Using the Object Gateway for .NET

Version 2019.1
2019-03-19

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

Using the Object Gateway for .NET
InterSystems IRIS Data Platform Version 2019.1 2019-03-19

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

About This Book

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

This book contains the following chapters:

There is also a detailed Table of Contents.

1

Object Gateway Architecture

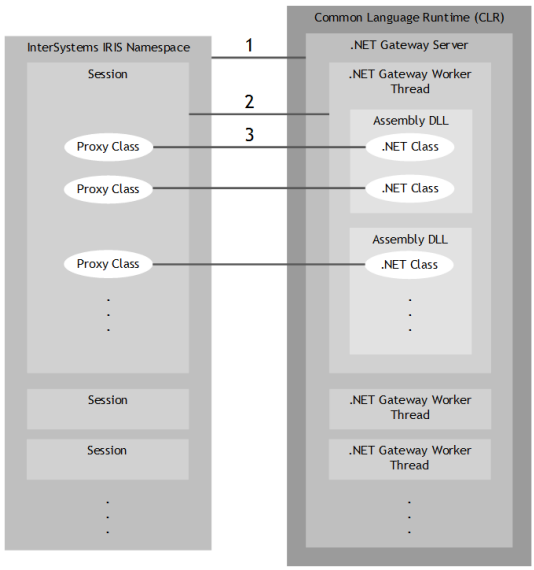
The InterSystems IRIS™ Object Gateway for .NET (which this book will usually refer to as simply “the Object Gateway”) provides an easy way for InterSystems IRIS to interoperate with Microsoft .NET Framework components. The Object Gateway can instantiate an external .NET object and manipulate it as if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

1.1 Using Proxy Classes

The external .NET object is represented within InterSystems IRIS by a proxy class. A proxy object looks and behaves just like any other InterSystems IRIS 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 InterSystems IRIS proxy object triggers the corresponding method of a .NET object inside the CLR.

The following diagram offers a conceptual view of InterSystems IRIS and the Object Gateway at runtime.

Figure 1–1: .NET Object Gateway Operational Model



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

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

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

- InterSystems IRIS sends a message over the TCP/IP connection to the Object Gateway worker thread. The message consists of the method name, parameters, and occasionally some other information.
- The Object 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 InterSystems IRIS proxy object over the same TCP/IP channel, and the proxy method returns the results to the InterSystems IRIS application.

1.2 Using Wrapper Classes with .NET APIs

In most cases, you will use the Object 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 InterSystems IRIS and the .NET framework very clean and eliminates many potential issues. Wrapper classes provide the following advantages:

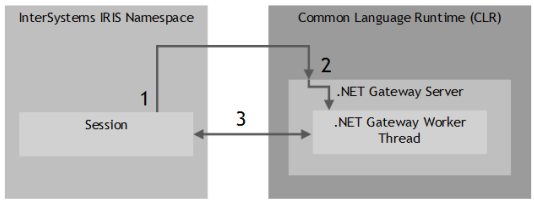
- *Fewer proxy classes* — When a large .NET DLL is imported directly, hundreds of proxy classes may be generated. Your application may actually need only a few of these classes. A significant amount of work may be required to research and define an exclusion list for the import. Defining a wrapper is 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.

1.3 Creating and Running an Object Gateway Server

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

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

Figure 1–2: Connecting to a .NET Object Gateway Worker Thread



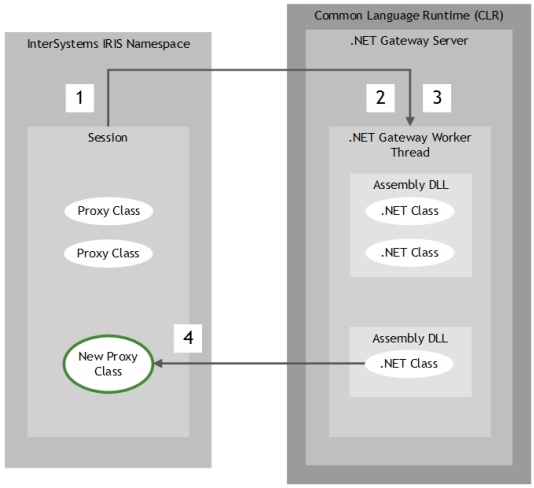
1. ObjectScript code sends a connection request.
2. Upon receiving the request, the Object Gateway server starts a worker thread in which the .NET class methods subsequently run.
3. The connection between this Object Gateway worker thread and the corresponding InterSystems IRIS session remains established 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 Object Gateway Server Properties for a detailed description of how to create an Object Gateway Server property definition, and Running an Object Gateway Server for details on how to start, connect, disconnect, and stop a server.

1.4 Importing Proxy Classes

InterSystems IRIS proxy classes are generated by sending a query to the Object 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 1–3: Importing .NET Classes



1. The InterSystems IRIS session sends an import request.
2. Upon receiving the request, the Object 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 InterSystems IRIS side, the Object Gateway worker thread returns the results of the introspection to the InterSystems IRIS session, which uses the information to generate new proxy classes.

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

1.5 The .NET Object Gateway API

The following classes provide most of the functionality used by your InterSystems IRIS Object Gateway applications:

- %Net.Remote.ObjectGateway — an ObjectGateway object contains the property settings required to run and monitor an instance of the Object Gateway Server. See Defining an Object Gateway Server for a detailed description.
- %Net.Remote.Service — a Service object controls the interface between an InterSystems IRIS namespace and an instance of the Object Gateway Server. See Running an Object Gateway Server.
- %Net.Remote.Gateway — a Gateway object controls the connection between ab InterSystems IRIS session and a worker thread within an instance of the Object 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 InterSystems IRIS Proxy Classes.

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

2

Setting Object Gateway Server Properties

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

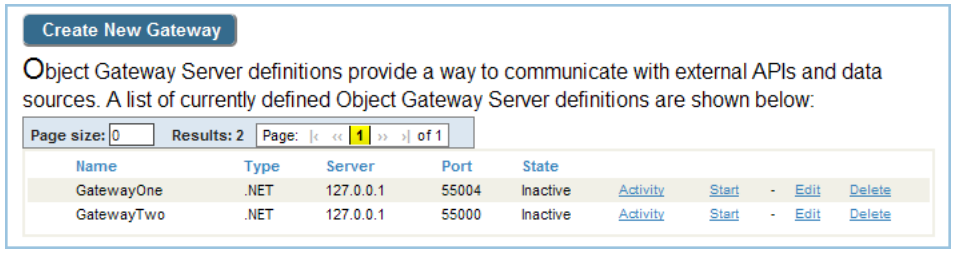
- Using the New Object Gateway Form — The simplest way to define a set of server properties is to use the New Object Gateway form in the Management Portal, which allows you to create and store a predefined ObjectGateway object.
- Defining Server Properties Programmatically — It is also possible to set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

2.1 Using the New Object Gateway Form

While it is possible to specify the settings for an Object 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].**

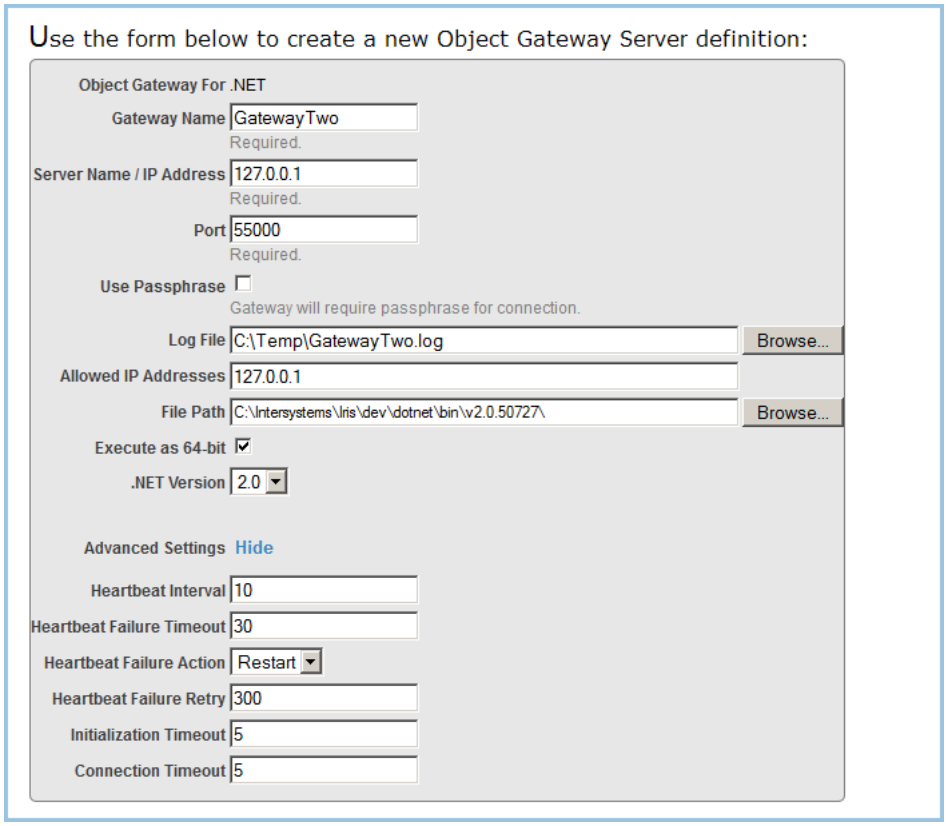
Figure 2–1: The Object Gateways Page



   You can use the options on this page to view an Object Gateway's log of recent activity, start or stop an Object Gateway, and create, edit, or delete an Object 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 2–2: 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 Object Gateway definition.
   - Server — *Required*. IP address or name of the machine where the Object Gateway server executable is located. The default is `"127.0.0.1"`.
   - Port — *Required*. TCP port number for communication between the Object Gateway server and the proxy classes in InterSystems IRIS™. Several methods of class Service (see "Running an Object Gateway Server") accept the port number as an optional parameter with a default value of `55000`.
   - PassPhrase — If this property is checked, the Object Gateway will require a passphrase to connect.
   - LogFile — Full pathname of the file used to log Object Gateway messages. These messages include acknowledgment of opening and closing connections to the server, and any difficulties encountered in mapping .NET classes to InterSystems IRIS 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 Object Gateway server executable is located. This is used to find the target executable and assemble the command to start the Object 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 Object 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 or 4.0) to be used (see "Object Gateway Server Versions"). Defaults to .NET 2.0.
- *Advanced settings* (hidden by default)
  - HeartbeatInterval — Number of seconds between each communication with the Object 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 Object Gateway is in failure state. If this value is smaller than the HeartbeatInterval property, the Object Gateway is in failure state every time the Object Gateway communication check fails. The maximum value is 86400 seconds (1 day). The default is 30 seconds.
  - HeartbeatFailureAction — Action to take if the Object Gateway goes into a failure state. Valid values are `"R"` (Restart) or `"N"`(None). The default action is Restart, which causes the Object Gateway to restart.
  - HeartbeatFailureRetry — Number of seconds to wait before retrying the HeartbeatFailureAction if the Object 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 Object 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 Object 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 Object Gateway Server definition should now be listed there.

2.2 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 Object Gateway (rather than an Object Gateway for Java).

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

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

// 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\IRIS\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).

2.3 Object Gateway Server Versions

Different versions of the Object Gateway Server assembly are provided for .NET 2.0 and 4.0. 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 InterSystems IRIS installation procedure does not install or upgrade any version of the .NET Framework.

3

Running an Object Gateway Server

Each Object Gateway server session consists of the following components:

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

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

3.1 Starting the Server

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

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

3.1.1 Starting a Server from the Command Prompt

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

Note: The Object Gateway server executable will normally be located in a default directory (see "Object 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 Object 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.

3.2 Connecting to a Server Thread

Connecting creates a %Net.Remote.Gateway object.

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

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

3.2.1 The Gateway.%Connect() Method

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

Argument	Description
host	Identifies the machine on which the Object Gateway server is running.
port	Port number over which the proxy classes communicate with the .NET classes.
namespace	InterSystems IRIS 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 Object Gateway. See the Import Arguments section for details using this argument.

3.3 Disconnecting and Stopping the Server

ObjectScript code that establishes an Object 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 Object Gateway server.

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

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

3.4 Monitoring and Debugging the Object Gateway

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

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

3.4.1 Error Checking

The Object Gateway provides error checking as follows:

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

In both cases, InterSystems IRIS 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:

```
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.

3.4.2 Troubleshooting

The following suggestions may help in certain situations:

- *Activate logging*

Should you encounter problems while using the Object 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 Object Gateway Server Properties ") when you start the Object Gateway.

- *Terminal session problems*

  Sometimes, while using the Object 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 Object 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:

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

  Where *port* is the port number to close.

- *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 Object 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/

4

Using Dynamic Object Gateways

This chapter describes how to implement Object Gateways using *dynamic proxy objects*. In previous releases, *static proxies* had to be generated separately, compiled, and stored in the class library just like any other class (see the next chapter, Creating Static Proxy Classes, for more information). Dynamic proxy objects are generated at runtime, and are all instances of a single class, %Net.Remote.Object.

- Introducing Dynamic Gateways — provides an overview of dynamic proxies.
- Creating and Using a Dynamic Object Gateway — describes how to use dynamic proxies and gives specific usage examples.
- Array Access — describes how .NET arrays are implemented as dynamic proxies.
- Gateway Reentrancy — describes how to ObjectScript and .NET code can run safely using the same connection and context.
- Example Classes — contains detailed information on the classes used as examples in this chapter.

Note: **Dynamic Proxies vs. Static Proxies**

Static proxies have some useful features, and will continue to be supported, but dynamic proxies are recommended for most new development.

4.1 Introducing Dynamic Gateways

All versions of the Object Gateway can instantiate an external .NET object and manipulate it as if it were a native InterSystems IRIS object. This is accomplished by creating a proxy object that provides all of the same methods and properties as the corresponding .NET object. Any call on the proxy object triggers the corresponding method or property of the .NET object. There are currently two kinds of proxy objects:

- *Static proxies* are standard ObjectScript classes, which must be individually generated and compiled before they can be used. If a new version of the .NET class changes the interface in any way, the corresponding proxy class must be regenerated and recompiled to match it.
- *Dynamic proxies* are created at runtime by introspecting the current version of the .NET class. All dynamic proxies are instances of %Net.Remote.Object. Although each Object instance appears to have the same methods and properties as the corresponding .NET object, it actually uses an internal map of the interface to simulate these features.

In use, static and dynamic proxies are identical except for the way they are instantiated. For example:

```
// Static proxy: just call %New() on a pregenerated class
  set newFruit = ##class(Fruit).%New()

// Dynamic proxy: create an Object instance and specify the class to introspect
  set newFruit = ##class(%Net.Remote.Object).%New(gateway,"Fruit")
```

4.2 Creating and Using a Dynamic Object Gateway

This section describes the complete procedure for creating and running an Object Gateway that uses dynamic proxies:

- Start the Object Gateway server — dynamic proxies can't be created without querying the server at runtime.
- Get the server definition and test the server — the server definition is required for methods that monitor and control the server.
- Define the location of your .NET classes and create a Gateway connection — connection information must include the path to each DLL you intend to use.
- Create a proxy and call some methods — after they've been created, static and dynamic proxies work the same way.
- Disconnect and shut down — the server will continue to run until you explicitly stop it.

Start the Object Gateway server

Since dynamic proxies are created at runtime, you need access to a server that can introspect .NET classes and return the information needed to generate the proxies.

To start the gateway, open the Management Portal and go to the Object Gateways Page, **[System Administration] > [Configuration] > [Connectivity] > [Object Gateways]**. This page displays all ObjectGateway descriptions that have been persisted to the database, and provides a convenient way to control, monitor, and modify the servers they describe. The following example displays information on the *NetGate* server, and indicates that an instance of that server is currently running:

The examples in this chapter will assume that the server named *NetGate* has been defined (see "Creating the NetGate Server Definition" for details).

Get the server definition and make sure the server is running

The Service class provides methods to start, stop, and test the server. Most of these methods require the server definition contained in an ObjectGateway object. In the following example, Service.**OpenGateway()** returns an ObjectGateway instance defining the server named *NetGate*. Service.**IsGatewayRunning()** uses the information to test the current state of the server, and Service.**StartGateway()** can start it if necessary.

```
try {
// Get a ObjectGateway instance that defines the NetGate server
  set st = ##class(%Net.Remote.Service).OpenGateway("NetGate",.OG)
  write !, "Using "_OG.Name_" server definition"

// Test to make sure the server is running. If not, start it
  if ('##class(%Net.Remote.Service).IsGatewayRunning(OG.Server,OG.Port,,.status)) {
    // Instantiate the server on the host machine
    set status = ##class(%Net.Remote.Service).StartGateway(OG.Name)
  }
}
catch ex {
  write !,$system.OBJ.DisplayError(ex.AsStatus())
}
```

Note: **Exception Handling**

Your ObjectScript code should always use `try`/`catch` for Object Gateway error handling. When the class running on the host machine encounters an exception, the dynamic proxy will throw %Net.Remote.Exception. In addition to standard status details, the Exception object contains call stack information, which is returned by calling method Exception.**StackAsArray(.array)**. The information may be different between Java and .NET.

Define the location of your .NET classes and create a Gateway connection

Each dynamic proxy will access the corresponding .NET object through a Gateway object connected to the server. The Gateway object needs a path to each DLL containing a class you want to use. In this example, the location of the DLL containing class Fruit is specified in the last argument to Gateway.**%Connect()**.

```
// define the location of the file containing the Fruit class
set fruitPath = ##class(%ListOfDataTypes).%New()
do fruitPath.Insert("C:\Dev\Fruit.dll")

// Connect a Gateway instance to server NetGate on the host machine
set GW = ##class(%Net.Remote.Gateway).%New()
set st = GW.%Connect(OG.Server, OG.Port, "USER",,fruitPath)
```

This example only adds one path, but you can insert a separate path for each DLL you want to use.

Create a dynamic proxy for .NET class Fruit and call some methods

In this example, an new instance of Object is created, specifying Fruit as the .NET class to introspect. The resulting dynamic proxy will communicate with the .NET object through Gateway instance *GW* (created in the previous example). Once the dynamic proxy is instantiated, it works just like a static proxy.

```
// Use GW connection to create a proxy for the Fruit class
set proxyFruit = ##class(%Net.Remote.Object).%New(GW,"Fruit")

write !,proxyFruit.id()
write !,"Current fruit preference: ",!,"   "_proxyFruit.getFruit()
do proxyFruit.setFruit("Jujube")
write !,"Fruit preference has been reset: ",!,"   "_proxyFruit.getFruit()
```

Prints:

```
This .NET class displays your favorite fruit.
Current fruit preference: UglyFruit
Fruit preference has been reset: Jujube
```

Disconnect and shut down the server

The connection remains open and the server continues to run until you explicitly disconnect and shut down. In this example, the Gateway.**%Disconnect()** method breaks the connection, but the server won't release the port it uses until the call to Service.**StopGatewayObject()** stops it.

```
// Disconnect from the server
  set st = GW.%Disconnect()

// Stop the server
  set st =  ##class(%Net.Remote.Service).StopGatewayObject(OG)
```

4.3 Array Access

When a property is declared as an array, it is projected to the proxy as an instance of %Net.Remote.Object. To create a new array, call the Object.**%New()** method and use array class name syntax in place of the regular class name. Arguments are used as the initial values for the array. Here are some examples:

Array initialization

Array can be initialized by rank or elements:

```
// by rank
  set newArray = ##class(%Net.Remote.Object).%New(gateway,"String[2]")

// by elements
  set newArray = ##class(%Net.Remote.Object).%New(gateway,"String[]","A","B","C")
```

When rank and elements are both specified, rank must match element count.

```
// both rank and elements specified
  set newArray = ##class(%Net.Remote.Object).%New(gateway,"int[3]",11,22,33)
```

.NET multidimensional arrays are supported. They can only be initialized by rank:

```
// Create a two dimensional array by rank
  set newArray = ##class(%Net.Remote.Object).%New(gateway,"String[2,3]")
```

Getting and setting array elements

Besides supporting all the normal properties and methods available on an array object, Dynamic Gateway supports two special methods for accessing array elements:

- `%get(index)` — `object.%get(index)` is translated to `object[index]`
- `%set(index,value)` — `object.%set(index,value)` is translated to `object[index] = value`.

For example:

```
// Create a two dimensional array and populate it
  set arr = ##class(%Net.Remote.Object).%New(gateway,"String[2,3]")
  for x=0:1:1 {for y=0:1:2 {set i=x*3+y  do arr.%set(x,y,$CHAR(65+i)) }}

  write !,"Array length is" _arr.Length_" and rank is "_arr.Rank
  for x=0:1:1 {for y=0:1:2 {write "("_x_","_y_")="_arr.%get(x,y)_"  " }}
```

Prints:

```
Array length is 6 and rank is 2
(0,0)=A  (0,1)=B  (0,2)=C  (1,0)=D  (1,1)=E  (1,2)=F
```

4.4 Gateway Reentrancy

Reentrancy means that when InterSystems IRIS code sends a request to .NET, the .NET code can respond with a request of its own using the same physical connection and context as the original request. The following is a typical call sequence:

- Starting from the InterSystems IRIS side, a connection is made to .NET using a %Net.Remote.Gateway object, allowing us to create proxy objects that can make calls to .NET methods.
- From a proxy, the .NET Gateway.GatewayContext.**getIRIS()** method can be called to create an instance of ADO.IRIS. The IRIS object uses the same connection and context as the proxy, and provides access to all the methods for invoking Native API calls (see "Using the InterSystems Native API for .NET").
- The IRIS object can call back into InterSystems IRIS, where it can get and set global values or call classmethods and functions. Making one of these calls puts us back in InterSystems IRIS code again.
- From here, we can make another inner call back to .NET using the same Gateway connection. Method `##class(%Net.Remote.Gateway).%GetContextGateway()` returns the Gateway object for the current context.
- Calls back and forth between InterSystems IRIS and .NET can go on indefinitely, always using the same connection and context.

Getting an .ADO.IRIS object from GatewayContext()

The following example gets an ADO.IRIS object in .NET, and uses it to read a global value from the InterSystems database:

```
IRIS native = GatewayContext.getIRIS();
// Read the value of global ^GlobalName("value2")
if ( native != null ) {
  String globalVal = native.GetString("GlobalName","value2");
}
```

If the code is not invoked inside a valid Gateway context, the **getIRIS()** method will return a null object.

Getting a Gateway context object from within Native API calls

This example gets a %Net.Remote.Gateway object for the current context, and uses it to create a proxy for a .NET object in the same context:

```
set gateway = ##class(%Net.Remote.Gateway).%GetContextGateway()
if gateway'=$$$NULLOREF {
  set objFruit = ##class(%Net.Remote.Object).%New(gateway,"Fruit")
}
```

If the context is not valid, the Gateway object will be $$$NULLOREF.

Note: In general, reentrancy support is not compatible with prior versions of the Object Gateway .

4.5 Example Classes

This section provides detailed descriptions of the two sample classes used in this chapter:

- The .NET Fruit Class is the trivial .NET class from which the proxy objects are projected.
- Creating the NetGate Server Definition is an instance of ObjectGateway that defines the Object Gateway server named *NetGate*.

4.5.1 The .NET Fruit Class

This section lists the Fruit class used by most of the examples in this chapter. This trivial .NET class has all the features needed to demonstrate a simple proxy object: static class method **id()**, and property accessors **getFruit()** and **setFruit()**.

The Fruit class

```
using System;
namespace demo {
  public class Fruit {
    public static String id() {
      return "This .NET class displays your favorite fruit.";
    }
    public String fruit;
    public Fruit () {
      fruit = "Uglyfruit";
    }
    public void setFruit(String newFruit) {
      fruit = newFruit;
    }
    public String getFruit() {
      return "My favorite fruit is "+fruit;
    }
    public static void main(String []args) {
      System.out.println("\nGenerating output from .NET class Fruit: ");
      Fruit myFruit = new Fruit ();
      System.out.println(myFruit.getFruit());
```

```
    }
  }
}
```

4.5.2 Creating the NetGate Server Definition

This section demonstrates how to create the *NetGate* Object Gateway server definition used by most of the examples in this chapter.

The Management Portal provides convenient interactive tools for creating and controlling Object Gateway servers (as mentioned earlier in this chapter), but this example will demonstrate what those tools are actually doing in the background. You can create and run a server with a few lines of ObjectScript code, specifying exactly the same information you would enter in the Management Portal.

The properties of an Object Gateway server are defined in an instance of %Net.Remote.ObjectGateway. The following code specifies properties for a .NET-based server named *NetGate*, persists the server definition in the InterSystems IRIS database, and then starts an instance of the server on the host machine.

Define server properties in an instance of ObjectGateway

This server will run on the same machine as InterSystems IRIS (since the *Server* property specifies `localhost` as the host machine), but this is not a requirement. The host machine only needs a copy of the server executable.

```
  set OG =  ##class(%Net.Remote.ObjectGateway).%New()

// Properties used in all server definitions
  set OG.Type = "2"   // Defines a .NET server
  set OG.Name  = "NetGate"
  set OG.Server  = "127.0.0.1"
  set OG.Port  = "55602" // port number must be unique

// .NET-only properties
  set OG.AllowedIPAddresses = "127.0.0.1""
  set OG.FilePath  = "C:\InterSystems\IRIS\dev\dotnet\bin\v4.5\"
  // keep Exec64 default value
  set OG.DotNetVersion = "4.5"
```

The Object Gateway needs the *FilePath* and *DotNetVersion* paths to execute the command that will instantiate the server on the host machine. They are optional in this example (where `localhost` is the host machine), since they just specify the default paths to a version of the DotNetGatewaySS64.exe executable (installed with InterSystems IRIS) and the corresponding version of .NET.

Save the server definition and run the server

Calling ObjectGateway.**%Save()** adds this definition to the list stored in the InterSystems IRIS database. When the server definition is persisted in the database, it will be listed on the Object Gateway Page in the Management Portal.

```
  write "Saving "_OG.Name
  set status = OG.%Save()
```

The definition can now be used to construct and run the command that will start an instance of the server on the host machine. You can start the server either by clicking `Start` on the Object Gateway Page, or by calling the %Net.Remote.Service.**StartGateway()** method as demonstrated below.

```
// Instantiate the server on the host machine
   set status = ##class(%Net.Remote.Service).StartGateway(OG.Name)
```

5

Creating Static Proxy Classes

This chapter discusses how to generate *static proxy objects*, which are precompiled and stored in the class library just like any other class. See the previous chapter, Using Dynamic Object Gateways, for information about *dynamic proxies*, which are created at runtime. The following topics are covered here:

- Using the Studio .NET Gateway Wizard — The simplest way to generate proxy classes for a .NET assembly is to use the .NET Object Gateway Wizard plug-in 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 Object 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 InterSystems IRIS™ proxy classes.

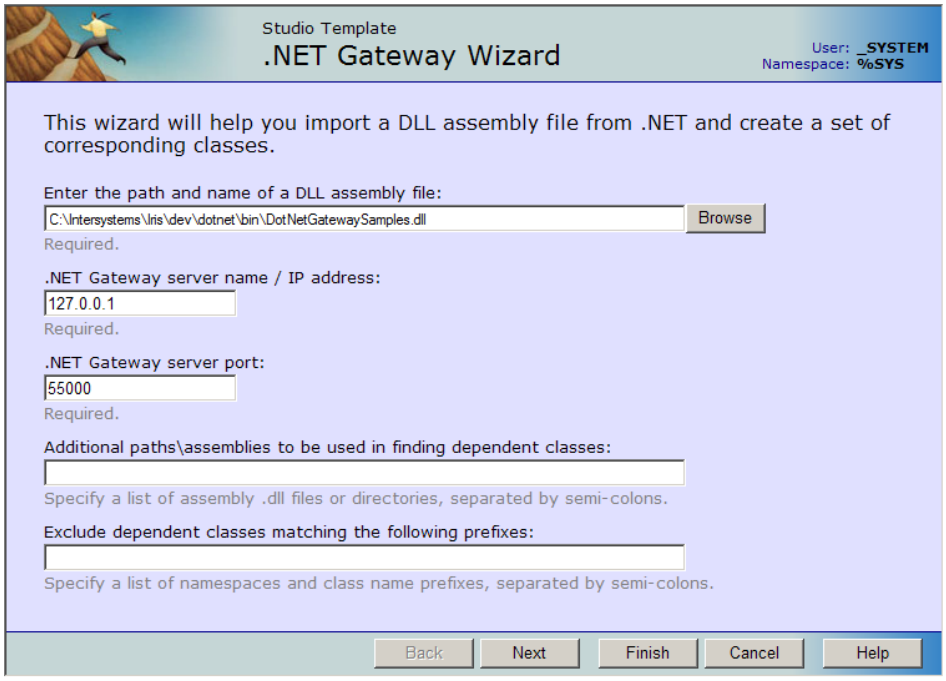
Note: The Object Gateway cannot generate proxy classes for .NET generic classes. It similarly cannot import .NET classes with generic subclasses or subinterfaces.

5.1 Using the Studio .NET Object Gateway Wizard

The following steps summarize the procedure:

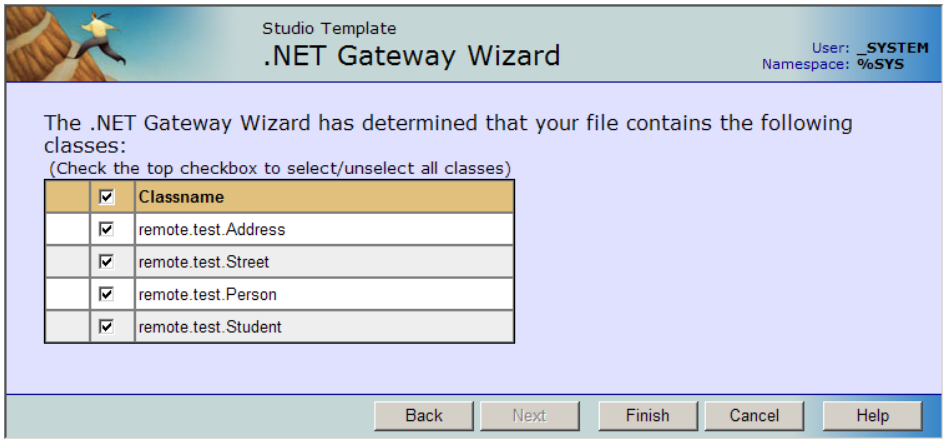
1. Start a .NET Object Gateway server. The server must be running before the Wizard can be used.
2. In Studio, select `Tools > Add-ins > Add-ins...` to display the list of available add-ins.
3. Select `.Net Gateway Wizard` from the `Add-ins` dialog and click `OK`. The `.Net Gateway Wizard` Studio template 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 Object 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 InterSystems IRIS 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 InterSystems IRIS class documentation for a complete listing of all Object Gateway methods.

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

1. The InterSystems IRIS session sends an import request.
2. Upon receiving the request, the Object 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 InterSystems IRIS side, the Object 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 InterSystems IRIS proxy classes. For details of how .NET class definitions are mapped to InterSystems IRIS 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 InterSystems IRIS 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 Object 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 InterSystems IRIS proxy class and importing it into InterSystems IRIS, the Object 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 Object 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 Object 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 InterSystems IRIS™ installation includes a sample ObjectScript project that demonstrates how to generate and use InterSystems IRIS proxy classes with .NET code. This chapter describes how to compile and run the project, and provides a convenient copy of the ObjectScript source code.

6.1 Compiling and Running the .NET Test Project

The project code consists of two parts:

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

6.1.2 Creating and Running the Server

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

6.1.3 Generating InterSystems IRIS 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 Object 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\IRIS\dev\dotnet\samples\remote\test\obj\Debug\DotNetGatewaySamples.dll",
  "55000")
```

The imported classes 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 InterSystems IRIS DotNet.Test Examples

The %Net.Remote.DotNet.Test class includes two ObjectScript 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 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 InterSystems IRIS. Use the following statement to run it at the Terminal:

```
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 statement to run it at the Terminal:

```
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 ObjectScript test methods, **Test()** and **TestArrays()**.

```
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.

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

The ObjectScript method code is listed here under the following section headings:

- 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

The original source code for the %Net.Remote.DotNet.Test class is available in the InterSystems IRIS class library under in the %SYS namespace.

Test() method (part 1) — connect.

```
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.

```
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

```
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.

```
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.

```
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.

```
        // 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.

```
        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.

```
        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.

```
        // 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.

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

The ObjectScript method code is listed here under the following section headings:

- 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.

The original source code for the %Net.Remote.DotNet.Test class is available in the InterSystems IRIS class library under in the %SYS namespace.

TestArrays() method (part 1) — connect.

```
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.

```
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.

```
        // 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.

```
        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.

```
        // 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.

```
        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 InterSystems IRIS™ 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 InterSystems IRIS 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 InterSystems IRIS as shown in the following table.

.NET	InterSystems IRIS
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 Object 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 Object Gateway also generates set and get class methods for all static members.

7.4 Methods

After you perform the Object Gateway import operation, all methods in the resulting InterSystems IRIS 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 InterSystems IRIS proxy methods is %Library.ObjectHandle(). The Object 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 ObjectScript does not support overloading, you can still map overloaded .NET methods to InterSystems IRIS 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 InterSystems IRIS side; that method takes five arguments, all of %ObjectHandle type. You can then invoke the method on the InterSystems IRIS side with two, four, or five arguments. The Object 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 Object 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 Object Gateway, use overloaded .NET methods only when absolutely necessary.

7.4.2 Method Names

InterSystems IRIS 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 InterSystems IRIS 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
```

InterSystems IRIS 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 InterSystems IRIS proxy class code. Better yet, ensure that logging is turned on before the import operation. The Object 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 InterSystems IRIS side, the method is not imported.

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

7.4.3 Static Methods

InterSystems IRIS projects .NET static methods as class methods in the InterSystems IRIS 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 Object Gateway. The first thing **%New()** does is to associate the proxy instance with the provided Object 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 Object 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 Object 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:

```
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:

```
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 Object 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 InterSystems IRIS 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:

```
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 InterSystems IRIS side:

```
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 InterSystems IRIS proxies are abstract classes, this should be sufficient.

7.10 .NET Standard Output Redirection

The Object Gateway automatically redirects any standard .NET output in the corresponding .NET code to the calling InterSystems IRIS session. It collects any calls to System.out in your .NET method calls and sends them to InterSystems IRIS 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:

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

7.11 Restrictions

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

The Object Gateway engine always makes an attempt to preserve assembly and method names, parameter types, etc. That way, calling an InterSystems IRIS proxy method is almost identical to calling the corresponding method in .NET. It is therefore important to keep in mind 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 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 Object 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 — InterSystems IRIS code cannot be called from delegate notifications.

For details on ObjectScript naming conventions, see Variables in *Using ObjectScript*, Naming Conventions in *Defining and Using Classes*, and Rules and Guidelines for Identifiers in the *Orientation Guide for Server-Side Programming*.

8

Using the Object Gateway in a Production

This chapter describes how to use the Object Gateway within a production.

There are different ways to invoke the basic Object Gateway **Start**, **Connect**, **Import**, **Disconnect**, and **Stop** commands. Practical approaches to this functionality include:

- Creating a Business Service
- Calling Business Service Methods
- Creating a Business Operation
- Calling API Methods
- Using the Command Prompt
- Using the Object Gateway Wizard

This chapter describes each approach and explains how to work with it. Related topics include:

- Error Checking
- Troubleshooting

8.1 Creating a Business Service

While it is possible to start the Object Gateway server from the command prompt, the simplest way to use the Object Gateway with a production is to configure the EnsLib.DotNetGateway.Service class as a business service within the production. You can only do this if the Object Gateway server is on the local machine where you are running the production.

Otherwise, you need to start the Object Gateway server from the command prompt. For details, see the Using the Command Prompt section.

To configure the EnsLib.DotNetGateway.Service class as a business service:

1. From the Management Portal main menu, choose **Productions**.
2. Find your production in the list and click **Configure** beside its name.
3. Click **Add Service** to start the Business Service Wizard.
4. Click **Other** and in the **ServiceClass** list, click **EnsLib.DotNetGateway.Service**.

5. Click **OK** to display the updated production diagram that now contains the Object Gateway business service. Click the **EnsLib.DotNetGateway.Service** box to configure it.

The wizard fills in the associated Object Gateway adapter class. The Business Service Settings for the Object Gateway section lists the configurable settings.

8.1.1 Business Service Settings for the Object Gateway

Dot Net Server

IP address or name of the machine where the Object Gateway server executable is located.

Port

TCP port number for communication between the Object Gateway server and the proxy classes in the production. The default is `55000`.

File Path

Location of the Object Gateway server executable. It is used to find the target executable and assemble the command to start the Object Gateway on a local server. If you do not specify this setting, the service uses the default directory ...\Dev\dotnet\bin\ under the installation directory.

Allowed IP Addresses

IP addresses allowed to connect to the Object Gateway server. If this setting is `0.0.0.0` (default) or `""`, any system (local or remote) may connect; otherwise any listed IP addresses are allowed to connect.

Exec64

If you select this check box, the business service uses the 64–bit version of the Object Gateway executable. Otherwise, it uses the 32–bit version of the executable. This setting is available only on Windows 64-bit platforms.

.NET Version

Select the .NET version to use: 2.0 or 4.0.

Log File

Full pathname of the file where the Object Gateway logs messages. These messages include acknowledgment of opening and closing connections to the server, and any difficulties encountered in mapping .NET classes to proxy classes.

Heartbeat Interval

Number of seconds between each communication with the Object Gateway to check whether it is active. When enabled, the minimum value is 5 seconds and the maximum value is 3600 seconds (1 hour). The default is 10 seconds. A value of 0 disables this feature.

Heartbeat Failure Timeout

Number of seconds without responding to the heartbeat, to consider that the Object Gateway is in failure state. If this value is smaller than the HeartbeatInterval property, the gateway is in failure state every time the Object Gateway communication check fails. The maximum value is 86400 seconds (1 day). The default is 30 seconds.

Heartbeat Failure Action

Action to take if the Object Gateway goes into a failure state. Setting it to Restart (default) causes the Object Gateway to restart. Setting it to Alert generates an alert entry in the Event Log. This is independent of the **Alert on Error** setting.

Heartbeat Failure Retry

Time to wait before retrying the HeartbeatFailureAction if the Object Gateway server goes into failure state, and stays in failure state. The default is 300 seconds (5 minutes). 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.

Once you have added and configured the Object Gateway business service, it automatically manages the Object Gateway as follows:

- When the production starts, the Object Gateway business service starts an instance of the Object Gateway server, using the settings that you specify on the configuration page.
- When the production receives a signal to stop, the Object Gateway business service attaches to the Object Gateway server and instructs it to stop, as well.

For more information, see the EnsLib.DotNetGateway.Service entry in the *Class Reference*.

8.2 Calling Business Service Methods

The Object Gateway business service provides methods that you can use to start, connect to, and stop the Object Gateway engine. You can call the following methods from the production code after you have configured the Object Gateway business service as a member of the production:

- StartGateway()
- ConnectGateway()
- StopGateway()

See the EnsLib.DotNetGateway.Service entry in the *Class Reference* for details on these methods.

8.2.1 StartGateway() Method

```
EnsLib.DotNetGateway.Service:StartGateway(pFilePath As %String,
     pPort As %String,
     pAllowedIPAddresses As %String,
     pLogfile As %String = "",
     ByRef pDevice As %String = "",
     pServer As %String = "127.0.0.1",
     pCmdLine As %String = "")
```

This class method starts the Object Gateway server using the specified arguments. If *pLogFile* specifies a valid file name, then messages regarding gateway activities are written to this file. These messages include acknowledgment of opening and closing connections to the server, and difficulties encountered (if any) in mapping .NET classes to the production proxy classes.

8.2.2 ConnectGateway() Method

```
EnsLib.DotNetGateway.Service:ConnectGateway(pEndpoint As %String,
     ByRef pGateway As %Net.Remote.Gateway,
     pTimeout As %Integer = 5,
     pAdditionalPaths As %String = "")
```

This class method connects to the Object Gateway server at the specified *pEndpoint* (hostname:port:namespace).

8.2.3 StopGateway() Method

```
EnsLib.DotNetGateway.Service:StopGateway(pPort As %String,
     pServer As %String = "127.0.0.1",
     pTimeout As %Integer = 5)
```

This class method connects to the Object Gateway server and shuts it down.

8.3 Creating a Business Operation

An abstract business operation is available as a base for building Object Gateway oriented business operations for productions. You can simply subclass the abstract class EnsLib.DotNetGateway.AbstractOperation and implement the appropriate message handlers.

Call the **GetConnection()** method to verify there is a valid Object Gateway connection. For example:

```
 Set tSC = ..GetConnection(.tGateway)
     If $$$ISOK(tSC) {
        // Start using the Object Gateway connection object tGateway
      ...
     }
```

This method returns a private gateway connection object to be used with the proxy classes.

You can configure the Object Gateway IP address and port in the business operation settings when you add the business operation to the production. Note that the connection to the Object Gateway instance is made during **OnInit()** and closed in **OnTearDown()**. You must override these methods in the business operation class to implement your own setup and tear down procedures.

See the EnsLib.DotNetGateway.AbstractOperation entry in the *Class Reference* for details on these methods and also the AdditionalPaths, ConnectTimeout, DotNetServer, and Port properties.

8.4 Calling API Methods

In addition to using connect, disconnect, and stop from the business service, the following methods are also available in the %Net.Remote.Gateway class. You can use them when the business service model is not appropriate for your situation:

The %Net.Remote.Gateway class provides the following types of methods:

- API methods that let you %Connect to the Object Gateway server, %Disconnect from it, and %Shutdown the Object Gateway server.
- The %Import method, which imports .NET classes or assemblies from the .NET and generates all the necessary proxy classes for the InterSystems IRIS™ side.
- The %ExpressImport method, which combines calls to **%Connect**, **%Import**, and **%Disconnect**.
- The utility method %GetAllClasses.

8.4.1 %Connect() Method

```
Method %Connect(host As %String,
                port As %Integer,
                namespace As %String,
                timeout As %Numeric = 5,
                additionalClassPaths As %ListOfDataTypes = "")
    As %Status [ Final ]
```

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

Argument	Description
host	Identifies the machine on which the Object Gateway server is running.
port	Port number over which the proxy classes communicate with the .NET classes.
namespace	interoperability-enabled 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 Object Gateway. See the Import Arguments section for details using this argument.

8.4.2 %Disconnect() Method

```
Method %Disconnect() As %Status [ Final ]
```

The **%Disconnect()** method closes a connection to the Object Gateway engine.

8.4.3 %Shutdown() Method

```
Method %Shutdown() As %Status [ Final ]
```

The **%Shutdown()** method shuts down the Object Gateway engine.

8.4.4 %Import() Method

```
Method %Import(class As %String,
               ByRef imported As %ListOfDataTypes,
               additionalClassPaths As %ListOfDataTypes = "",
               exclusions As %ListOfDataTypes = "")
    As %Status [ Final ]
```

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

%Import() is a onetime, startup operation. It only needs to be called the first time you wish to generate the proxy classes. It is necessary again only if you recompile your .NET code and wish to regenerate the proxies. The following sections provide further details about the **%Import()** method:

- Import Arguments
- Import Dependencies and Exclusions

8.4.4.1 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 Object 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.

8.4.4.2 Import Dependencies and Exclusions

While mapping a .NET class into a proxy class and importing it into InterSystems IRIS, the Object 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 Object 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 Object Gateway automatically excludes a small subset of assemblies such as Microsoft.* assemblies.

8.4.5 %ExpressImport() Method

```
ClassMethod %ExpressImport(name As %String,
                           port As %Integer,
                           host As %String = "127.0.0.1",
                           silent As %Boolean = 0,
                           additionalClassPaths As %ListOfDataTypes = "",
                           exclusions As %ListOfDataTypes = "")
    As %Status [ Final ]
```

%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.

8.4.6 %GetAllClasses()Method

```
 %GetAllClasses(jarFileOrDirectoryName As %String,
                ByRef allClasses As %ListOfDataTypes)
     As %Status
```

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*.

8.5 Using the Command Prompt

Usually you start and stop the Object Gateway server automatically, by configuring the EnsLib.DotNetGateway.Service business service as a member of the production. Once this is done, the Object Gateway server starts and stops automatically with the production. The **StartGateway()** class method is also available to manually start the Object Gateway server.

However, during development or debugging, or when InterSystems IRIS and the Object Gateway server run on different machines, you may find it useful to start the gateway server from a command prompt. Copy the file DotNetGatewaySS.exe to the directory where you load an assembly. By default, DotNetGatewaySS.exe is shipped in the directory *install-dir*\dev\dotnet\bin. Run DotNetGatewaySS.exe from *install-dir*\dev\dotnet\bin as follows:

`DotNetGatewaySS` *port listener logfile*

Argument	Description
port	Port number on which to listen for the incoming requests.
listener	*Optional* — Contains the local IP address on the local machine where the gateway listens. Specify null, `""`, or `0.0.0.0` (the default) to listen on *all* IP addresses local to the machine (127.0.0.1, VPN address, etc.) You can restrict the listener to one existing local IP address or listen on all of them; you cannot enter a list of acceptable addresses. You must provide a value for this argument if you are specifying a *logfile*.
logfile	*Optional* — If specified, the command procedure creates a log file of this name; you must specify the full pathname in the string. The *listener* argument is required if you enter a value for *logfile*.

For example:

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

Note: When using classes in local side-by-side assemblies (assemblies are not installed into the GAC), run DotNetGatewaySS.exe from the same directory as those assemblies to resolve their dependencies.

8.6 Using the .NET Object Gateway Wizard

You can import a DLL assembly file from .NET and create a set of corresponding classes using the .NET Object Gateway wizard built into Studio. To start the wizard:

1. Start Studio.
2. From the **Tools** menu, point to and click **Add-Ins**.
3. Click **.NET Gateway Wizard** to start the .NET Object Gateway Wizard dialog.
4. Enter the path and name of a DLL assembly file; or click **Browse** to help navigate to one.
5. Enter the **.NET Gateway server name / IP address** and **.NET Gateway server port** for the Object Gateway server.
6. You can also enter **Additional paths\assemblies to be used in finding dependent classes** and **Exclude dependent classes matching the following prefixes** as instructed in the dialog.
7. Click **Next** to generate proxy classes. The wizard displays the class name as it generates each proxy class.
8. When the import operation is complete, click **Finish** to exit the wizard.

8.7 Error Checking

The Object Gateway provides error checking as follows:

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

In both cases, InterSystems IRIS 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:

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

8.8 Troubleshooting

Should you encounter problems while using the Object Gateway it is always a good idea to turn logging on. That might be necessary for InterSystems staff to help you troubleshoot problems. To activate logging, simply identify a log file when you start the Object Gateway. You can do this whether you start from the command line or use the **StartGateway()** API method.

Sometimes, while using the Object 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 Object 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:

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

Where *port* is the port number to close.