Using the Caché Gateway for .NET

Version 2018.1
2024-11-07

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using the Caché Gateway for .NET
PDF generated on 2024-11-07
InterSystems Caché® Version 2018.1

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

List of Figures

1

About This Book

See the Table of Contents for a detailed listing of the subjects covered in this document.

This book explains how to enable easy interoperation between Caché and Microsoft .NET Framework components using the Caché Object Gateway for .NET, which can instantiate an external .NET object and manipulate it as if it were a native object within Caché.

The following topics are covered:

- Gateway Architecture
- Setting Gateway Server Properties
- Running a Gateway Server
- Creating Proxy Classes
- Sample Code
- Mapping Specification

2

Gateway Architecture

The Caché Object Gateway for .NET (which this book will usually refer to as simply "the .NET Gateway") provides an easy way for Caché to interoperate with Microsoft .NET Framework components. The .NET Gateway can instantiate an external .NET object and manipulate it as if it were a native object within Caché.

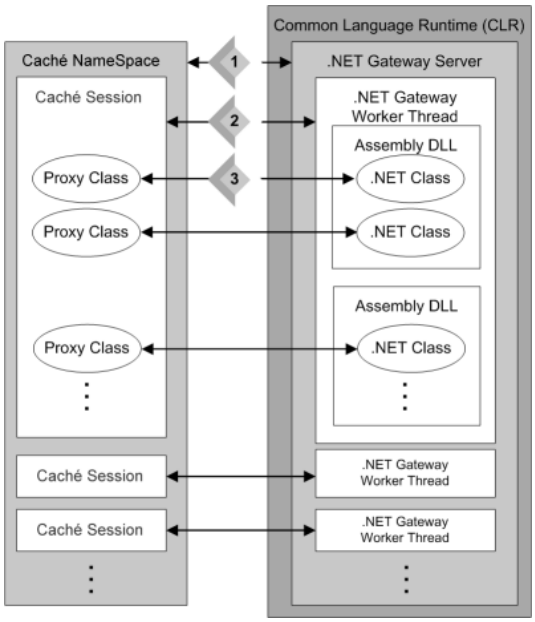
Note: The .NET Gateway can also be used in an Ensemble production (see the Ensemble document *Using the Object Gateway for .NET*). You can create and test your .NET Gateway classes in Caché, as described in this book, and then add them to a production using the Ensemble .NET Gateway business service.

2.1 Using Proxy Classes

The external .NET object is represented within Caché by a proxy class. A proxy object looks and behaves just like any other Caché object, but it has the capability to issue method calls out to the .NET Common Language Runtime (CLR), either locally or remotely over a TCP/IP connection. Any method call on the Caché proxy object triggers the corresponding method of a .NET object inside the CLR.

The following diagram offers a conceptual view of Caché and the .NET Gateway at runtime.

Figure 2–1: .NET Gateway Operational Model



Instances of the .NET Gateway Server run in the CLR. Caché and the CLR may be running on the same machine or on different machines. The numbered items in the *.NET Gateway Operational Model* diagram point out the following relationships:

1. A Caché namespace accesses an instance of the .NET Gateway Server. Access is controlled by an instance of the Caché %Net.Remote.Service class.
2. Each Caché session is connected to a separate thread within the Gateway server. The connection is controlled by an instance of the Caché %Net.Remote.Gateway class.
3. Each proxy object communicates with a corresponding .NET object.

A call to any Caché proxy method initiates the following sequence of events:

- Caché sends a message over the TCP/IP connection to the .NET Gateway worker thread. The message consists of the method name, parameters, and occasionally some other information.
- The .NET Gateway worker thread finds the appropriate method or constructor call and invokes it using .NET reflection.
- The results of the method invocation (if any) are sent back to the Caché proxy object over the same TCP/IP channel, and the proxy method returns the results to the Caché application.

Note: You can access a proxy class with code written in either Caché Basic or ObjectScript. The examples in this document use ObjectScript.

2.2 Using Wrapper Classes with .NET APIs

In most cases, you will use the .NET Gateway by creating proxy classes for your custom .NET components (see the chapter on "Creating Proxy Classes" for details). However, it is also possible to create proxy mappings for an entire third party .NET application interface specification.

It may be tempting to create mappings for extremely large APIs (such as ADO, Remoting, or ASP.Net), which could then be reused in any number of applications, but this is not recommended. Such mappings can create hundreds of proxy classes, even though your application may only need a few of them. You can specify a list of classes that you do not want the proxy generator to map, but a very large exclusion list is difficult to create and manage.

When using a third party DLL in your application, the best approach is to build a small wrapper class for it, and then create a proxy for this wrapper. A wrapper class exposes only the functionality you want, which makes the interface between Caché and the .NET framework very clean and eliminates many potential issues. Wrapper classes provide the following advantages:

- *Fewer proxy classes* — When a large .NET DLL is imported directly, hundreds of proxy classes may be generated. Your application may actually need only a few of these classes. A significant amount of work may be required to research and define an exclusion list for the import. Defining a wrapper is a much easier way to minimize the number of classes imported and managed.
- *Fewer problems with dependencies* — The imported DLL may depend on external classes for successful compilation. The files and directories containing these classes can be specified in an inclusion list when you run the proxy generator, but this will cause even more unwanted proxy classes to be generated. When a wrapper is used, the dependent files and their classes are hidden from the importer, eliminating the need for inclusion and exclusion lists.
- *Better performance* — A wrapper may allow you to reduce the number of calls made to a DLL. For example, assume that an imported DLL has a class with a **readByte()** method that reads one byte at a time. If you import the class directly, each call to the method will require a separate call to the DLL. It would be much more efficient to define a wrapper class with a **readManyBytes()** method that repeatedly calls the **readByte()** method internally, within .NET.

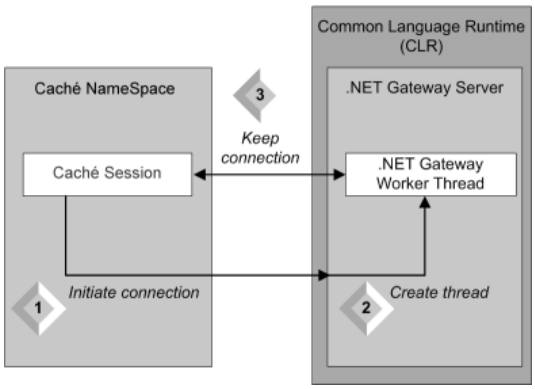
- *Ability to use legacy DLLs* — ActiveX DLLs cannot be used directly in a 64–bit Windows environment, and a DLL that was not written as a .NET assembly cannot be used with the .NET Gateway even in a 32–bit environment. However, it is possible to create a wrapper that the .NET Gateway can use to call a DLL indirectly.

2.3 Creating and Running a Gateway Server

Before you can use the .NET Gateway, you must start an instance of the .NET Gateway Server and tell Caché the name of the host on which the server is running. Once started, a server runs until it is explicitly shut down.

Once the .NET Gateway server is running, each Caché session that needs to invoke .NET class methods must create its own connection to the server, as shown in the following diagram:

Figure 2–2: Connecting to a .NET Gateway Worker Thread



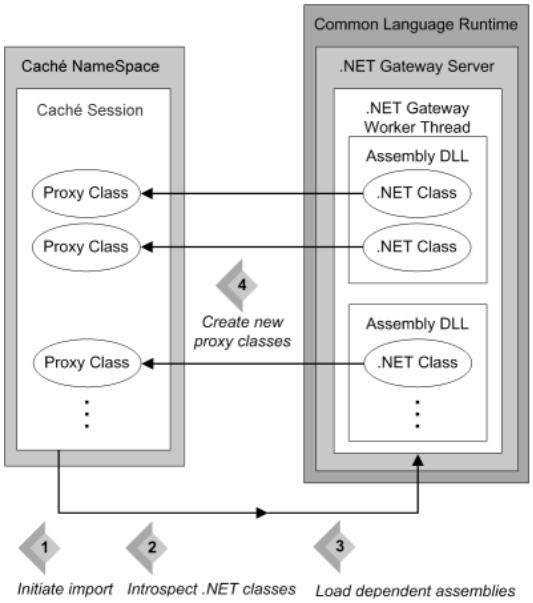
1. Caché Basic or ObjectScript code sends a connection request.
2. Upon receiving the request, the .NET Gateway server starts a worker thread in which the .NET class methods subsequently run.
3. The connection between this .NET Gateway worker thread and the corresponding Caché session remains established until it is explicitly disconnected. As long as it remains connected, the assigned port for the connection stays "in use" and is unavailable for use in other connections.

See Setting Gateway Server Properties for a detailed description of how to create a .NET Gateway Server property definition, and Running a Gateway Server for details on how to start, connect, disconnect, and stop a server.

2.4 Importing Proxy Classes

Caché proxy classes are generated by sending a query to the .NET Gateway Server, which returns information about the methods for which proxy classes are required. The imported method information is then used to construct the proxy classes, as shown in the following diagram:

Figure 2–3: Importing .NET Classes



1. The Caché session sends an import request.
2. Upon receiving the request, the .NET Gateway worker thread introspects the indicated .NET assemblies and classes.
3. The thread also loads dependent assemblies either from the local directory or from the Global Assembly Cache (GAC).
4. If it finds any .NET classes that are new or changed, or that have no proxy classes on the Caché side, the .NET Gateway worker thread returns the results of the introspection to the Caché session, which uses the information to generate new proxy classes.

See Creating Proxy Classes for details on how to generate proxy classes.

2.5 The .NET Gateway API

The following classes provide most of the functionality used by your Caché .NET Gateway applications:

- %Net.Remote.ObjectGateway — an ObjectGateway object contains the property settings required to run and monitor an instance of the .NET Gateway Server. See Defining a Gateway Server for a detailed description.
- %Net.Remote.Service — a Service object controls the interface between a Caché namespace and an instance of the .NET Gateway Server. See Running a Gateway Server.
- %Net.Remote.Gateway — a Gateway object controls the connection between a Caché session and a worker thread within an instance of the .NET Gateway Server, and provides methods to generate proxy classes. See Connecting to a Server and Generating Proxy Classes Programmatically.
- %Net.Remote.ImportHelper — the ImportHelper class provides some extra class methods for inspecting assemblies and generating proxy classes. See Generating Caché Proxy Classes.

See the Caché class library documentation for the most complete and up to date information on each of these classes.

3

Setting Gateway Server Properties

Properties of the ObjectGateway class specify the settings used by the Service class (see "Running a Gateway Server") to access and monitor an instance of the .NET Gateway Server. This chapter covers the following topics:

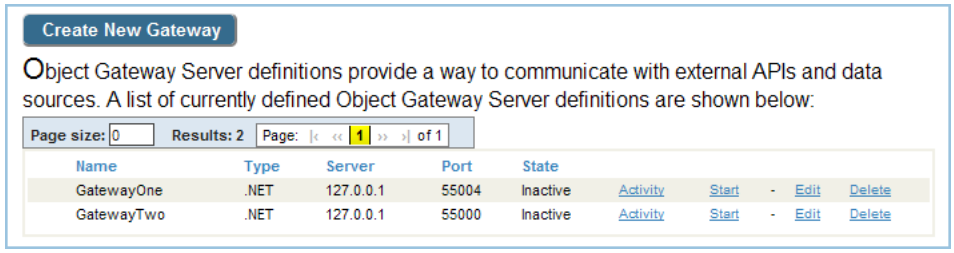
- Using the New Object Gateway Form — The simplest way to define a set of server properties is to use the New Object Gateway form in the Management Portal, which allows you to create and store a predefined ObjectGateway object.
- Defining Server Properties Programmatically — It is also possible to set the properties of a ObjectGateway object at runtime.
- Gateway Server Versions — Different versions of the .NET Gateway Server executable are provided for .NET 2.0, 4.0, and 4.5, and for 32-bit and 64-bit systems.

3.1 Using the New Object Gateway Form

While it is possible to specify the settings for a .NET Gateway Server session entirely in ObjectScript code, it is usually simpler to use the New Object Gateway form to create and store a persistent %Net.Remote.ObjectGateway object. The following steps summarize the configuration procedure:

1. In the Management Portal, go to System Administration > Configuration > Connectivity > Object Gateways.

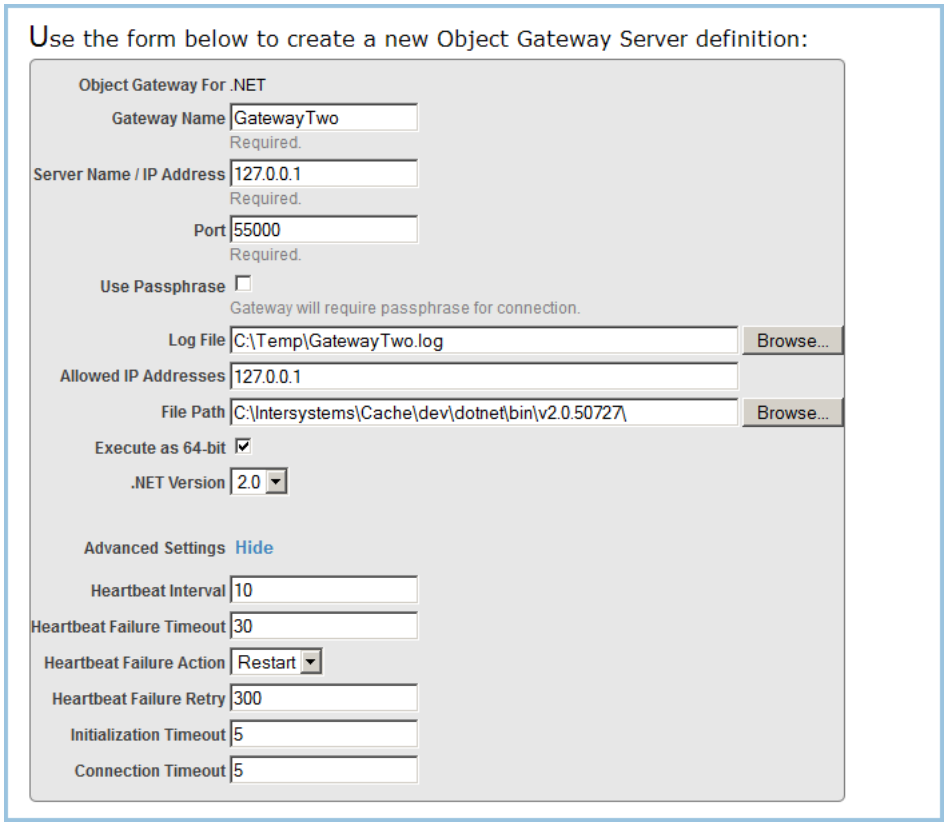
Figure 3–1: The Object Gateways Page



   You can use the options on this page to view a Gateway's log of recent activity, start or stop a Gateway, and create, edit, or delete a Gateway Server definition.

2. On the `Object Gateways` page, click `Create New Gateway`. The New Object Gateway form is displayed. Only the first three fields are required. The following example leaves the default values in place for all fields except `Gateway Name`, `Port`, and `Log file` (`File path` is set to the value that would be used if the field were left blank):

Figure 3–2: The New Object Gateway Form



3. Fill out the form and click `Save`. The following properties of ObjectGateway can be set:

   - Name — *Required*. A name that you assign to this .NET Gateway definition.
   - Server — *Required*. IP address or name of the machine where the .NET Gateway server executable is located. The default is `"127.0.0.1"`.
   - Port — *Required*. TCP port number for communication between the .NET Gateway server and the proxy classes in Caché. Several methods of class Service (see "Running a Gateway Server") accept the port number as an optional parameter with a default value of `55000`.
   - PassPhrase — If this property is checked, the Gateway will require a passphrase to connect.
   - LogFile — Full pathname of the file used to log .NET Gateway messages. These messages include acknowledgment of opening and closing connections to the server, and any difficulties encountered in mapping .NET classes to Caché proxy classes. This optional property should only be used when debugging.
   - AllowedIPAddresses — IP address of the local network adapter to which the server listens. Specify `0.0.0.0` (the default) to listen on all IP adapters on the machine (including 127.0.0.1, VPNs, and others) rather than just one adapter. You may not enter more than one address; you must either specify one local IP address, or listen on all of them. You must provide a value for this argument if you define the LogFile property.
   - FilePath — Specifies the full path of the directory where the .NET Gateway server executable is located. This is used to find the target executable and assemble the command to start the .NET Gateway on a local server, and is required only when you want to use an executable that is not in the default location. If you do not specify this setting, the form will use the appropriate default executable for your system and your specified .NET Version.
   - Exec64 — (applies only to .NET Gateways on 64-bit platforms) If this property is checked, the Object Gateway server will be executed as 64-bit. Defaults to 32-bit execution.

- DotNetVersion — Specifies the .NET version (2.0, 4.0, or 4.5) to be used (see "Gateway Server Versions"). Defaults to .NET 2.0.
- *Advanced settings* (hidden by default)
  - HeartbeatInterval — Number of seconds between each communication with the .NET Gateway to check whether it is active. A value of 0 disables this feature. When enabled, valid values are from 1 to 3600 seconds (1 hour). The default is 10 seconds.
  - HeartbeatFailureTimeout — Number of seconds to wait for a heartbeat response before deciding that the .NET Gateway is in failure state. If this value is smaller than the HeartbeatInterval property, the Gateway is in failure state every time the .NET Gateway communication check fails. The maximum value is 86400 seconds (1 day). The default is 30 seconds.
  - HeartbeatFailureAction — Action to take if the .NET Gateway goes into a failure state. Valid values are `"R"` (Restart) or `"N"`(None). The default action is Restart, which causes the .NET Gateway to restart.
  - HeartbeatFailureRetry — Number of seconds to wait before retrying the HeartbeatFailureAction if the .NET Gateway server goes into failure state, and stays in failure state. A value of 0 disables this feature, meaning that once there is a failure that cannot be immediately recovered, there are no attempts at automatic recovery. Valid values when enabled are from 1 to 86400 (24 hours). The default is 300 seconds (5 minutes).
  - InitializationTimeout — Number of seconds to wait for a response during initialization of the Gateway server. Valid values are from 2 to 300 (5 minutes). The default is 5 seconds.
  - ConnectionTimeout — Number of seconds to wait for a connection to be established with the Gateway server. Valid values are 2 through 300 (5 minutes). The default is 5 seconds.

4. After saving the form, go back to the Object Gateways page. Your new .NET Gateway Server definition should now be listed there.

3.2 Defining Server Properties Programmatically

The ObjectGateway server properties can also be set programmatically.

Note: In addition to the properties defined by the New Object Gateway form (as shown in the previous section), the Type property must also be set to `"2"`. This defines the server as a .NET Gateway (as opposed to an Ensemble Java Gateway, which cannot be used in Caché).

The following example creates a server definition identical to the one generated by the New Object Gateway form described in the previous section:

ObjectScript

```
// Create the object and define it as a .NET Gateway
  set gw =  ##class(%Net.Remote.ObjectGateway).%New()
  set gw.Type = "2"   // a .NET Gateway, not an Ensemble Java Gateway

// Set the properties
  set gw.Name  = "GatewayTwo"
  set gw.Server  = "127.0.0.1"
  set gw.Port  = "55000"
  set gw.PassPhrase = 1
  set gw.LogFile  = "C:\Temp\GatewayTwo.log"
  set gw.AllowedIPAddresses  = "0.0.0.0"
  set gw.FilePath  = "C:\Intersystems\Cache\dev\dotnet\bin\v2.0.50727"
  set gw.Exec64 = 1
  set gw.DotNetVersion = "2.0"
  set gw.HeartbeatInterval  = "10"
```

```
   set gw.HeartbeatFailureTimeout  = "30"
   set gw.HeartbeatFailureAction  = "R"
   set gw.HeartbeatFailureRetry  = "300"
   set gw.InitializationTimeout  = "5"
   set gw.ConnectionTimeout  = "5"

 // Save the object
   set status = gw.%Save()
```

The call to **%Save()** is only necessary if you wish to add the object to persistent storage, making it available on the Object Gateways page of the Management Portal (as described in the previous section).

3.3 Gateway Server Versions

Different versions of the .NET Gateway Server assembly are provided for .NET 2.0, 4.0, and 4.5. The following versions are shipped (where *install-dir* is the path that `$SYSTEM.Util.InstallDirectory()` returns on your system).

.NET Version 2.0:

- *install-dir*\dev\dotnet\bin\v2.0.50727\DotNetGatewaySS.exe
- *install-dir*\dev\dotnet\bin\v2.0.50727\DotNetGatewaySS64.exe

.NET Version 4.0:

- *install-dir*\dev\dotnet\bin\v4.0.30319\DotNetGatewaySS.exe
- *install-dir*\dev\dotnet\bin\v4.0.30319\DotNetGatewaySS64.exe

.NET Version 4.5:

- *install-dir*\dev\dotnet\bin\v4.5\DotNetGatewaySS.exe
- *install-dir*\dev\dotnet\bin\v4.5\DotNetGatewaySS64.exe

In some applications, these gateways may be used to load unmanaged code libraries. Since a 64-bit process can only load 64-bit DLLs and a 32-bit process can only load 32-bit DLLs, both 32-bit and 64-bit assemblies are provided for each supported version of .NET. This makes it possible to create gateway applications for 64-bit Windows that can load 32-bit libraries into the gateway.

Note: The appropriate version of the .NET Framework must be installed on your system in order to use these assemblies. The Caché installation procedure does not install or upgrade any version of the .NET Framework.

4

Running a Gateway Server

Each Gateway server session consists of the following components:

- a NET Gateway Server instance (see "Gateway Server Versions") running in the .NET CLR
- a %Net.Remote.Service object running in a Caché namespace
- a unique TCP port over which the two objects communicate
- one or more connections, where each connection links a Caché %Net.Remote.Gateway object to a thread within the server instance

See "Gateway Architecture" for a detailed description and several diagrams showing how these components interact.

4.1 Starting the Server

These %Net.Remote.Service methods are available to start the server:

- **StartGateway()** — Start the .NET Gateway server specified by a Gateway name.
- **StartGatewayObject()** — Start the .NET Gateway server specified by a Gateway definition object.
- **OpenGateway()** — Get the Gateway definition object for a given Gateway name.

4.1.1 Starting a Server from the Command Prompt

During development or debugging, or when Caché and the .NET Gateway server run on different machines, you may find it useful to start the Gateway server from a command prompt.

Note: The Gateway server executable will normally be located in a default directory (see "Gateway Server Versions"). If you are using classes in local side-by-side assemblies (assemblies not installed into the GAC), copy the executable to the same directory as those assemblies and run it from there to resolve their dependencies.

Run the program as follows:

`DotNetGatewaySS` *port host logfile*

For example:

```
DotNetGatewaySS 55000 "" ./gatewaySS.log
```

Argument	Description
port	Port number on which to listen for the incoming requests.
host	*Optional* — Contains the IP address or hostname where the Gateway server listens. Specify `""`, `0.0.0.0`, or default to listen on all IP adapters on the machine (127.0.0.1, VPNs, etc.) rather than just one adapter.
logfile	*Optional* — If specified, the command procedure creates a log file of that name. You must specify the full pathname in the string.

4.2 Connecting to a Server Thread

Connecting creates a %Net.Remote.Gateway object.

Once the .NET Gateway server is running, each Caché session that needs to invoke .NET class methods must create its own connection to the .NET Gateway server:

- Caché Basic or ObjectScript code sends a connection request.
- Upon receiving the request, the .NET Gateway server starts a worker thread in which the .NET class methods subsequently run.
- The connection between this .NET Gateway worker thread and the corresponding Caché session remains established until it is explicitly disconnected.

4.2.1 The Gateway.%Connect() Method

The **%Connect()** method establishes a connection with the .NET Gateway engine. It accepts the following arguments:

Argument	Description
host	Identifies the machine on which the .NET Gateway server is running.
port	Port number over which the proxy classes communicate with the .NET classes.
namespace	Caché namespace.
timeout	Number of seconds to wait before timing out, the default is 5.
additionalClassPaths	Optional — use this argument to supply additional class paths, such as the names of additional assembly DLLs that contain the classes you are importing via the .NET Gateway. See the Import Arguments section for details using this argument.

4.3 Disconnecting and Stopping the Server

Caché Basic or ObjectScript code that establishes a .NET Gateway worker thread must explicitly disconnect before exiting; otherwise, the assigned port for the connection stays "in use" and is unavailable for use in other connections. A worker thread can be disconnected by calling the **%Disconnect()** method of the %Net.Remote.Gateway object.

- The **%Disconnect()** method closes a connection to the .NET Gateway server.

The following %Net.Remote.Service methods are available to stop the Gateway server:

- **StopGateway()** — Stop the .NET Gateway server specified by the Gateway name passed to this method.
- **StopGatewayObject()** — Stop the .NET Gateway server specified by the Gateway definition object passed to this method.
- **ShutdownGateway()** — Shutdown the Gateway server.

4.4 Monitoring and Debugging the Gateway

The following %Net.Remote.Service methods are available to monitor a Gateway server:

- **CheckMonitor()** — Check if Gateway is being monitored and return the monitor job number.
- **GatewayMonitor()** — The Gateway server is monitored with PING requests, according to the time interval specified by the HeartbeatInterval property of the Gateway server. Hourly, a record of type "Info" is logged.
- **StartMonitor()** — If the Gateway server has the HeartbeatInterval property set to a value greater than 0, then job off a background process to monitor the Gateway server.
- **StopMonitor()** — Terminate process currently monitoring a Gateway server.
- **PingGateway()** — "Ping" the Gateway server to check if it's alive.

4.4.1 Error Checking

The .NET Gateway provides error checking as follows:

- When an error occurs while executing Caché proxy methods, the error is, in most cases, a .NET exception, coming either from the original .NET method itself, or from the .NET Gateway engine. When this happens, an error is trapped.
- The .NET Gateway API methods like **%Import()** or **%Connect()** return a typical Caché %Status variable.

In both cases, Caché records the last error value returned from a .NET class (which in many cases is the actual .NET exception thrown) in the local variable *%objlasterror*.

You can retrieve the complete text of the error message by calling **$system.OBJ.DisplayError**, as follows:

ObjectScript

```
Do $system.OBJ.DisplayError(%objlasterror)
```

Note that *%objlasterror* should be used as a debug resource only (for example, in development code that does not yet report errors properly), so that the underlying problem can be diagnosed and the offending code's error reporting can be corrected. It is appropriate for such code to kill *%objlasterror* whenever it uses an error status that is an expected condition and not a reportable error.

4.4.2 Troubleshooting

The following suggestions may help in certain situations:

- *Activate logging*

  Should you encounter problems while using the .NET Gateway it is always a good idea to turn logging on. This will also aid InterSystems staff in helping you troubleshoot problems. To activate logging, just define the *logfile* argument

for the Object Gateway definition you are using (see the chapter on "Setting Gateway Server Properties ") when you start the .NET Gateway.

- *Terminal session problems*

  Sometimes, while using the .NET Gateway in a debugging or test situation, you may encounter problems with a Terminal session becoming unusable, or with write errors in the Terminal window. It is possible that a .NET Gateway connection terminated without properly disconnecting. In this case, the port used for that connection may be left open.

  If you suspect this is the case, to close the port, type the following command at the Terminal prompt:

  ObjectScript

  ```
  Close "|TCP|port"
  ```

  Where *port* is the port number to close.

  Alternatively, you can attempt to reconnect Caché to the Object Gateway server without closing the port by calling the **%LostConnectionCleanup()** method and **%Reconnect()** method of the gateway connection object in succession.

  If you prefer to automate the process of reconnecting to the Object Gateway server in the event of a disconnection, set the AttemptReconnect property for the gateway connection object to `true`.

- *Connection timeout errors*

  When writing a query, a .NET application may encounter an `<ALARM>` error due to a connection timeout error. The default timeout parameter can be overridden with the following command (assuming you have a command CMD):

  ```
  CMD.CommandTimeout=/NewTimeoutValue/
  ```

- *Out of Memory errors*

  Handling large amounts of data over the .NET Gateway may cause `System.OutOfMemoryException` errors. In this situation, raising the numbers of GDI Handles may help. You can raise the number of handles by changing the following registry entry:

  HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\GDIProcessHandleQuota

  The default is 10000 (2710 in Hex). It may help to set it to 20000 (4E20 in Hex). Larger Values like 25000 or 30000 are also possible.

  See the following Microsoft MDSN article for more information on this subject:

  http://social.msdn.microsoft.com/forums/en-US/wpf/thread/8ebf824a-5585-4b24-8e89-61f8be25d5c4/

5

Creating Proxy Classes

This chapter covers the following topics:

- Using the Studio .NET Gateway Wizard — The simplest way to generate proxy classes for a .NET assembly is to use the .NET Gateway Wizard plugin supplied with Studio.
- Generating Proxy Classes Programmatically — You can also generate proxy classes from within an ObjectScript program. The %Net.Remote.Gateway class contains the methods used to import class definitions from .NET assemblies and generate Gateway proxy classes.

See Using Wrapper Classes with .NET APIs for a description of the preferred method for importing third-party DLLs. See the Mapping Specification chapter for a detailed description of how .NET classes are mapped to Caché proxy classes.

Note: The .NET Gateway cannot generate proxy classes for .NET generic classes. It similarly cannot import .NET classes with generic subclasses or subinterfaces.

5.1 Using the Studio .NET Gateway Wizard

The following steps summarize the procedure:

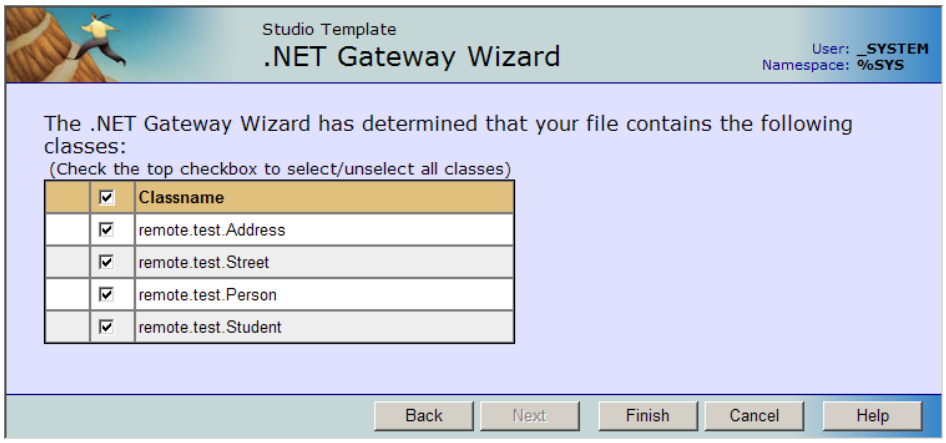
1. Start a .NET Gateway server. The server must be running before the Wizard can be used.
2. In Studio, select `Tools > Add-ins > Add-ins...` to display the list of available add-ins.
3. Select `.Net Gateway Wizard` from the `Add-ins` dialog and click `OK`. The `.Net Gateway Wizard` Studio template is displayed:

Figure 5–1: The .NET Gateway Wizard Studio Template

4. Fill out the form. The following items can be defined:
   - `Enter the path and name of a DLL assembly file:` — *Required*. DLL name.
   - `.NET Gateway server name / IP address:` — *Required*. IP address or name of the machine where the .NET Gateway server executable is located. The default is `"127.0.0.1"`.
   - `.NET Gateway server port:` — *Required*. Server port.
   - `Additional paths\assemblies to be used in finding dependent classes:` — *Optional*. Specify a list of assembly .dll files or directories, separated by semicolons.
   - `Exclude dependent classes matching the following prefixes:` — *Optional*. Specify a list of namespaces and class name prefixes, separated by semicolons.

5. The Wizard displays a list of available classes:

Figure 5–2: Selecting Classes to Import



Check the classes that you want to use, then click `Finish`.

5.2 Generating Proxy Classes Programmatically

This section discusses the following methods of the %Net.Remote.Gateway class:

- **%Import()** — imports .NET classes or assemblies from the .NET side and generates all the necessary proxy classes for the Caché side.
- **%GetAllClasses()** — returns a list of all public classes available in the specified assembly DLL.
- **%ExpressImport()** — combines calls to **%Connect()**, **%Import()**, and **%Disconnect()**.

See the %Net.Remote.Gateway Caché class documentation for a complete listing of all Gateway methods.

The **%Import()** method of the Gateway class sets off the following chain of sequential events:

1. The Caché session sends an import request.
2. Upon receiving the request, the .NET Gateway worker thread introspects the indicated .NET assemblies and classes.
3. The thread also loads dependent assemblies either from the local directory or from the Global Assembly Cache (GAC).
4. If it finds any .NET classes that are new or changed, or that have no proxy classes on the Caché side, the .NET Gateway worker thread generates new proxy classes for them.

5.2.1 %Import() Method

The **%Import()** method imports the given class and all its dependencies by creating and compiling all the necessary proxy classes. The **%Import()** method returns (in the ByRef argument *imported*) a list of generated Caché proxy classes. For details of how .NET class definitions are mapped to Caché proxy classes, see the "Mapping Specification" chapter in this guide.

%Import() is a onetime, startup operation. It only needs to be called the first time you wish to generate the Caché proxy classes. It is necessary again only if you recompile your .NET code and wish to regenerate the proxies.

Import Arguments

Before you invoke **%Import()**, prepare the *additionalClassPaths* and *exclusions* arguments. That is, for each argument, create a new %ListOfDataTypes object and call its **Insert()** method to fill the list. The optional *additionalClassPaths* argument can be used to supply additional path arguments, such as the names of additional assembly DLLs that contain the classes you are importing via the .NET Gateway. List elements should correspond to individual additional assembly DLL entries, which require the following format:

```
" rootdir\...\mydll.dll "
```

You can try to load an assembly from a directory outside of where DotNetGatewaySS.exe is running, but you might experience a load error for your assembly when you try to use a class in the assembly. InterSystems recommends that you put all local assemblies in the same directory as DotNetGatewaySS.exe. You can also specify assemblies in the GAC by using partial names for them, `System.Data`, for example.

Import Dependencies and Exclusions

While mapping a .NET class into an Caché proxy class and importing it into Caché, the .NET Gateway loops over all class dependencies discovered in the given .NET class, including all classes referenced as properties and in argument lists. In other words, the .NET Gateway collects a list of all class dependencies needed for a successful import of the given class, then walks that dependency list and generates all necessary proxy classes.

You can control this process by specifying a list of assembly and class name prefixes to exclude from this process. While this situation would be rare, it does give you some flexibility to control what classes get imported. The .NET Gateway automatically excludes a small subset of assemblies such as Microsoft.* assemblies.

5.2.2 %GetAllClasses() Method

This method returns, in the ByRef argument *allClasses*, a list of all public classes available in the assembly DLL specified by the first argument, *jarFileOrDirectoryName*.

5.2.3 %ExpressImport() Method

%ExpressImport() is a one-step convenience class method that combines calls to **%Connect()**, **%Import()**, and **%Disconnect()**. It returns a list of generated proxies. It also logs that list, if the *silent* argument is set to 0. The *name* argument is a semicolon-delimited list of classes or assembly DLLs.

6

Sample Code

The standard Caché installation includes a sample project that demonstrates how to generate and use Caché proxy classes with .NET code. This chapter describes how to compile and run the project, and provides a convenient copy of the Caché source code.

6.1 Compiling and Running the .NET Test Project

The project code consists of two parts:

- The RemoteTest .NET project located in <Cache-root>\dev\dotnet\samples\remote\test\
- The %Net.Remote.DotNet.Test class, which is part of the Caché class library.

The following sections describe how to compile and run the project.

6.1.1 Compiling the .NET Assembly

Open the RemoteTest project file (RemoteTest.csproj) in Visual Studio and compile it. The DotNetGatewaySamples.dll file will be created in ..remote\test\obj\Debug. This is the assembly that will be imported into Caché.

6.1.2 Creating and Running the Server

1. Create a .NET Gateway server description, as described in Defining a Gateway Server.
2. Start the server, as described in Running a Gateway Server.

6.1.3 Generating Caché Proxy Classes

To import the sample .NET classes, start a Terminal session and use the following syntax to run the **ExpressImport()** method of the %Net.Remote.ImportHelper class:

```
do ##class(%Net.Remote.ImportHelper).ExpressImport(gatewaySamplesDll,port)
```

where *gatewaySamplesDll* is a fully qualified path to DotNetGatewaySamples.dll (which you generated in step 1), and *port* is the port number of the Gateway server over which the proxy classes communicate with the .NET classes (which you started in step 3).

For example:

```
do ##class(%Net.Remote.ImportHelper).ExpressImport(
  "C:\InterSystems\Cache\dev\dotnet\samples\remote\test\obj\Debug\DotNetGatewaySamples.dll",
  "55000")
```

You can open Atelier to examine the imported classes, which will be located in the remote/test package in the %SYS namespace.

This step should be repeated whenever you have modified or recompiled your .NET classes.

6.1.4 Running the Caché DotNet.Test Examples

The %Net.Remote.DotNet.Test class includes two test methods, **Test()** and **TestArrays()**. For both methods, the *port* argument is the number of the port over which the proxy classes communicate with the .NET classes (as described in the previous section), and the *host* argument identifies the machine on which the .NET Gateway server is running. The *port* argument is required; the *host* argument is optional and defaults to `127.0.0.1` (the local machine).

Test()

The **Test()** method shows how to use the sample classes delivered with Caché. Use the following syntax to run it at the Terminal:

ObjectScript

```
do ##class(%Net.Remote.DotNet.Test).Test(port,host)
```

See The Test() Method for a listing of the ObjectScript code.

TestArrays()

The **TestArrays()** method demonstrates the use of .NET arrays. Use the following syntax to run it at the Terminal:

ObjectScript

```
do ##class(%Net.Remote.DotNet.Test).TestArrays(port,host)
```

See The TestArrays() Method for a listing of the ObjectScript code.

6.2 Source Code for the Test Class

This section provides a convenient copy of the code found in the %Net.Remote.DotNet.Test class, which includes two test methods, **Test()** and **TestArrays()**.

Class Definition

```
Class %Net.Remote.DotNet.Test Extends %RegisteredObject [ Abstract, ProcedureBlock ]
 {
    ClassMethod Test(port As %Integer, host As %String = "127.0.0.1") [ Final ]
    { ///...
    }
    ClassMethod TestArrays(port As %Integer, host As %String = "127.0.0.1") [ Final ]
    { ///...
    }
 }
```

Note: **Using the *%objlasterror* error status variable**

The Test class includes references to *%objlasterror*, which should be used as a debug resource only (for example, in development code that does not yet report errors properly), so that the underlying problem can be diagnosed and the offending code's error reporting can be corrected. It is appropriate for such code to kill *%objlasterror* whenever it uses an error status that is an expected condition and not a reportable error.

6.2.1 Test() Method

The **Test()** method demonstrates how to interact with the DotNetGatewaySamples.dll assembly.

Class Member

```
ClassMethod Test(port As %Integer, host As %String = "127.0.0.1") [ Final ]
 { ///...
 }
```

For convenience, the code in this section has been divided into the following parts:

- part 1 — connect to current namespace
- part 2 — create and populate a Student object
- part 3 — hashtable example
- part 4 — modify the hashtable
- part 5 — ArrayList and List examples
- part 6 — create and populate an Address object
- part 7 — change an Address
- part 8 — get an array of Strings
- part 9 — disconnect and catch errors

You can also examine this code by opening it in Atelier.

Test() method (part 1) — connect.

ObjectScript

```
Set %objlasterror="",$ZT="Error"
// connect to current namespace, use 2 second timeout
Set namespace=$Namespace,timeout=2
Set classPath=##class(%ListOfDataTypes).%New()
Set sampleDLL="dev/dotnet/bin/DotNetGatewaySamples.dll"
Set samplePath=$SYSTEM.Util.InstallDirectory()_sampleDLL
Do classPath.Insert(samplePath)

// get a connection handle and connect
Set gateway=##class(%Net.Remote.Gateway).%New()
Set status=gateway.%Connect(host,port,namespace,timeout,classPath)
If 'status Goto Error
```

Test() method (part 2) — create and populate a Student object.

ObjectScript

```
Set student=##class(remote.test.Student).%New(gateway,29,"976-01-6712")

// get, set Date
Write !,"setNextClass returned: "
Write student.setNextClass($zd($h,3),$zt($h),"White Hall",3.0,0)
Write !,"Next class on: ",$E(student.myGetNextClassDate(),1,10)
Write $E(student.getNextClassTime(),11,*),!,!
```

```
// set a String
Do student.mySetName("John","Smith")
// set an int

Do student.mySetID(27)
Write "Name: ",student.myGetName(),!
Write "ID: ",student.myGetID(),!
Write "SSN: ",student.getSSN(),!,!
Write "Static method execute: "
Write ##class(remote.test.Person).myStaticMethod(gateway),!,!
Do ##class(remote.test.Person).setstaticProperty(gateway,89)
Write "Static set/get: "
Write ##class(remote.test.Person).getstaticProperty(gateway),!!
```

Test() method (part 3) — hashtable example

ObjectScript

```
Set grades=##class(System.Collections.Hashtable).%New(gateway)
Do grades.Add("Biology",3.8)
Do grades.Add("French",3.75)
Do grades.Add("Spanish",2.75)
Do student.mySetGrades(grades)
Set grades=student.myGetGrades()

Write "Student has completed the following "
Write grades.getuCount()," classes:",!
Set keys=grades.getuKeys().GetEnumerator()
Set values=grades.getuValues().GetEnumerator()
while (keys.MoveNext())
{
   If (values.MoveNext())
      Write "  ",keys.getuCurrent()," ",values.getuCurrent(),!
}
Write !,"Highest grade: ",student.myGetHighestGrade()
```

Test() method (part 4) — modify the hashtable.

ObjectScript

```
Write !,"Now taking: Calculus, Chemistry, English Comp",!,!
Do student.setGrade("Calculus",3.5)
Do student.setGrade("Chemistry",3.92)
Do student.setGrade("English Comp",2.5)
Write "English Comp Grade: ",student.getGrade("English Comp"),!
Set grades=student.myGetGrades()
Write !,"Student has completed the following "
Write grades.getuCount()," classes:",!
Set keys=grades.getuKeys().GetEnumerator()
Set values=grades.getuValues().GetEnumerator()
while (keys.MoveNext())
{
   If (values.MoveNext())
      Write "  ",keys.getuCurrent()," ",values.getuCurrent(),!
}
Write !,"Highest grade now: "
Write student.myGetHighestGrade()
```

Test() method (part 5) — ArrayList and List examples.

ObjectScript

```
Set sports=##class(System.Collections.ArrayList).%New(gateway)
// Example of using Type.GetType; to use it, replace the above line with the
// following two lines: (also make sure to import System.Activator)
// Set arrayListClass=
//    ##class(System.Type).GetType(gateway,"System.Collections.ArrayList")
// Set sports=##class(System.Activator).CreateInstance(gateway,arrayListClass)
Do sports.Add("Basketball")
Do sports.Add("Tennis")
Do sports.Add("Running")
Do sports.Add("Swiming")
Do student.mySetFavoriteSports(sports)

 // set/get a list of Strings
Set list=student.myGetFavoriteSports()
Write !,"Student's favorite sports are: ",!
For i=0:1:list.getuCount()-1 {
   Write "  "_list.getuItem(i),!
}
```

Test() method (part 6) — create and populate an Address object.

ObjectScript

```
// set an object
Set home=##class(remote.test.Address).%New(gateway)
Set street=##class(remote.test.Street).%New(gateway)
Do street.setname("Memorial Drive")
Do street.setnumber("One")
Do home.mySetCity("San Diego")
Do home.mySetStreet(street)
Do home.mySetState("CA")
Do home.mySetZip("20098")
Do student.sethome(home)

// get an object
Write !,"Student's address: ",!
Set home2=student.gethome()
Write "  "_student.gethome().getstreet().getname(),!
Write "  "_home2.myGetCity()_", "_home2.myGetState()_" "_home2.myGetZip(),!,!
```

Test() method (part 7) — change an Address.

ObjectScript

```
Write "Change address",!
Set newHome=##class(remote.test.Address).%New(gateway)
Set newStreet=##class(remote.test.Street).%New(gateway)
Do newStreet.setnumber("456")
Do newStreet.setname("Del Monte")
Do newHome.mySetCity("Boston")
Do newHome.mySetState("MA")
Do newHome.mySetStreet(newStreet)
Do newHome.mySetZip("40480")
Set tempAddress=##class(remote.test.Address).%New(gateway)
Do student.getAddressAsReference(.tempAddress)
Write "City, before address change: "_tempAddress.myGetCity(),!
Do student.changeAddress(home,newHome)
Do student.getAddressAsReference(.tempAddress)
Write "City, after address change: "_tempAddress.myGetCity(),!
```

Test() method (part 8) — get an array of Strings.

ObjectScript

```
Set list=student.getAddressAsCollection()
Write !,"Student's new address is: ",!
Write "  "_list.GetAt(4),!
Write "  "_list.GetAt(1)_", "_list.GetAt(2)_" "_list.GetAt(3),!
Write !,"Old addresses:",!
Set oldAddresses=##class(%ListOfObjects).%New(gateway)
```

```
   Set newAdd=##class(remote.test.Address).%New(gateway)
   Set add2=student.getOldAddresses(home,.oldAddresses,.newAdd)
   For i=1:1:oldAddresses.Count() {
      Set oldAddress=oldAddresses.GetAt(i)
       Write !,"Address "_i_":",!
       Write oldAddress.getstreet().getnumber()
       Write " "_oldAddress.getstreet().getname(),!
       Write oldAddress.getcity()
       Write ", "_oldAddress.getstate()
       Write " "_oldAddress.getzip(),!
   }
   Write !,"Most recent Address:",!
   Write add2.getstreet().getnumber()_" "_add2.getstreet().getname(),!
   Write add2.getcity()
   Write ", "_add2.getstate()
   Write " "_add2.getzip(),!
   Write !,"Least Recent Address: ",!
   Write newAdd.getstreet().getnumber()_" "_newAdd.getstreet().getname(),!
   Write newAdd.getcity()
   Write ", "_newAdd.getstate()
   Write " "_newAdd.getzip(),!
```

Test() method (part 9) — disconnect and catch errors.

ObjectScript

```
   // Disconnect
   Do gateway.%Disconnect()
   Write !,"Test Successfully Completed"
   Quit
Error ; an error occurred
   Use 0
   If %objlasterror'=""
      { Write $system.OBJ.DisplayError(%objlasterror) }
   Else { Write $ze }
```

6.2.2 The TestArrays() Method

The **TestArrays()** method demonstrates the use of .NET arrays.

Class Member

```
ClassMethod TestArrays(port As %Integer, host As %String = "127.0.0.1") [ Final ]
 { ///...
 }
```

For convenience, the code in this section has been divided into the following parts:

- part 1 — connect.
- part 2 — create test Person object and test string arrays.
- part 3 — create and populate Address objects.
- part 4 — insert array of Address objects.
- part 5 — use a binary stream.
- part 6 — disconnect and catch errors.

You can also examine this code by opening it in Atelier.

TestArrays() method (part 1) — connect.

ObjectScript

```
Set %objlasterror="",$ZT="Error",namespace=$Namespace,timeout=2
Set classPath=##class(%ListOfDataTypes).%New()
Set sampleDLL="dev/dotnet/bin/DotNetGatewaySamples.dll"
Set samplePath=$SYSTEM.Util.InstallDirectory()_sampleDLL
Do classPath.Insert(samplePath)
Set gateway=##class(%Net.Remote.Gateway).%New()
Set status=gateway.%Connect(host,port,namespace,timeout,classPath)
If 'status Goto Error
```

TestArrays() method (part 2) — create test Person object and test string arrays.

ObjectScript

```
Set test=##class(remote.test.Person).%New(gateway)

// test simple string arrays
Set stringArray=##class(%ListOfDataTypes).%New()
Do stringArray.Insert("test string one")
Do stringArray.Insert("test string two")
Do stringArray.Insert("test string three")
Do stringArray.Insert("test string four")

// test simple string arrays
Do test.setStringArray(stringArray)
Set outStringArray=test.getStringArray()

For i=1:1:outStringArray.Count() {
    Write "String "_i_" : "_outStringArray.GetAt(i),!
}
```

TestArrays() method (part 3) — create and populate Address objects.

ObjectScript

```
// test array of objects
Set home=##class(remote.test.Address).%New(gateway)
Set street=##class(remote.test.Street).%New(gateway)
Do street.setname("Memorial Drive")
Do street.setnumber("One")
Do home.mySetCity("Cambridge")
Do home.mySetStreet(street)
Do home.mySetState("MA")
Do home.mySetZip("02142")

Set home2=##class(remote.test.Address).%New(gateway)
Set street2=##class(remote.test.Street).%New(gateway)
Do street2.setname("Santa Cruz Ave")
Do street2.setnumber("4555")
Do home2.mySetCity("San Diego")
Do home2.mySetStreet(street2)
Do home2.mySetState("CA")
Do home2.mySetZip("92109")
```

TestArrays() method (part 4) — insert array of Address objects.

ObjectScript

```
Set addressArray=##class(%ListOfObjects).%New()
Do addressArray.Insert(home)
Do addressArray.Insert(home2)

Do test.setAddressArray(addressArray)
Set addressArray=test.getAddressArray()
For i=1:1:addressArray.Count() {
   Set home=addressArray.GetAt(i)
    Write !,"Address "_i_":",!
    Write home.getstreet().getnumber()_" "_home.getstreet().getname(),!
    Write home.getcity()
    Write ", "_home.getstate()
    Write " "_home.getzip(),!
}
```

TestArrays() method (part 5) — use a binary stream.

ObjectScript

```
   // byte[] is mapped as %GlobalBinaryStream
   Write !,"Byte array test:",!
   Set byteStream=##class(%GlobalBinaryStream).%New()
   Do byteStream.Write("Global binary stream")

   // Note that byteStream is passed in by value, so any changes on the DotNet
   // side will be ignored. The next example will pass the stream by reference
   // meaning changes on the DotNet side will be reflected here as well
   Do test.setByteArray(byteStream)

   Set result=test.getByteArray()
   Write result.Read(result.SizeGet()),!

   Set readStream=##class(%GlobalBinaryStream).%New()
   // we need to 'reserve' a number of bytes since we are passing the stream
   // by reference (DotNet's equivalent is byte[] ba = new byte[max];)
   For i=1:1:50 Do readStream.Write("0")

   Set bytesRead=test.read(.readStream,50)
   Write readStream.Read(bytesRead),!
```

TestArrays() method (part 6) — disconnect and catch errors.

ObjectScript

```
   Do gateway.%Disconnect()
   Write !,"Test Successfully Completed"
   Quit
Error
   Use 0
   If %objlasterror'=""
      { Write $system.OBJ.DisplayError(%objlasterror) }
   Else { Write $ze }
 // end of method TestArrays()
```

7

Mapping Specification

This chapter describes the mapping between .NET objects and the Caché proxy classes that represent the .NET objects.

Important: Only classes, methods, and fields marked as `public` are imported.

This chapter describes mappings of the following types:

- Assembly and Class Names
- Primitives
- Properties
- Methods
- Constructors
- Constants
- OUT and REF Parameters
- .NET Arrays
- Recasting
- .NET Standard Output Redirection
- Restrictions

7.1 Assembly and Class Names

Assembly and class names are preserved when imported, except that each underscore (_) in an original .NET class name is replaced with the character u and each dollar sign ($) is replaced with the character d in the Caché proxy class name. Both the u and the d are case-sensitive (lowercase).

7.2 Primitives

Primitive types and primitive wrappers map from .NET to Caché as shown in the following table.

.NET	Caché
bool	%Library.Boolean
byte	%Library.Integer
char	%Library.String
double	%Library.Numeric
float	%Library.Double
int	%Library.Integer
long	%Library.Integer
sbyte	%Library.Integer
short	%Library.SmallInt
string	%Library.String
System.Boolean	%Library.Boolean
System.Byte	%Library.Integer
System.Char	%Library.String
System.DateTime	%Library.TimeStamp
System.Double	%Library.Numeric
System.Int16	%Library.SmallInt
System.Int32	%Library.Integer
System.Int64	%Library.Integer
System.SByte	%Library.Integer
System.Single	%Library.Double
System.String	%Library.String
System.UInt16	%Library.SmallInt
System.UInt32	%Library.Integer
System.UInt64	%Library.Integer
uint	%Library.Integer
ulong	%Library.Integer
ushort	%Library.SmallInt

7.3 Properties

The result of importing a .NET class is an ObjectScript abstract class. For each .NET property that does not already have corresponding getter and setter methods (imported as is), the .NET Gateway engine generates corresponding Object Script getter and setter methods. It generates Setters as set*XXX*, and getters as get*XXX*, where *XXX* is the property name. For

example, importing a .NET String property called Name results in a getter method **getName()** and a setter method **setName(%Library.String)**. The Gateway also generates set and get class methods for all static members.

7.4 Methods

After you perform the .NET Gateway import operation, all methods in the resulting Caché proxy class have the same name as their .NET counterparts, subject to the limitations described in the Method Names section. They also have the same number of arguments. The type for all the Caché proxy methods is %Library.ObjectHandle(). The .NET Gateway engine resolves types at runtime.

For example, the .NET method **test**:

```
public boolean checkAddress(Person person, Address address)
```

is imported as:

```
Method checkAddress(p0 As %Library.ObjectHandle,
                    p1 As %Library.ObjectHandle) As %Library.ObjectHandle
```

7.4.1 Overloaded Methods

While Caché Basic and ObjectScript do not support overloading, you can still map overloaded .NET methods to Caché proxy classes. This is supported through a combination of largest method cardinality and default arguments. For example, if you are importing an overloaded .NET method whose different versions take two, four, and five arguments, there is only one corresponding method on the Caché side; that method takes five arguments, all of %ObjectHandle type. You can then invoke the method on the Caché side with two, four, or five arguments. The .NET Gateway engine then tries to dispatch to the right version of the corresponding .NET method.

While this scheme works reasonably well, avoid using overloaded methods with the same number of arguments of similar types. For example, the .NET Gateway has no problems resolving the following methods:

```
test(int i, string s, float f)
test(Person p)
test(Person p, string s, float f)
test(int i)
```

However, avoid the following:

```
test(int i)
test(float f)
test(boolean b)
test(object o)
```

Tip: For better results using the .NET Gateway, use overloaded .NET methods only when absolutely necessary.

7.4.2 Method Names

Caché has a limit of 31 characters for method names. Ensure your .NET method names are not longer than 31 characters. If the name length is over the limit, the corresponding Caché proxy method name contains only the first 31 characters of your .NET method name. For example, if you have the following methods in .NET:

```
thisDotNetMethodHasAVeryLongName(int i)       // 32 characters long
thisDotNetMethodHasAVeryLongNameLength(int i) // 38 characters long
```

Caché imports only one method with the following name:

```
thisDotNetMethodHasAVeryLongNam              // 31 characters long
```

The .NET reflection engine imports the first one it encounters. To find out which method is imported, you can check the Caché proxy class code. Better yet, ensure that logging is turned on before the import operation. The .NET Gateway log file contains warnings of all method names that were truncated or not imported for any reason.

Each underscore (_) in an original method name is replaced with the character u and each dollar sign ($) is replaced with the character d. Both the u and the d are case-sensitive (lowercase). If these conventions cause an unintended overlap with another method name that already exists on the Caché side, the method is not imported.

Finally, Caché class code is not case-sensitive. So, if two .NET method names differ only in case, Caché only imports one of the methods and writes the appropriate warnings in the log file.

7.4.3 Static Methods

Caché projects .NET static methods as class methods in the Caché proxy classes. To invoke them from ObjectScript, use the following syntax:

```
// calls static .NET method staticMethodName(par1,par2,...)
Do ##class(className).staticMethodName(gateway,par1,par2,...)
```

7.5 Constructors

You invoke .NET constructors by calling **%New()**. The signature of **%New()** is exactly the same as the signature of the corresponding .NET constructor, with the addition of one argument in position one: an instance of the .NET Gateway. The first thing **%New()** does is to associate the proxy instance with the provided Gateway instance. It then calls the corresponding .NET constructor. For example:

```
// calls Student(int id, String name) .NET constructor
Set Student=##class(gateway.Student).%New(Gateway,29,"John Doe")
```

7.6 Constants

The .NET Gateway projects and imports .NET static final variables (constants) as Final Parameters. It preserves the names when imported, except that it replaces each underscore (_) with the character u and each dollar sign ($) with the character d. Both the u and the d are case-sensitive (lowercase).

For example, the following static final variable:

```
public const int DOTNET_CONSTANT = 1;
```

is mapped in ObjectScript as:

```
Parameter DOTNETuCONSTANT As INTEGER = 1;
```

From ObjectScript, access the parameter as:

```
##class(MyDotNetClass).%GetParameter("DOTNETuCONSTANT"))
```

7.7 OUT and REF Parameters

The .NET Gateway supports passing parameters by reference, by supporting the .NET *OUT* and *REF* parameters. Only objects may be used as *OUT* and *REF* parameters; scalar values are not supported. For this convention to work, you must preallocate a temporary object of the corresponding type. Then call the method and pass that object by reference. The following are some examples:

```
public void getAddressAsReference(out Address address)
```

To call this method from ObjectScript, create a temporary object; there is no need to set its value. Then call the method and pass the *OUT* parameter by reference, as follows:

ObjectScript

```
Set tempAddress=##class(remote.test.Address).%New(gateway)
Do student.getAddressAsReference(.tempAddress)
```

The following example returns an array of Address objects as an *OUT* parameter:

```
void getOldAddresses(out Address[] address)
```

To call the previous method from ObjectScript, use the following code:

ObjectScript

```
Set oldAddresses=##class(%ListOfObjects).%New(gateway)
Do person.getOldAddresses(.oldAddresses)
```

7.8 .NET Arrays

Arrays of primitive types and wrappers are mapped as %Library.ListOfDataTypes. Arrays of object types are mapped as %Library.ListOfObjects. Only one level of subscripts is supported.

The Gateway projects .NET byte arrays (byte[]) as %Library.GlobalBinaryStream. Similarly, it projects .NET char arrays (char[]) as %Library.GlobalCharacterStream. This allows for a more efficient handling of byte and character arrays.

You can pass byte and stream arrays either by value or by reference. Passing by reference allows changes to the byte or character stream on the .NET side visible on the Caché side as well. For example, using the following:

System.Net.Sockets.Stream.Read(byte[] buffer, int offset, int size)

in .NET:

```
byte[] buffer = new byte[maxLen];
int bytesRead = inputStream.Read(buffer,offset,maxLen);
```

The equivalent code in ObjectScript:

ObjectScript

```
Set readStream=##class(%GlobalBinaryStream).%New()
// we need to 'reserve' a number of bytes since we are passing the stream
// by reference (DotNet's equivalent is byte[] ba = new byte[max];)
For i=1:1:50 Do readStream.Write("0")
Set bytesRead=test.read(.readStream,50)
Write readStream.Read(bytesRead)
```

The following example passes a character stream by value, meaning that any changes to the corresponding .NET char[] is not reflected on the Caché side:

ObjectScript

```
Set charStream=##class(%GlobalCharacterStream).%New()
Do charStream.Write("Global character stream")
Do test.setCharArray(charStream)
```

7.9 Recasting

ObjectScript has limited support for recasting; namely, you can recast only at a point of a method invocation. However, since all Caché proxies are abstract classes, this should be sufficient.

7.10 .NET Standard Output Redirection

The .NET Gateway automatically redirects any standard .NET output in the corresponding .NET code to the calling Caché session. It collects any calls to System.out in your .NET method calls and sends them to Caché to display in the same format as you would expect to see if you ran your code from .NET. To disable this behavior and direct your output to the standard output device as designated by your .NET code (in most cases that would be the console), set the following global reference in the namespace where the session is running:

ObjectScript

```
Set ^%SYS("Gateway","Remote","DisableOutputRedirect") = 1
```

7.11 Restrictions

Important: Rather than aborting import, the .NET Gateway engine silently skips over all the members it is unable to generate. If you repeat the import step with logging turned on, Caché records all skipped members (along with the reason why they were skipped) in the `WARNING` section of the log file.

The .NET Gateway engine always makes an attempt to preserve assembly and method names, parameter types, etc. That way, calling an Caché proxy method is almost identical to calling the corresponding method in .NET. It is therefore important to keep in mind Caché Basic and ObjectScript restrictions and limits while writing your .NET code. In a vast majority of cases, there should be no issues at all. You might run into some Caché Basic or ObjectScript limits. For example:

- .NET method names should not be longer than 30 characters.
- You should not have 100 or more arguments.
- You should not try to pass String objects longer than 32K.
- Do not rely on the fact that .NET is case-sensitive when you choose your method names.
- Do not try to import a static method that overrides an instance method.
- The .NET Gateway cannot generate proxy classes for .NET generic classes. It similarly cannot import .NET classes with generic subclasses or subinterfaces.
- .NET Events are not supported — Caché code cannot be called from delegate notifications.

For details on Caché Basic and ObjectScript naming conventions, see Variables in *Using Caché ObjectScript*, Naming Conventions in *Using Caché Objects*, Identifiers and Variables in *Using Caché Basic*, and Rules and Guidelines for Identifiers in the *Caché Programming Orientation Guide*.