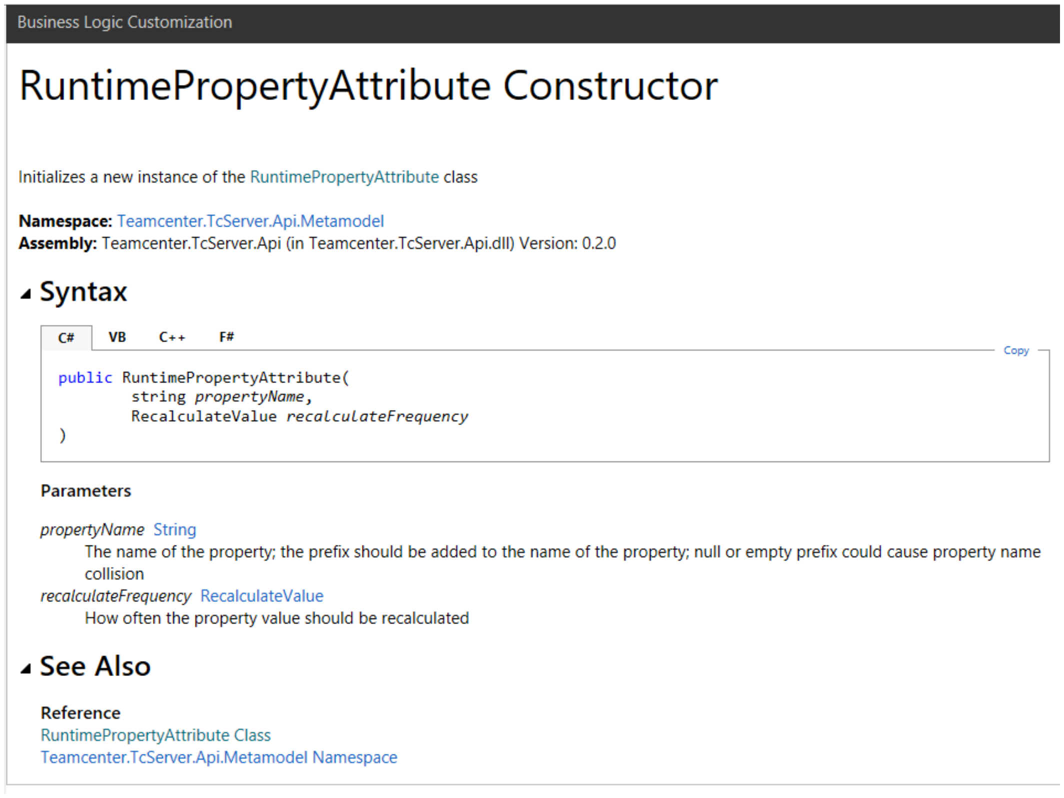
[RuntimeProperty("**acm0**runtime_string_with_precondition", RecalculateValue.OnEachRequest)]

public void RuntimeStringWithPrecondition(IEnumerable<Tag> objectTags, out IReadOnlyCollection<(Tag, string)> values)

{

………………

}



Business Logic Customization

RuntimePropertyAttribute Constructor

Initializes a new instance of the RuntimePropertyAttribute class

Namespace: Teamcenter.TcServer.Api.Metamodel
Assembly: Teamcenter.TcServer.Api (in Teamcenter.TcServer.Api.dll) Version: 0.2.0

Syntax

```
public RuntimePropertyAttribute(
	string propertyName,
	RecalculateValue recalculateFrequency
)
```

Parameters

propertyName String
The name of the property; the prefix should be added to the name of the property; null or empty prefix could cause property name collision
recalculateFrequency RecalculateValue
How often the property value should be recalculated

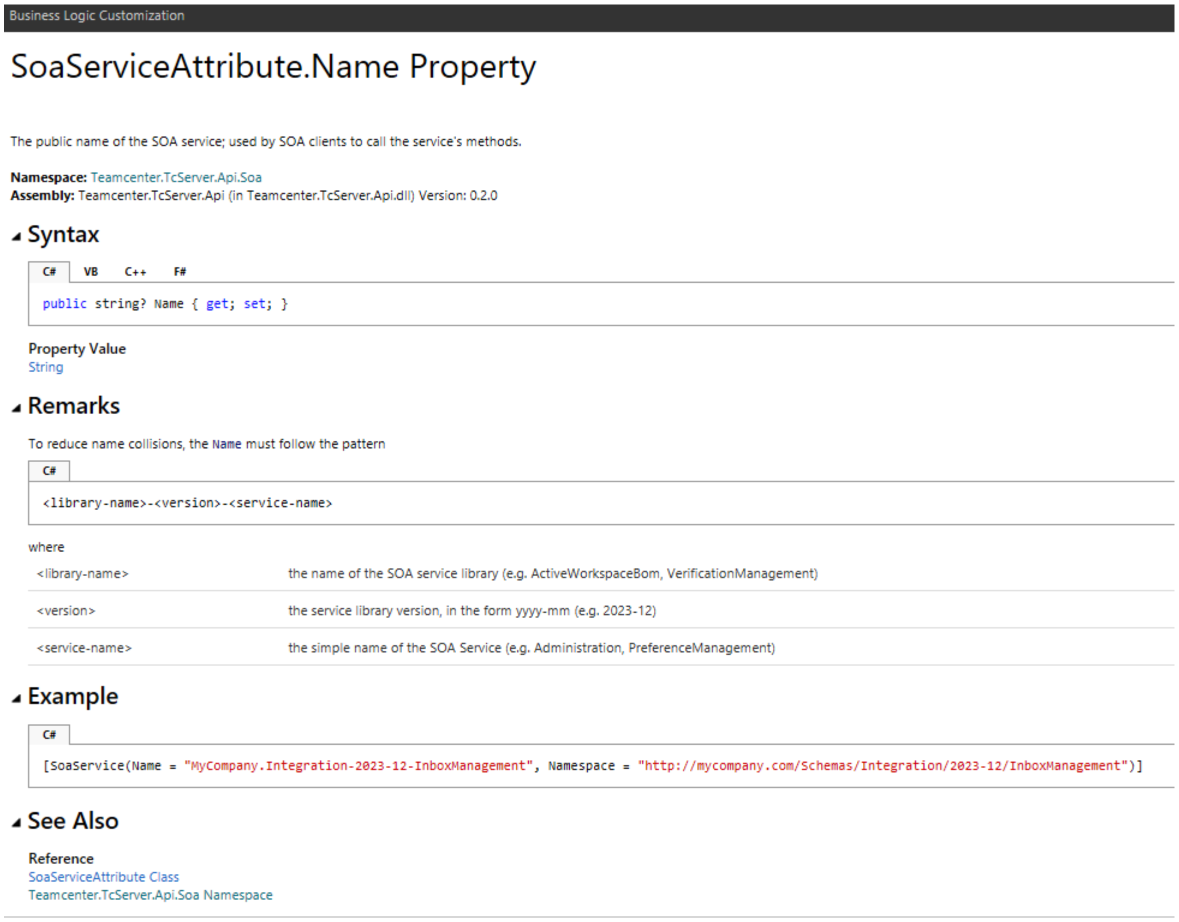
See Also

Reference
RuntimePropertyAttribute Class
Teamcenter.TcServer.Api.Metamodel Namespace

SoaService Name and Namespace consideration:

To avoid name collision **do not keep SOA “Name” and “Namespace” empty.** If you leave them empty they both will default to the class name, which is not recommended.

See the following for SoaService Name prefix:



Business Logic Customization

SoaServiceAttribute.Name Property

The public name of the SOA service; used by SOA clients to call the service's methods.

Namespace: Teamcenter.TcServer.Api.Soa
Assembly: Teamcenter.TcServer.Api (in Teamcenter.TcServer.Api.dll) Version: 0.2.0

Syntax

C# VB C++ F#

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

Property Value
String

Remarks

To reduce name collisions, the `Name` must follow the pattern

C#

```
<library-name>-<version>-<service-name>
```

where

<library-name>	the name of the SOA service library (e.g. ActiveWorkspaceBom, VerificationManagement)
<version>	the service library version, in the form yyyy-mm (e.g. 2023-12)
<service-name>	the simple name of the SOA Service (e.g. Administration, PreferenceManagement)

Example

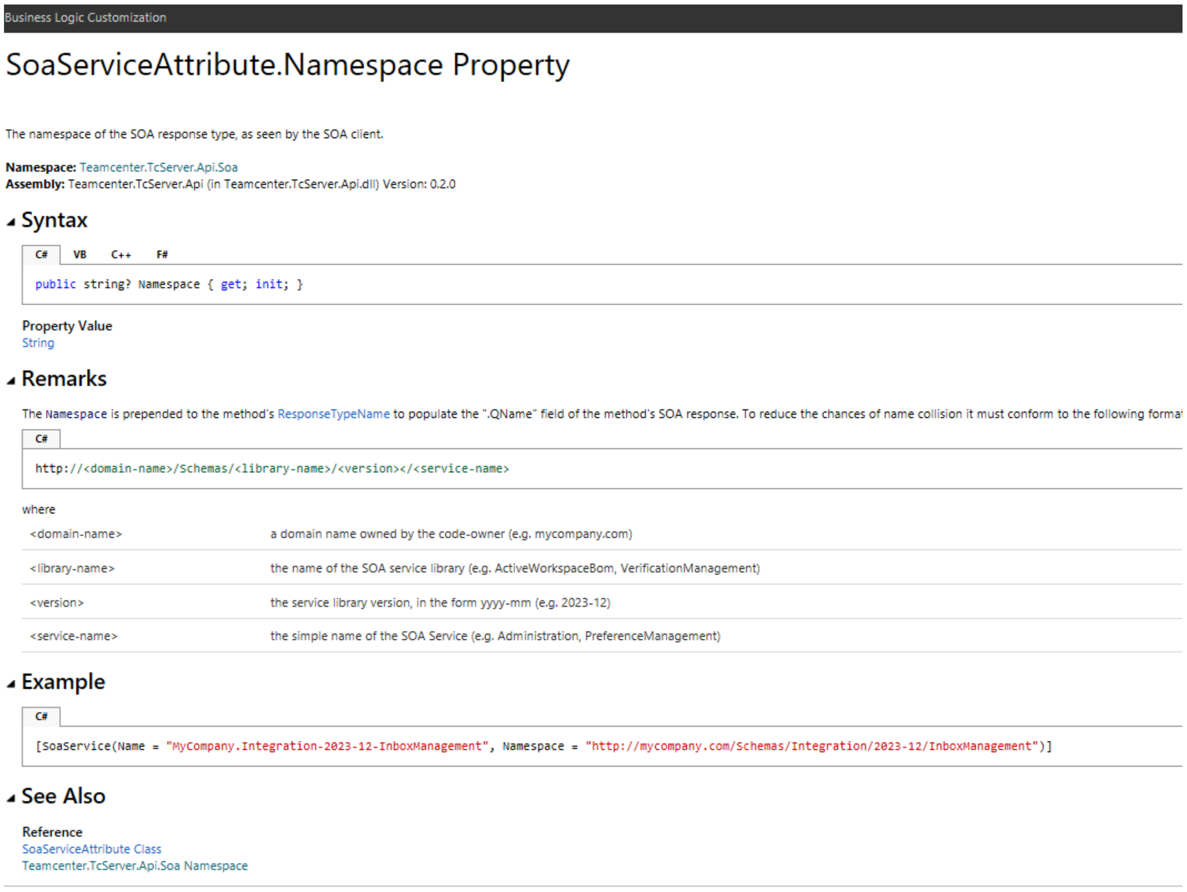
C#

```
[SoaService(Name = "MyCompany.Integration-2023-12-InboxManagement", Namespace = "http://mycompany.com/Schemas/Integration/2023-12/InboxManagement")]
```

See Also

Reference
SoaServiceAttribute Class
Teamcenter.TcServer.Api.Soa Namespace

See the following for SoaService Namespace prefix:



Business Logic Customization

SoaServiceAttribute.Namespace Property

The namespace of the SOA response type, as seen by the SOA client.

Namespace: Teamcenter.TcServer.Api.Soa
Assembly: Teamcenter.TcServer.Api (in Teamcenter.TcServer.Api.dll) Version: 0.2.0

Syntax

C# VB C++ F#

```
public string? Namespace { get; init; }
```

Property Value
String

Remarks

The Namespace is prepended to the method's ResponseTypeName to populate the ".QName" field of the method's SOA response. To reduce the chances of name collision it must conform to the following format

C#

```
http://<domain-name>/Schemas/<library-name>/<version></<service-name>
```

where

<domain-name>	a domain name owned by the code-owner (e.g. mycompany.com)
<library-name>	the name of the SOA service library (e.g. ActiveWorkspaceBom, VerificationManagement)
<version>	the service library version, in the form yyyy-mm (e.g. 2023-12)
<service-name>	the simple name of the SOA Service (e.g. Administration, PreferenceManagement)

Example

C#

```
[SoaService(Name = "MyCompany.Integration-2023-12-InboxManagement", Namespace = "http://mycompany.com/Schemas/Integration/2023-12/InboxManagement")]
```

See Also

Reference
SoaServiceAttribute Class
Teamcenter.TcServer.Api.Soa Namespace

See Teamcenter.TcServer.Metamodel.RuntimePropertyAttribute and Teamcenter.TcServer.Soa. SoaServiceAttribute in the attached TcServerDotNetCustomizationAPI.chm help file for details.

Localization and Constants for Runtime Property

To generate default localization and constants for runtime property use TemplateGen utility.

<u>Command For generating localization and constants file:</u>

TemplateGen --action AssemblyToTemplate --output c:\workdir\metamodel --template <path of the installed templates> --assembly <path to customization assembly>

Description of the arguments:

--**action** AssemblyToTemplate indicates to generate deployable template from customization assembly.

-- output indicates folder for auto generated files

--template indicates the directory where all the installed templates reside

--assembly indicates the path of the customization assembly. Note that TemplateGen will generate a json file in the assembly location if there is no existing json file. The file will have the following entry. You can change the Guid and Prefix. Here “SoaCoreTestExtensions” is the assembly name where customization exists.

{"Name":"SoaCoreTestExtensions","DisplayName":"SoaCoreTestExtensions","Prefix":"SoaA","Guid":"00000000-0000-0000-0000-000000000000","Version":"tc2312.0000"}

The above TemplateGen command will generate 4 xml files:

<customization_assembly_name>_dependency.xml – dependency template file

<customization_assembly_name>_deployable_template.xml – which has the constant definition and can be deployed.

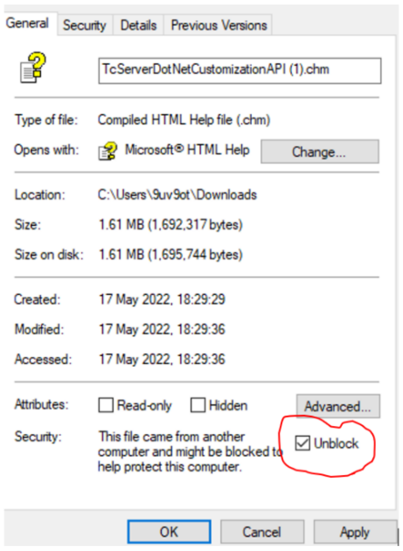
<customization_assembly_name>_template.xml – Which is a BMIDE loadable file, don’t deploy this

<customization_assembly_name>_template_en_US.xml – which has the localization and can be deployed

Help Contents

To see all the active content, you will have to copy the files to your local disk; you may also have to right-click on the file, select “Properties...” and check “Unblock” box at the bottom right:

You can create your own help file for your own customization by using “Sandcastle Help File Builder” (SHFB)


General | Security | Details | Previous Versions

TcServerDotNetCustomizationAPI (1).chm

Type of file: Compiled HTML Help file (.chm)

Opens with: Microsoft® HTML Help

Location: C:\Users\9uv9ot\Downloads

Size: 1.61 MB (1,692,317 bytes)

Size on disk: 1.61 MB (1,695,744 bytes)

Created: 17 May 2022, 18:29:29

Modified: 17 May 2022, 18:29:36

Accessed: 17 May 2022, 18:29:36

Attributes: ☐ Read-only ☐ Hidden

Security: This file came from another computer and might be blocked to help protect this computer. ☑ Unblock

Change...
Advanced...
OK
Cancel
Apply

TcServerDotNetCustomizationAPI.chm
TcServerDotNetCustomizationMock.chm