SmartSpace®

Managed Browser User Guide

From version 3.8

Ubisense Limited, St Andrew's House, St Andrew's Road, Cambridge CB4 1DL, United Kingdom.

Telephone: +44 (0)1223 535170. Website: https://www.ubisense.com

Copyright © 2023, Ubisense Limited 2014 - 2023. All Rights Reserved. You may not reproduce this document in whole or in part without permission in writing from Ubisense at the following address:

Ubisense Limited
St Andrew's House
St Andrew's Road
Cambridge CB4 1DL
United Kingdom

Tel: +44 (0)1223 535170

WWW: https://www.ubisense.com

All contents of this document are subject to change without notice and do not represent a commitment on the part of Ubisense. Reasonable effort is made to ensure the accuracy of the information contained in the document. However, due to on-going product improvements and revisions, Ubisense and its subsidiaries do not warrant the accuracy of this information and cannot accept responsibility for errors or omissions that may be contained in this document.

Information in this document is provided in connection with Ubisense products. No license, express or implied to any intellectual property rights is granted by this document.

Ubisense encourages all users of its products to procure all necessary intellectual property licenses required to implement any concepts or applications and does not condone or encourage any intellectual property infringement and disclaims any responsibility related thereto. These intellectual property licenses may differ from country to country and it is the responsibility of those who develop the concepts or applications to be aware of and comply with different national license requirements.

UBISENSE®, the Ubisense motif, SmartSpace® and AngleID® are registered trademarks of Ubisense Ltd. DIMENSION4™ and UB-Tag™ are trademarks of Ubisense Ltd.

Windows® is a registered trademark of Microsoft Corporation in the United States and/or other countries. The other names of actual companies and products mentioned herein are the trademarks of their respective owners.

Contents

Introduction to the Managed Browser ... **1**

Getting Started ... **2**

Getting the ManagedBrowser NuGet package ... 2

ManagedBrowser on Windows ... 6

ManagedBrowser on Linux ... 6

Debugging .NET Core Applications that use ManagedBrowser with the Visual Studio Debugger ... 7

Publishing your Code ... 8

Objects and Names ... **9**

Events ... **10**

Queries ... **13**

Getting Objects ... 13

Finding Types and Properties ... 13

Getting and Setting Property Values ... 13

Setting Properties ... 14

Converting Between Columns and Strings ... 15

Introduction to the Managed Browser

The ManagedBrowser is an interface to the SmartSpace User Data Model (UDM) from .NET applications. It is a more powerful alternative to the UDMAPI interface, but unlike the latter is not accessible through web service calls. Instead a .NET application must reference the ManagedBrowser assembly.

The ManagedBrowser supports creating and deleting objects, querying and setting properties, and callback when data model properties change.

ManagedBrowser is available on both Windows and Linux.

Getting Started

Getting the ManagedBrowser NuGet package

You can download the ManagedBrowser package from the Ubisense NuGet server at https://nuget.ubisense.net/.

You can also find the **ManagedBrowser.3.8.xxxx.nupkg** package in the **api\dotnet** folder of your distribution directory.

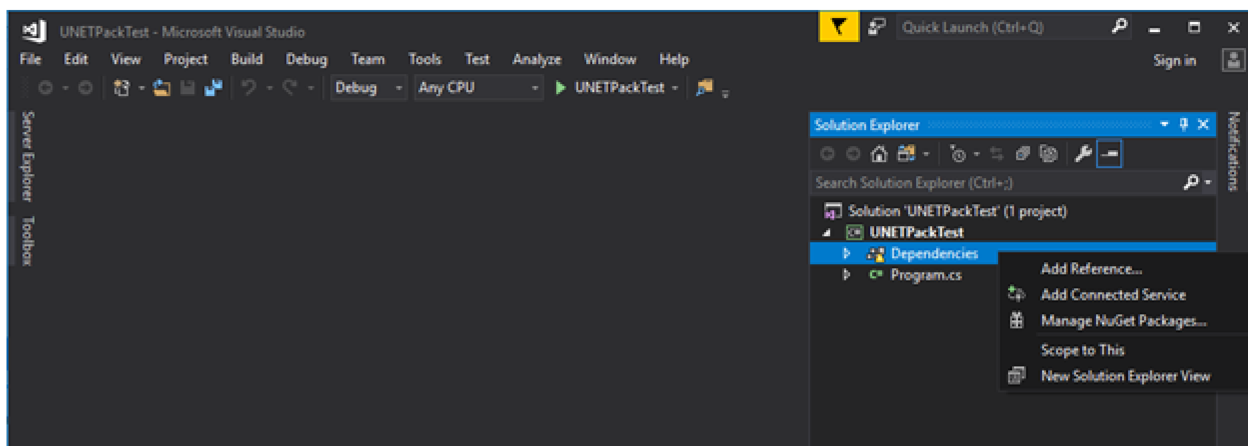
Setting up the package source

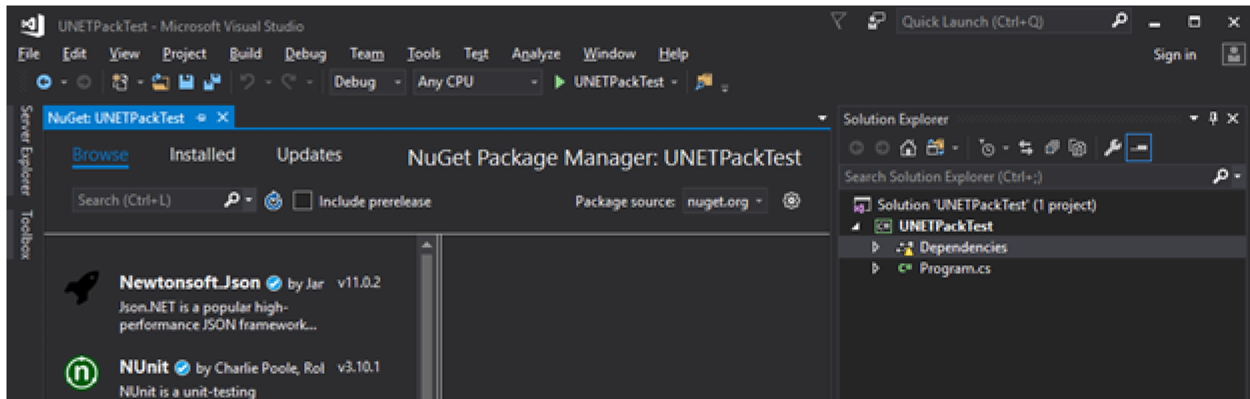
Download the Ubisense package ManagedBrowser file to an accessible location, for example **C:\Ubisense\packages**. Now use one of the following methods to add the package(s) to your project.

Adding sources within the Visual Studio GUI via the NuGet package manager

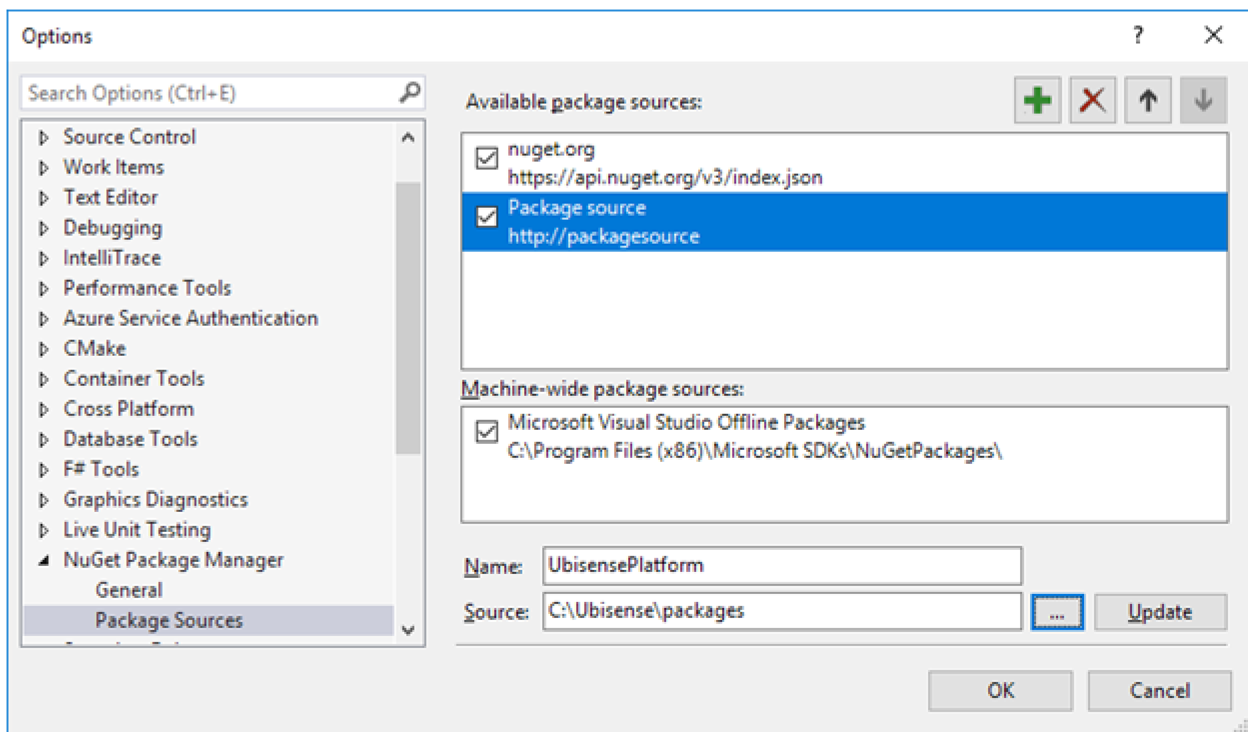
Configuration of sources can be done within the Visual Studio GUI. By default, this will configure your global NuGet settings. If you would like to configure NuGet sources for a specific project only, it is recommended you follow the instructions in *Creating a local NuGet.config file for a Visual Studio solution*.

Open your project within Visual Studio and open its NuGet package manager, then click the cog next to the package source box. Alternatively, choose Tools, Options... and navigate to NuGet Package Manager, Package Sources.

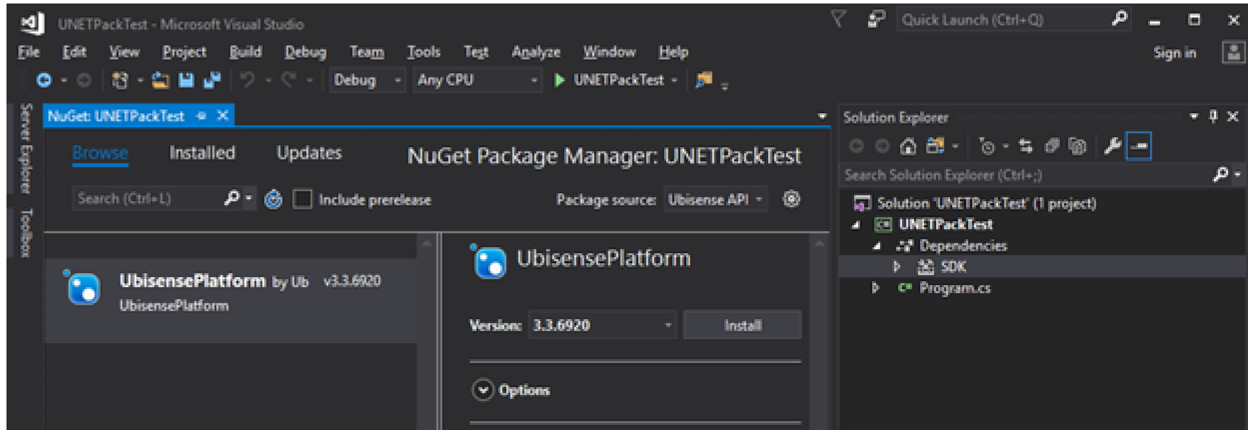




In this dialog, click the green plus to add a new package source. Give it a recognizable name and set the source to the directory containing the **ManagedBrowser.nupkg** file. Click **OK**.



Back in your project's NuGet package manager, select the new source from the package source drop down. The Managed browser should now be available to install to your project.



Adding sources within the Visual Studio GUI via the csproj file (.NET Core/Standard only)

For .NET Core and .NET standard projects, you can add the source and package details directly to a **.csproj** file within the VS GUI. Right click on your project and click **edit <ProjectName>.csproj**. The following lines will add a directory to the sources used when restoring your project, where `<source directory>` is the directory you want to add, e.g. **C:/Ubisense/packages**.

```
    <PropertyGroup>
      <RestoreSources>$(RestoreSources);<source
directory>;https://api.nuget.org/v3/index.json</RestoreSources>
    </PropertyGroup>
```

The package can then be added with the following lines.

```
    <ItemGroup>
      <PackageReference Include="ManagedBrowser" Version="3.x.xxxx" />
    </ItemGroup>
```

The version number should be the version of your package. Building and restoring packages for this project should fetch the required files.

This method will not edit the NuGet configuration so the package manager UI and other projects will not be affected.

Adding sources from the NuGet CLI

NuGet sources can be configured using the NuGet command-line tool. In a command prompt with NuGet in the path, sources are added using to following command:

```
nuget sources add -name <source name> -source <path to source>
```

This will add a source named `<source name>` and URL/file path of `<source>` to your NuGet configuration. By default, this is added to your global NuGet configuration but you can specify a different configuration file with the **-configfile** argument as follows:

```
nuget sources add -name <source name> -source <path to source> -configfile
<path to config file>
```

This configuration file must already exist and be of a valid format.

Creating a local NuGet.config file for a Visual Studio solution

A **NuGet.config** file in the same directory as a Visual Studio solution or project file will be detected by Visual Studio and used for the purposes of package management, in addition to the global settings, overriding in case of a conflict.

You can create a new, empty NuGet configuration file by creating a new file named **NuGet.config** with the following contents:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
</configuration>
```

You can then add a source to this configuration file with the nuget sources command, pointing at this configuration file, allowing use of this source with the package management UI within Visual Studio or dotnet CLI.

Adding sources from the dotnet CLI

A NuGet package can be added to a project directly with the .NET command-line interface. In the Visual Studio command prompt, or similar command prompt with the dotnet CLI in the path, run the following command:

```
dotnet add <solution> package ManagedBrowser --source <path to source
directory>
```

This will add the package to your project and immediately resolve it from the source directory supplied in the argument. However, this will not add this source to your NuGet configuration. Future restoring and building of this project may succeed, restoring from your NuGet package cache, but if this cache is cleared the restore will fail. It is recommended that you first add a source to NuGet for this package, as described in the other sections, then add the package to your solution without the source argument.

If you have already configured the NuGet sources for this project, the package can be added with the above command without the source argument.

ManagedBrowser on Windows

The ManagedBrowser is distributed as an **x86** architecture assembly, not a MSIL or x64. Thus it can only be referenced from a product which also targets x86.

If you are using .NET 4.0 or higher, you need to enable the legacy activation policy.
For example, in your app.config:

```
 <startup useLegacyV2RuntimeActivationPolicy="true">
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.1"/>
</startup>
```

You also need to set the platform target in the **.csproj** file:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <PlatformTarget>x86</PlatformTarget>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="UbisensePlatform" Version="3-*" />
    <PackageReference Include="ManagedBrowser" Version="3-*" />
  </ItemGroup>

</Project>
```

Add **ManagedBrowser.dll** and **UbisensePlatform.dll** from the distribution as references in your project. You should now be able to construct an instance of the Ubisense.UDMAPI.ManagedBrowser class.

```
ManagedBrowser browser = new ManagedBrowser();
```

ManagedBrowser on Linux

The ManagedBrowser is available on Linux where it is distributed as an **x64** architecture assembly. So you need to set the platform target in the **.csproj** file as follows:

```
<PropertyGroup>
…
   <PlatformTarget>x64</PlatformTarget>
…
</PropertyGroup>
```

Debugging .NET Core Applications that use ManagedBrowser with the Visual Studio Debugger

The following work-around is required to get a .NET core application that uses the Ubisense ManagedBrowser API to run within the Visual Studio debugger. This doesn't work out of the box with .NET Core 2.x because the Visual Studio debugger tries to start the 64-bit dotnet.exe, rather than the 32-bit dotnet.exe required by an x86 executable.

To override this behavior, the Directory.Build.targets file needs to be dropped into a folder in or above the project file. This ensures the correct executable is run. The content of **Directory.Build.targets** is shown below:

```
<Project>
  <PropertyGroup
      Condition="'$(OS)' == 'Windows_NT' and
                 '$(TargetFrameworkIdentifier)' == '.NETCoreApp' and
                 '$(SelfContained)' != 'true'"
                  >
    <RunCommand Condition="'$(PlatformTarget)' ==
'x86'">$(MSBuildProgramFiles32)\dotnet\dotnet</RunCommand>
    <RunCommand Condition="'$(PlatformTarget)' ==
'x64'">$(ProgramW6432)\dotnet\dotnet</RunCommand>
  </PropertyGroup>
</Project>
```


```
<Project>
  <PropertyGroup
      Condition="'$(OS)' == 'Windows_NT' and
                 '$(TargetFrameworkIdentifier)' == '.NETCoreApp' and
                 '$(SelfContained)' != 'true'"
                  >
    <RunCommand Condition="'$(PlatformTarget)' ==
'x86'">$(MSBuildProgramFiles32)\dotnet\dotnet</RunCommand>
    <RunCommand Condition="'$(PlatformTarget)' ==
'x64'">$(ProgramW6432)\dotnet\dotnet</RunCommand>
  </PropertyGroup>
</Project>
```

Publishing your Code

When you publish your code, you must specify the correct runtime version with the **publish** command. These are:

- Windows: `dotnet publish -r win-x86`
- Linux: `dotnet publish -r linux-x64`

Objects and Names

Objects are represented in results as a string form including the object id and type. This is intended as an opaque identifier which can be used to refer to the object. For example:

```
04007zLfF_94Wzfm000IsG0000F:UserDataModel::Product
```

Most objects will also have a name, and the name can be retrieved using the `get_name` method.

```
var n = browser.get_name(cobj);
Console.WriteLine("{0} has name {1}", cobj, n);
```

If the object is not a known object, such as when it has been deleted, the result will be null.

Conversely, the object can be retrieved for a given name using `get_object`.

```
var cobj2 = browser.get_object("Product", n);
```

Note that if the object of that name and type is not found, this method returns null.

Events

To receive events when properties change, first create a callback class which implements the `IRowEvents`.

```
    class MyCallback : IRowEvents
        {
            #region IRowEvents Members

            public void data_inserted(string prop, List<string> row)
            {
                WriteRow("insert", prop, row);
            }

            public void data_removed(string prop, List<string> row)
            {
                WriteRow("remove", prop, row);
            }

            public void data_updated(string prop, List<string> before, List<string>
after)
            {
                WriteRow("update from", prop, before);
                WriteRow("update to", prop, after);
            }

            public void establish()
            {
                Console.WriteLine("establish");
            }

            public void schema_changed()
            {
                Console.WriteLine("schema changed");
            }

            #endregion

            private void WriteRow(string desc, string prop, List<string> row)
            {
                Console.Write("{1} {0}:", desc, prop);
                bool first = true;
                foreach (var p in row)
                {
                    if (!first) Console.Write(",");
                    Console.Write(p);
                    first = false;
                }
                Console.WriteLine();
            }

        };
```

Now set an instance of the class as the callback for the browser.

```
ManagedBrowser browser = new ManagedBrowser();
 var cb = new MyCallback();
 browser.set_event_callback(cb);
```

Now use `add_property` to specify which property change events to receive.

```
browser.add_property("name<Product>");
 browser.add_property("[Custom]test_bool<Product>");
```

NOTE: If you expect the data model schema to change (specifically properties or named types to be removed) then you should also periodically call `update_definitions`. Otherwise the browser only checks for updated data model schema when queries are performed, so if your application only responds to events, it would stop receiving any when the data model schema changes.

Queries

The browser supports a number of queries of the data model.

Getting Objects

The set of objects of a given type can be retrieved using `get_named_objects` or `get_named_objects_with_descendants`. The first returns only objects exactly matching the given type. The second returns objects which are of the given type or inherited from that type. In either case the output is a dictionary mapping from object to name.

Finding Types and Properties

To see the names of properties and types currently defined in the data model, use the `all_properties` and `all_types` methods. Note that types and properties include the namespace to which they belong, such as `[Offline]` or `[Custom]`.

Each property row maps from a key to a value, where the key can be one or more columns, and the value is a single column. Simple properties have a single column in the key (the object on which the property is defined).

The types of a property can be queried:

- To get the types of each key column use `get_property_key_types`
- To get the type of the value column use `get_property_value_type`

Getting and Setting Property Values

Individual rows of a property can be retrieved using `get_property_value`, which takes the name of the property and a list of strings as the key.

```
var cobj = browser.get_object("Product", n);
string val;
if (browser.get_property_value("[Custom]test_bool<Product>", new List<string> { cobj },
out val))
{
    Console.WriteLine("{0} -> {1}", cobj, val);
}
```

This method returns true on success, or false on failure. Failure can be because the property is not known, or the types or length of the key parameter do not agree with the property definition.

The complete set of rows in a property can be retrieved using `get_property_values`, which returns a dictionary mapping from key to value.

```
Dictionary<List<string>, string> rows;
browser.get_property_values("[Offline]entry_time<Product>", out rows);
foreach (var r in rows)
{
    StringBuilder kb = new StringBuilder();
    bool first = true;
    foreach (var k in r.Key)
    {
        if (!first) kb.Append(",");
        kb.Append(k);
        first = false;
    }
    var s = r.Value;
    var d = browser.convert_datetime(s);
    Console.WriteLine("{0}: {1} = {2}", kb, r.Value, d.ToLocalTime());
}
```

This method returns false if the property is unknown.

Setting Properties

The value of a property can be set using `set_property_value`, and can be removed using `delete_property_value`.

```
var cobj = browser.get_object("Product", n);
browser.set_property_value("[Custom]test_bool<Product>", new List<string> { cobj },
"true");

...

browser.delete_property_value("[Custom]test_bool<Product>", new List<string> { cobj
});
```

Unlike the UDMAPI, setting a property value to the empty string does not delete the property. This allows empty string-valued properties to be set, if necessary. Instead always use `delete_property_value` if you intend to remove a row.

The methods both return a bool, which is true on success, false on failure. Reasons for failure could include: the property named does not exist, or the passed parameters do not match the types in the property, or insufficient columns were passed for the key.

Converting Between Columns and Strings

Columns of type Object and Time are returned in a special string form, which can be converted to types compatible with the rest of the platform .NET API using the convert methods of the browser.

- `convert_object`: turns an object column into a Ubisense.UBase.UObject
- `convert_datetime`: turns a date column into a UTC System.DateTime
- `convert_string`: takes either a Ubisense.UBase.UObject or a System.DateTime and returns the column string equivalent