Using InterSystems External Servers

Version 2023.3
2024-05-16

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

Using InterSystems External Servers
InterSystems IRIS Data Platform Version 2023.3 2024-05-16

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

1 Introduction to InterSystems External Servers 1

2 Working with External Languages 3
2.1 Creating a Gateway and Using a Proxy Object 4
2.2 Defining Paths to Target Software 5
2.2.1 Specifying Python Targets 7

3 Managing External Server Connections 9
3.1 Controlling Connections with the Management Portal 9
3.2 Controlling Connections with the $system.external Interface 10
3.2.1 Starting and Stopping External Servers 11
3.2.2 Displaying the Activity Log 12

4 Customizing External Server Definitions 15
4.1 Customizing Definitions in the Management Portal 15
4.1.1 The New External Server Form 16
4.2 Defining External Server Configurations for Java 17
4.3 Defining External Server Configurations for .NET 18
4.4 Defining External Server Configurations for Python 18
4.5 Defining Advanced Settings 19
4.5.1 Advanced Options for All External Servers 19
4.5.2 Advanced Setting for .NET 20
4.5.3 Advanced Settings for Python 20
4.6 Customizing Definitions with the $system.external Interface 21
4.6.1 Using getServer() to Retrieve Configuration Settings 21
4.6.2 Getting Other Information about Existing Definitions 22
4.6.3 Creating and Modifying External Server Definitions 24

5 InterSystems External Server Requirements 27
5.1 InterSystems IRIS Requirements 27
5.2 Java External Server Setup 28
5.3 .NET External Server Setup 28
5.4 Python External Server Setup 29
5.5 Troubleshooting External Server Definitions 29
5.6 Upgrading Object Gateway Code 30

6 Quick Reference for the ObjectScript $system.external Interface 33
6.1 All methods by Usage 33
6.2 $system.external Method Details 34
6.3 Gateway Methods 40
6.3.1 Gateway Method Details 40

List of Figures

Figure 2–1: Connection between a Gateway object and an InterSystems External Server 3
Figure 3–1: External Servers page (System Administration > Configuration > Connectivity) 9
Figure 4–1: External Servers page (System Administration > Configuration > Connectivity) 15
Figure 4–2: Creating an External Java Server definition 17
Figure 4–3: Creating an External .NET Server definition 18
Figure 4–4: Creating an External Python Server definition 19
Figure 4–5: Advanced Settings used by all External Servers 19
Figure 5–1: InterSystems External Servers (System Administration > Configuration > Connectivity) 30
Figure 5–2: Example of an error message on startup 30

1

Introduction to InterSystems External Servers

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

InterSystems External Servers provide instant, fully integrated bi-directional connections between InterSystems IRIS and external language platforms. External servers support these fast, simple, powerful features:

Instant access

The $system.external interface makes all of your Java, .NET, and Python objects effectively part of the ObjectScript language. External servers start automatically when you issue an command, providing instant access to your external language platforms without any preliminary setup.

Shared sessions

Shared sessions let ObjectScript and external applications work within the same context and transaction, sharing the same bi-directional connection.

Remote object control

Proxy objects allow either side of a shared session to control target objects on the other side in real time. ObjectScript can create and control Java, .NET, and Python objects, and those languages can create and control ObjectScript objects.

Fully reentrant bidirectional connections

The normal one-way client/server relationship is no longer a barrier. With bidirectional reentrancy, new objects or method calls on either side can enter an existing shared session at any time. This allows applications on either side of a shared session to act as both server and client, initiating client queries and serving requests from the other side.

Reentrant Native SDK methods

Native SDK methods are also fully reentrant and can be included in shared sessions, giving external applications direct access to a huge array of InterSystems IRIS resources.

This document describes how to work with InterSystems External Servers in ObjectScript. The section on "Working with External Languages" provides most of the information you will need to use external servers in your ObjectScript code. See the "Quick Reference for the $system.external Interface" for detailed information on all methods covered here. See "External Server Requirements" for software requirements and optional setup instructions.

External servers use an enhanced and simplified form of the older Dynamic Object Gateway technology. All Object Gateway features are still available, so upgrading your code is a simple matter of replacing certain class and method references (see "Upgrading Object Gateway Code" for details).

The rest of the document describes optional methods for controlling and configuring external servers. See "Managing External Server Connections" for information on how to view external server activity in the Management Portal, and how to control connections in ObjectScript. The section on "Customizing External Server Definitions" discusses advanced options that allow you to define new external server configurations for special purposes.

The InterSystems Native SDKs are a vital part of the InterSystems External Server environment. They provide the Java, .NET, and Python frameworks that allow your applications to take full advantage of external server connections. See the following documents for detailed information:

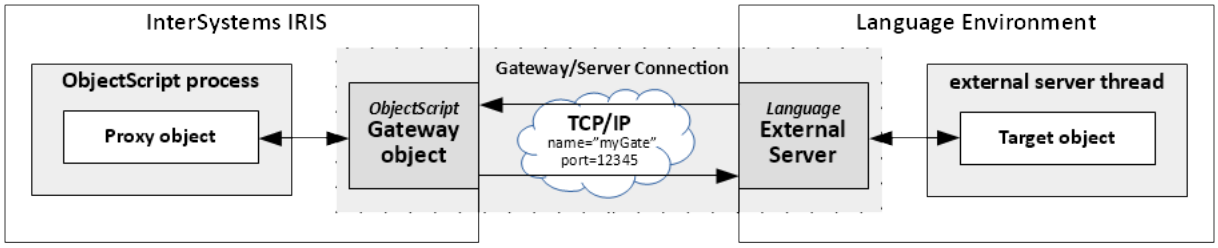
- Using the Native SDK for Java
- Using the Native SDK for .NET
- Using the Native SDK for Python

2

Working with External Languages

The $system.external interface allows you to generate ObjectScript *proxy objects* that control corresponding Java, .NET, or Python *target objects*. A proxy object has the same set of methods and properties as the target, and each call to the proxy is echoed by the target. Communications between the proxy and the target are managed by an ObjectScript Gateway object connected to a Java, .NET, or Python External Server.

Figure 2–1: Connection between a Gateway object and an InterSystems External Server



- The *Gateway* process runs in an InterSystems IRIS namespace, and manages the connection for the ObjectScript application.
- The InterSystems *External Server* process runs in the external language environment (Java, .NET, or Python) and provides the interface to external language shared libraries. Individual server threads provide access to class methods and properties, and manage communications between proxy and target objects.
- A *bidirectional TCP/IP connection* allows the gateway object and the external server to exchange messages using a port number that uniquely identifies the external server instance.

The following sections demonstrate how to establish connections, define targets, and use proxy objects:

- Creating a Gateway and Using a Proxy Object
- Defining Paths to Target Software

2.1 Creating a Gateway and Using a Proxy Object

The whole process of starting a connection, creating a proxy object, and calling a method can be compressed into a single statement. The following example starts the Java External Server, creates a proxy for an instance of target class java.util.Date, and calls a method to display the date:

```
write $system.external.getJavaGateway().new("java.util.Date").toString()

   Sun May 02 15:18:15 EDT 2021
```

This one line of code demonstrates the entire process of creating and using proxy objects. It establishes a Java gateway connection, creates an instance of class java.util.Date and a corresponding ObjectScript proxy, and writes the date returned by method toString().

Tip: Try running this call at the InterSystems Terminal. Here are equivalent commands for C# and Python:

```
write $system.external.getDotNetGateway.new("System.DateTime",0).Now
write $system.external.getPythonGateway.new("datetime.datetime",1,1,1).now().strftime(%c)
```

> The external servers are set up automatically when you install InterSystems IRIS, and will probably work without further attention (if not, see "Troubleshooting External Server Definitions" — you may have to change the path setting that identifies your preferred language platform).

The previous example compressed everything into one line, but that line is doing three important things:

- First, it creates an ObjectScript Gateway object that encapsulates the connection between ObjectScript and the external server. You can assign the gateway to a persistent variable:

  ```
  set javaGate = $system.external.getJavaGateway()
  ```

- Next it calls the gateway object's **new()** method, which creates an instance of java.util.Date in an external server thread and a corresponding proxy object in the ObjectScript process. The proxy object can also be assigned to a persistent variable:

  ```
  set dateJava = javaGate.new("java.util.Date")
  ```

- Finally, it calls a proxy object method. The target object echoes the call and returns the result to the proxy:

  ```
  write dateJava.toString()
  ```

The following examples demonstrate these steps for all three languages.

Creating gateway objects

There are specific gateway creation methods for the Java, .NET, and Python External Servers: **getJavaGateway()**, **getDotNetGateway()**, and **getPythonGateway()**. Each of these methods starts its external server in the default configuration and returns a Gateway object to manage the connection:

```
set javaGate = $system.external.getJavaGateway()
set netGate = $system.external.getDotNetGateway()
set pyGate = $system.external.getPythonGateway()
```

There is also a generic **getGateway()** method for use with customized external server configurations (see "Customizing External Server Definitions" for details) but the defaults should be sufficient for most purposes.

Creating proxy objects

Each gateway object has a **new()** method for creating target and proxy objects. Each call to **new()** specifies a class name and any required arguments. When the call is made, the external server creates a target instance of the class, and the gateway creates a corresponding proxy object that has the same set of methods and properties as the target. Each call to the proxy is echoed by the target, which returns the result to the proxy.

This example creates three proxy objects connected through three different external servers (since each language requires its own server):

```
set dateJava = javaGate.new("java.util.Date")
set dateNet = newGate.new("System.DateTime",0)
set datePy = pyGate.new("datetime.datetime",1,1,1).now()
```

Each proxy can be treated like any other ObjectScript object. All three can be used in the same ObjectScript application.

Calling proxy object methods and properties

You can use method and property calls from all three proxy objects in the same statement:

```
  write !,"  Java: "_dateJava.toString(),!,"  .NET: "_dateNet.Now,!, "Python:
"_datePy.strftime("%c")

       Java: Sun May 02 15:18:15 EDT 2021
       .NET: 2021-05-02 16:38:36.9512565
     Python: Sun May 02 15:23:55 2021
```

The Java and Python examples are method calls, and the .NET example uses a property.

So far the examples have only used system classes, which are easy to demonstrate because they're available at all times. In other cases, you will have to specify the location of a class before it can be used. For details, see "Defining Paths to Target Software" later in this section.

2.2 Defining Paths to Target Software

All Gateway objects can store a list of paths to software libraries for a specific language. Java gateways accept .jar files, .NET gateways accept .dll assemblies, and Python gateways accept .py (module or class) files.

The **addToPath()** method allows you to add new paths to the list. The *path* argument can be a simple string containing a single path, or a dynamic array containing multiple paths. For example:

Adding a single path

```
do javaGate.addToPath("/home/myhome/someclasses.jar")
do netGate.addToPath("C:\Dev\myApp\somedll.dll")
do pyGate.addToPath("/rootpath/person.py")
```

The Java and .NET gateways also accept paths to folders containing one or more shared libraries. See "Specifying Python Targets" for more advanced Python options

Adding paths to a dynamic array

The %DynamicArray class is an ObjectScript wrapper that provides a simple way to create a JSON array structure. Use the dynamic array **%Push()** method to add path strings to the array. The following example adds two paths to array *pathlist* and then passes it to the **addToPath()** method of a Java Gateway object:

```
set pathlist = []
do pathlist.%Push("/home/myhome/firstpath.jar")
do pathlist.%Push("/home/myhome/another.jar")
do javaGate.addToPath(pathlist)
```

For more information on dynamic arrays, see "Using %Push and %Pop with Dynamic Arrays" in *Using JSON*.

Note: The *path* argument can also be specified as an instance of %Library.ListOfDataTypes containing multiple path strings, but dynamic arrays are recommended for ease of use.

The following examples demonstrate how to specify classes for each language:

Defining paths to Java classes

For Java, the path can be a folder or a jar. The following example adds a path to someclasses.jar and then creates a proxy for an instance of someclasses.classname.

```
set javaGate = getJavaGateway()
do javaGate.addToPath("/home/myhome/someclasses.jar")
set someProxy = javaGate.new("someclasses.classname")
```

Defining paths to .NET classes

For .NET, the path can be a folder or an assembly. The following example adds a path to someassembly.dll and then creates a proxy for an instance of someassembly.classname

```
set netGate = getDotNetGateway()
do netGate.addToPath("C:\Dev\myApp\somedll.dll")
set someProxy = netGate.new("someassembly.classname")
```

Defining paths to Python classes

For Python, the path can be a module or a package. When a module is part of a package, the module path must be specified with dot notation starting at the top level package directory. For example, the path to module Foo.py could be specified in either of two ways:

- Standard path notation if treated as a module: `C:\Dev\demo\Foo.py`
- Dot notation if treated as part of package demo: `C:\Dev\demo.Foo.py`

The following example uses a dynamic array to add paths for two different files, both in folder C:\Dev\demo\. File Foo.py contains unpackaged class personOne, and file Bar.py contains class personTwo, which is part of package demo. Calls to **new()** create proxies for both classes:

```
set pyPaths = []
do pyPaths.%Push("C:\Dev\demo\Foo.py")
do pyPaths.%Push("C:\Dev\demo.Bar.py")

set pyGate = getPythonGateway()
do pyGate.addToPath(pyPaths)
set fooProxy = pyGate.new("Foo.personOne")
set barProxy = pyGate.new("demo.Bar.personTwo")
```

For more information on dynamic arrays, see "Using %Push and %Pop with Dynamic Arrays" in *Using JSON*.

It is also possible to assign targets for modules or packages that do not have classes, as described in the following section.

2.2.1 Specifying Python Targets

The syntax for specifying a Python target differs depending on whether the target is in a package, in a class, both, or neither. The following examples show the path as it should be specified in **addToPath()** and the class name as specified in **new()**.

In the examples with classes, person is the main class and company is a class that it imports. The imported class does not have to be specified even if it is in a separate file.

You can also assign targets for modules or packages that do not have classes, but they are effectively limited to static methods and properties (since you can`t have class instance methods without a class).

no package, no class

Modules with no package and no class use the file name. Note that the argument for **new()** is the filename followed by a period, indicating that there is no class:

```
do pyGate.addToPath("/rootpath/noclass.py")
set proxy = pyGate.new("noclass."))
```

no package, two classes in one file

Main class person and imported class company are both in file /rootpath/onefile.py.

```
do pyGate.addToPath(/rootpath/onefile.py")
set proxy = pyGate.new("onefile.person")
```

no package, two classes in separate files

Main class person and imported class company are in separate files within /rootpath:

```
do pyGate.addToPath(/rootpath/person.py")
set proxy = pyGate.new("person.person")
```

package, no class

Packages with no classes use the package name and file name. The actual path for the file in this example is /rootpath/demo/noclass.py. Note that the argument for **new()** ends with a period, indicating that there is no class:

```
do pyGate.addToPath(/rootpath/demo.noclass.py")
set proxy = pyGate.new("demo.noclass.")
```

package, two classes in one file

Main class person and imported class company are both in file /rootpath/demo/onefile.py:

```
do pyGate.addToPath(/rootpath/demo.onefile.py")
set proxy = pyGate.new("demo.onefile.person")
```

package, two classes in separate files

Main class person and imported class company are in separate files in /rootpath/demo:

```
do pyGate.addToPath(/rootpath/demo.person.py")
set proxy = pyGate.new("demo.person.person")
```

3

Managing External Server Connections

This section describes how to start and stop external servers, and how to get information about current server activity. You can do so in either of two ways: interactively through the Management Portal, or programmatically through the ObjectScript $system.external interface.

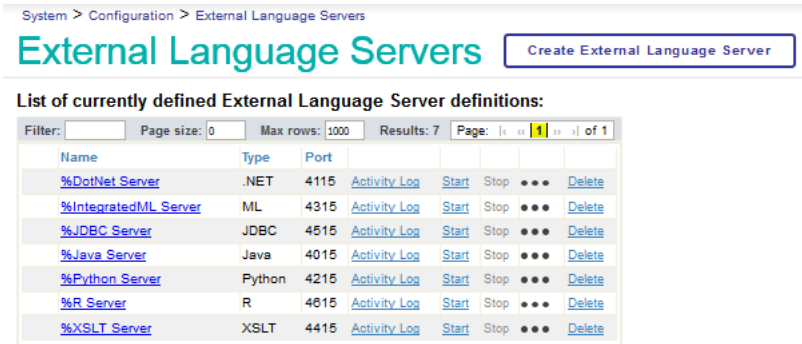
3.1 Controlling Connections with the Management Portal

InterSystems External Server definitions are named collections of configuration settings. These settings are used to construct the commands that start External Server processes. For example, definitions include information such as TCP/IP address, port number, and paths to classes or executables.

The Management Portal provides a page that displays the current status of all defined external servers. It allows you to start or stop external servers, and to control and monitor gateway connections. All of the definitions listed below contain default settings configured during installation. See "Customizing External Server Definitions" for information on modifying these definitions or creating new ones.

Note: By default, all servers require a gateway resource, which allows them be used with a passphrase secured connection. Unless security is disabled, you must have USE privileges on the appropriate resource to start, stop, and connect to a Gateway. Java, .NET, and Python servers require the **`%Gateway_Object`** resource. See the InterSystems Authorization Guide for more information.

Figure 3–1: External Servers page (System Administration > Configuration > Connectivity)



To open the InterSystems External Servers page, go to System Administration > Configuration > Connectivity > External Servers. In this illustration, all of the names start with %, indicating that they are the default external server configurations (defined automatically during InterSystems IRIS installation). The `%Java Server`, `%Python Server`, an `%DotNet`

`Server` definitions are the default external server configurations for the languages covered in this document. The other listed definitions are specialized configurations used internally by other InterSystems interfaces (as described elsewhere in relevant documentation).

Note: Do not confuse `%Java Server` with `%JDBC Server`, which is a specialized configuration used internally by other InterSystems interfaces.

Each line in the display identifies one unique configuration, and allows you to monitor and control the external server that uses it. For example, this line lists the default .NET external server configuration:

This line contains the following fields:

- *Name* is the unique identifier for this collection of configuration settings. The configuration it represents can only be used by one external server instance at a time. Clicking on the *Name* field brings up an editor that allows you to change most values of the configuration (see "Customizing External Server Definitions" for details).
- *Type* indicates which language the external server will run under.
- *Port* displays the port number used by this configuration. You can specify the same port number in two different definitions, but they can't be used at the same time. No two gateway connections can use the same port simultaneously.
- *Activity Log* is a link to a page that displays detailed information about all instances of this server definition that have been started or stopped since InterSystems IRIS was started. See "Displaying the Activity Log" in the next section for information about how to display the same information using the $system.external interface.
- *Start/Stop* allows you to control whether or not the external server process is currently running. *Start* will create an instance of the external server and display detailed startup information about the process. If the external server is already running, a *Stop* command will be displayed instead. See "Starting and Stopping External Servers" in the next section for information about how to do this with the $system.external interface.
- *Delete* allows you to delete the external server definition. See "Creating and Modifying External Server Definitions" for information about how to create and delete definitions with the $system.external interface.

The default configurations will be sufficient for most purposes, but it may be useful to define additional configurations in some cases. See "Customizing External Server Definitions" for information on how to modify an existing definition or create a new one.

3.2 Controlling Connections with the $system.external Interface

This section demonstrates how to use the $system.external interface to start and stop a Gateway process, manage the connection to a Gateway Server, and get information on the status of a Gateway:

- Starting and Stopping External Servers
- Displaying the Activity Log

3.2.1 Starting and Stopping External Servers

The $system.external **startServer()** and **stopServer()** methods allow you to start or stop an instance of the specified external server. **isServerRunning()** returns a boolean indicating whether the specified external server is currently running. The method signatures are:

```
startServer(serverName As %String) as %Boolean
```

```
stopServer(serverName As %String, verbose As %Boolean = 0) as %Boolean
```

```
isServerRunning(serverName As %String) as %Boolean
```

InterSystems External Servers are designed to be automatic. An external server will be started (if not already running) when a request for a connection is received, and stopped when the InterSystems IRIS instance that started the server is shut down.

Note: By default, all servers require a gateway resource, which allows them be used with a passphrase secured connection. Unless security is disabled, you must have USE privileges on the appropriate resource to start, stop, and connect to a Gateway. Java, .NET, and Python servers require the **%Gateway_Object** resource. See the InterSystems Authorization Guide for more information.

Generally the server is started when an instance of a Gateway object is created. However, you can use the **startServer()** and **stopServer()** methods to control an external server even when no Gateway object exists. (This is the same way the Management Portal `Start`/`Stop` links work, as described in the previous section, "Controlling Connections with the Management Portal").

The following example tests to see if an instance of `%Java Server` is running. If not, the external server is started and the return value is checked to make sure startup was successful. Finally the external server is stopped and the related activity log messages are displayed:

Starting and Stopping the Server

This example starts and stops an external server instance that uses `%Java Server`, the default Java configuration. The **isServerRunning()** method is used before starting to make sure the server is in the expected state.

```
set serverName = "%Java Server"
set isRunning = $system.external.isServerRunning(serverName)
write isRunning

   0
```

The **startServer()** method returns true (1) if the external server was successfully started:

```
if '(isRunning) set isRunning = $system.external.startServer(serverName)
write isRunning

   1
```

The **stopServer()** method also returns a boolean to indicate success or failure. If the *verbose* argument is set to true, it also displays the activity log messages on the console:

```
set isRunning = $system.external.stopServer(serverName,1)

   2020-06-24 13:09:39  Stopping Java Gateway Server '%Java Server'
   2020-06-24 13:09:39  Stopping process '89381' that is monitoring the Gateway
   Server '%Java Server' on port '53180'
   2020-06-24 13:09:39  Shutting down the Gateway Server
   2020-06-24 13:09:39  Invoking %Connect with Server='127.0.0.1', Port='53180',
   Namespace='USER', Timeout='5'
   2020-06-24 13:09:39  Shutting down Java Gateway Server '%Java Server'
   2020-06-24 13:09:39  Return from %Shutdown: OK
   2020-06-24 13:09:39  Gateway Server stopped

write isRunning

   0
```

Activity Log entries can be retrieved at a later time, as described in the following section.

3.2.2 Displaying the Activity Log

The **getActivity()** method returns a set of Activity Log entries for the specified InterSystems External Server. The method signature is:

```
getActivity(serverName As %String, entryCount As %Integer = 10, verbose As
%Boolean = 0) as %Library.DynamicArray
```

See "Troubleshooting External Server Definitions" for a description of the Activity Log interface in the Management Portal.

Each server logs messages about the state of the server. **getActivity()** returns the specified *entryCount* number of messages from this log as a dynamic array. If *verbose* is true, it will also display the messages on the current device.

you can use dynamic object method **%ToJSON()** to display the entire list of settings at once.

Note: The %DynamicArray class is one of several related ObjectScript features that allow you to work directly with JSON strings and data structures. See Using JSON for complete details.

getActivity() as JSON

The following example gets the first three activity log entries and displays them with **%ToJSON()** (linebreaks added for clarity):

```
set activity = $system.external.getActivity("%Java Server",3)
write activity.%ToJSON()

   [{"DateTime":"2021-05-08 17:19:28","Job":"15037","Port":4013,"Server":"%Java Server",
   "Text":"Starting Java Gateway Server '%Java Server'","Type":"Info"},
   {"DateTime":"2021-05-08 17:19:28","Job":"15037","Port":4013,"Server":"%Java Server",
   "Text":"Return from RunStartCmd: ERROR #5001: Java executable not found in the given
   directory: /nethome/bad/java/path/bin/","Type":"Error"},
   {"DateTime":"2021-05-08 17:19:28","Job":"15037","Port":4013,"Server":"%Java Server",
   "Text":"An error occurred while trying to start the Gateway Server","Type":"Info"}]
```

Optionally, the log items can be displayed on the current device by setting *verbose* to true (1):

getActivity() as formatted display

Here is the same set of log entries as a screen display

```
set activity = $system.external.getActivity("%Java Server",3,1)

+--------------+-------+---------------------+--------+-------+-----------------------------+
| Server       | Port  | DateTime            | Type   | Job   | Text                        |
+--------------+-------+---------------------+--------+-------+-----------------------------+
| %Java Server | 4013  | 2021-05-08 17:19:28 | Info   | 15037 | Starting Java Gateway Serve |
|              |       |                     |        |       | r '%Java Server'            |
| %Java Server | 4013  | 2021-05-08 17:19:28 | Error  | 15037 | Return from RunStartCmd: ER |
|              |       |                     |        |       | ROR #5001: Java executable  |
|              |       |                     |        |       | not found in the given dire |
|              |       |                     |        |       | ctory: /nethome/bad/java/pa |
|              |       |                     |        |       | th/bin/                     |
| %Java Server | 4013  | 2021-05-08 17:19:28 | Info   | 15037 | An error occurred while try |
|              |       |                     |        |       | ing to start the Gateway Se |
|              |       |                     |        |       | rver                        |
+--------------+-------+---------------------+--------+-------+-----------------------------+
```

4

Customizing External Server Definitions

External server definitions are named collections of configuration settings. These settings are used to construct the commands that start external server processes. For example, definitions include information such as TCP/IP address, port number, and paths to classes or executables.

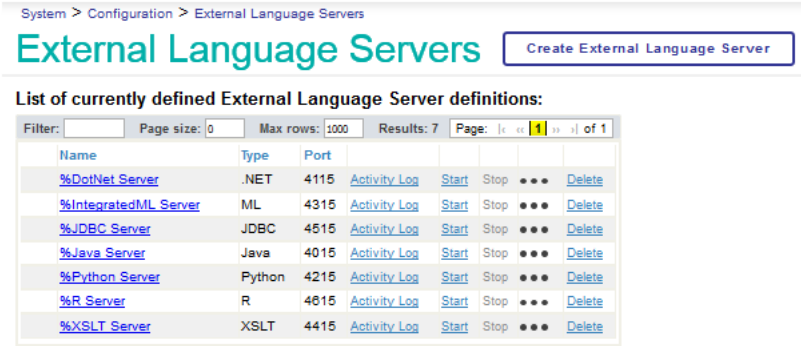
For most purposes, the predefined external server settings will provide all the functionality you need (see "Managing External Server Connections"), but you can easily change the default settings if necessary. Alternatively, you can create entirely new external server definitions containing your custom settings.

This section describes how to manage named collections of external server configuration settings, and how to create new ones. You can do so in two ways: interactively through the Management Portal, or programmatically with the ObjectScript $system.external interface.

4.1 Customizing Definitions in the Management Portal

An external server definition is a named collection of configuration settings stored in the InterSystems IRIS database. This information is used to construct the command line that starts an external server process. Each external server definition has a unique name, defining a connection that can only be used by one external server at a time. You can access a definition by clicking on the name in the External Servers page of the Management Portal:

Figure 4–1: External Servers page (System Administration > Configuration > Connectivity)



Note: To create and change server definitions as described in this section, you must have the **`%Admin_ExternalLanguageServerEdit:USE`** privilege, which by default is held by the **`%Manager`** role.

The following sections describe the Management Portal pages used to access existing definitions and create new ones:

- The New External Server Form
- Defining External Server Configurations for Java
- Defining External Server Configurations for .NET
- Defining External Server Configurations for Python
- Defining Advanced Settings

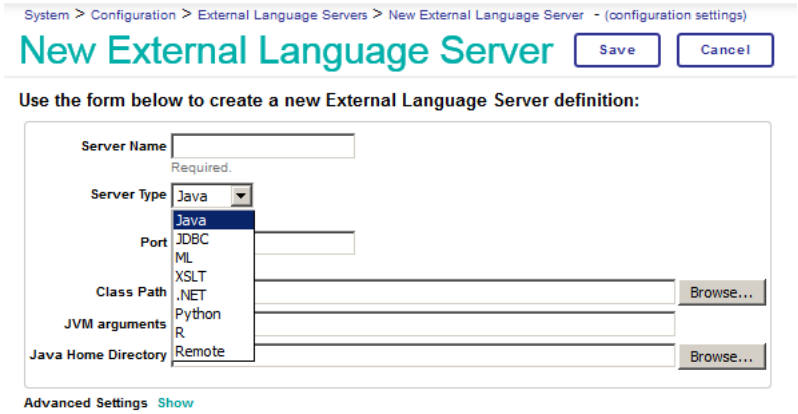
4.1.1 The New External Server Form

You must have the **%Admin_ExternalLanguageServerEdit:USE** privilege to create and change server definitions. In some cases, creating a new server definition may be a convenient alternative to changing the settings of the default server. You can specify language environments, versions, and system configurations that differ from the default values for your system. Most configuration settings are optional, with defaults that will be used if no other value is specified.

Each external server definition must include a Server Name, a Type (currently Java, .Net, and Python), and a Port. Additional properties depend on the Type, and allow you to define the path to the language platform, paths to user code, and other language-specific options. There are also configurable items for logging, connection parameters, and monitoring.

The *External Servers* page (System Administration > Configuration > Connectivity > External Servers) contains a `Create External Server` button:

Clicking the button brings up the form shown below, which is used to define configurations for all external servers covered in this document (External Java Server, External .NET Server, and External Python Server) as well as other specialized types not covered here.



This illustration shows the Server Type list with `Java` as the selected type. In addition to the required Server Name, Server Type, and Port options, the form displays three Java-specific fields. Similar type-specific fields would be shown for any selected type.

The following fields are identical for all languages:

- `Server Name` *Required*.

  Unique name that you assign to this server definition. The name is used as the database identifier for the stored definition, and cannot be changed after the definition has been saved.

- `Server Type` *Required.*

  Select `Java`, `.NET`, or `Python` to display the fields used only by the selected language.

- `Port` *Required.*

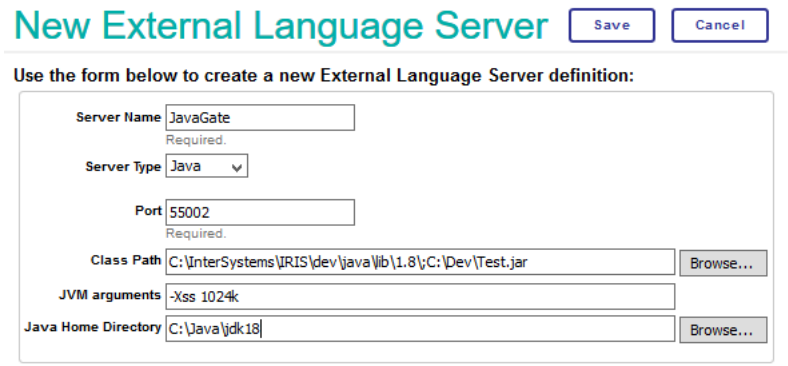
  Unique TCP port number for communication between the external server and InterSystems IRIS. No two external server connections can use the same port simultaneously.

The following sections show and describe the type-specific fields for Java, .NET, and Python.

4.2 Defining External Server Configurations for Java

See "The New External Server Form" for more information on fields common to all languages. The `Class Path`, `JVM arguments`, and `Java Home Directory` fields are specific to Java.

Figure 4–2: Creating an External Java Server definition



- `Server Name` (*required*) — Unique name for this external server configuration.
- `Server Type` (*required*) — Select `Java` to display the fields used only by Java external servers (*do not* select `JDBC`, which is a specialized configuration used only by older InterSystems interfaces).
- `Port` (*required*) — Unique TCP port number used by this external server configuration.
- `Class Path`

  This field will be added to the class path. It should include paths to all jar files that contain the target classes you will need to generate proxies. There is no need to include InterSystems External Server .jar files. If you are specifying file paths containing spaces or multiple files, you should quote the classpath and supply the appropriate separators for your platform.

- `JVM arguments`

  Optional arguments to be passed to the JVM in the command that starts the external server. For example, you can specify settings such as system properties (`Dsystemvar=value`) or maximum heap size (`Xmx256m`) as needed.

- `Java Home Directory`

  Path to the Java version required by the external server. If not specified, this setting will default to the Java Home Directory setting in the Management Portal.

- `Advanced Settings` — See "Advanced Settings" for details.

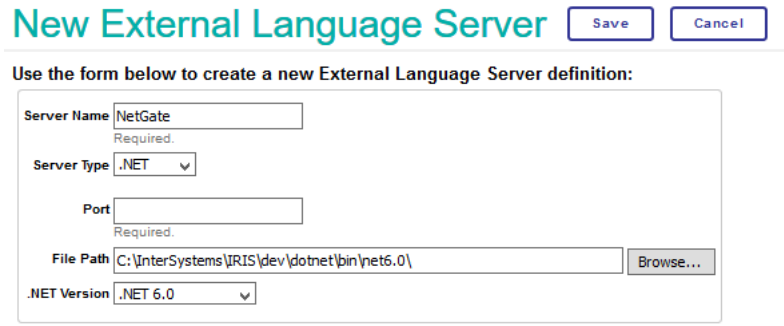
Click *Save* to store the server definition in the database. The `Server Name` is the unique identifier for the definition, and cannot be changed after the definition has been saved.

An existing server definition can be edited by clicking the *edit* link for that definition on the main External Servers page. The Edit page is identical to the page shown here except that the `Server Name` field cannot be changed.

4.3 Defining External Server Configurations for .NET

See "The New External Server Form" for more information on fields common to all languages. The `File Path` and `.NET Version` fields are specific to .NET.

Figure 4–3: Creating an External .NET Server definition



- `Server Name` (*required*) — Unique name for this external server configuration.
- `Server Type` (*required*) — Select `.NET` to display the fields used only by .NET external servers.
- `Port` (*required*) — Unique TCP port number used by this external server configuration.
- `File Path`

  Specifies the full path of the directory containing the required server executable for .NET (see ".NET External Server Setup"), used by the command that starts the external server. If this field is not specified, the startup command will use the default executable for your selected `.NET Version`.

  When using .NET, some systems may require you to specify <*install-dir*>\dev\dotnet (the top level directory for all .NET external server assemblies).
- `.NET Version`

  Specifies the .NET version required by the external server executable. The .NET options are available for all operating systems. In Windows, versions of .NET Framework are also supported.
- `Advanced Settings` — See "Defining Advanced Settings" and "Advanced Setting for .NET" for details.

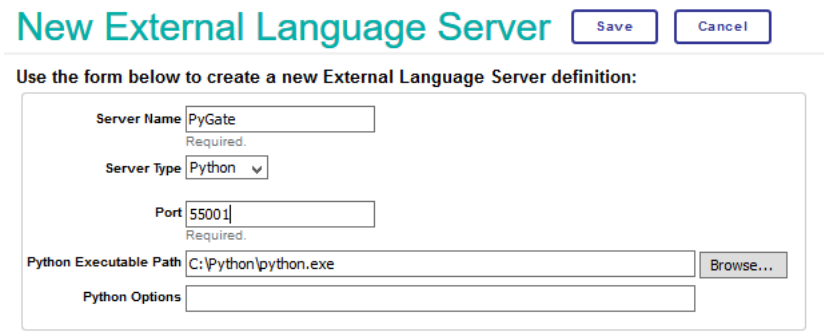
Click *Save* to store the server definition in the database. The `Server Name` is the unique identifier for the definition, and cannot be changed after the definition has been saved.

An existing server definition can be edited by clicking the *edit* link for that definition on the main External Servers page. The Edit page is identical to the page shown here except that the `Server Name` field cannot be changed.

4.4 Defining External Server Configurations for Python

See "The New External Server Form" for more information on fields common to all languages. The `Python Executable Path` and `Python Options` fields are specific to Python.

Figure 4–4: Creating an External Python Server definition



- `Server Name` (*required*) — Unique name for this external server configuration.
- `Server Type` (*required*) — Select `Python` to display the fields used only by Python external servers.
- `Port` (*required*) — Unique TCP port number used by this external server configuration.
- `Python Executable Path`

  Fully qualified filename of the Python executable that will be used to run the external server. This setting is optional if there is a default Python interpreter that can be used without specifying its location.
- `Python Options`

  Optional property that defines extra command line settings to be included when constructing the Python command that starts the external server.
- `Advanced Settings` — See "Defining Advanced Settings" and "Advanced Setting for Python" for details.

Click *Save* to store the server definition in the database. The `Server Name` is the unique identifier for the definition, and cannot be changed after the definition has been saved.

An existing server definition can be edited by clicking the *edit* link for that definition on the main External Servers page. The Edit page is identical to the page shown here except that the `Server Name` field cannot be changed.

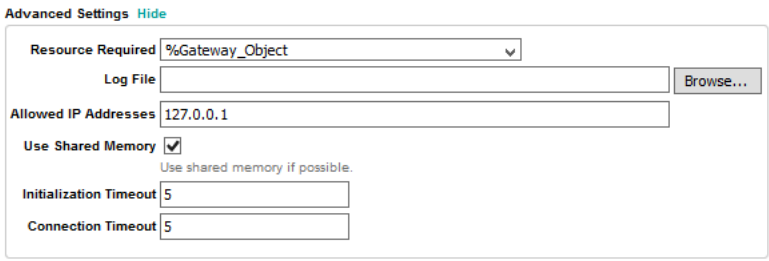
4.5 Defining Advanced Settings

The default values of the `Advanced Settings` do not have to be changed in most cases.

4.5.1 Advanced Options for All External Servers

The following options are available for Java, .NET, and Python.

Figure 4–5: Advanced Settings used by all External Servers



- `Resource Required`

  By default, all servers require a gateway resource, which allows them be used with a passphrase secured connection. Java, .NET, and Python servers require the **`%Gateway_Object`** resource with **`USE`** permission. A system administrator can create customized resources and make them required resources for using any specific Gateway. This field can also be set to an empty string, making the gateway public, but InterSystems strongly recommends protecting all gateways with the appropriate resource. See the InterSystems Authorization Guide for more information.

- `Log File`

  Full pathname of the log file on the host machine. External server messages are logged for events such as opening and closing connections to the server, and for error conditions encountered when mapping target classes to proxy classes. Due to performance considerations, this optional property should only be used for debugging.

- `Allowed IP Addresses`

  IP address or name of the host machine where the external server will run. The default is `"127.0.0.1"` (the local machine). Specify `"0.0.0.0"` to listen on all IP addresses local to the machine (127.0.0.1, VPN address, etc.).

- `Use Shared Memory`

  Indicates whether to use shared memory if available (default is true).

- `Initialization Timeout`

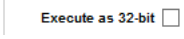
  Number of seconds to wait before assuming that external server initialization has failed.

- `Connection Timeout`

  Number of seconds to wait before assuming that an attempt to connect to the external server has failed.

4.5.2 Advanced Setting for .NET

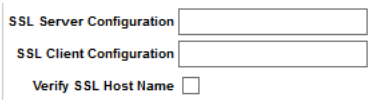
In addition to the settings described in "Options for All External Servers" the .NET advanced settings include an option to execute the external server as 32–bit:



- `Execute as 32-bit` — applies only to external servers on 64-bit platforms. If this property is checked, the external server will be executed as 32-bit. Defaults to 64-bit execution.

4.5.3 Advanced Settings for Python

In addition to the settings described in "Options for All External Servers" the Python advanced settings include options for TSL/SSL configuration (see "Create or Edit a TLS Configuration"):



- `SSL Server Configuration` — name of the SSL/TLS configuration to be used for server TLS/SSL.
- `SSL Cient Configuration` — name of the SSL/TLS configuration to be used for client TLS/SSL.
- `Verify SSL Host Name` — If this property is checked, the TLS/SSL client will perform hostname verification.

4.6 Customizing Definitions with the $system.external Interface

The Management Portal provides convenient interactive tools for defining and modifying external server configurations (as described in the previous section), but the same thing can be accomplished in ObjectScript using the $system.external interface.

Note: To create and change server definitions as described in this section, you must have the **%Admin_ExternalLanguageServerEdit:USE** privilege, which by default is held by the **%Manager** role.

You can create and use an external server definition with a few lines of ObjectScript code, specifying exactly the same information you would enter in the Management Portal. You can also access detailed information about all existing external server definitions:

- Using getServer() to Retrieve Configuration Settings describes how to get complete information on individual definitions.
- Getting Other Information about Existing Definitions describes how to get basic information about all existing definitions.
- Creating and Modifying External Server Definitions describes how to define and modify external server configurations for all languages.

Note: **%DynamicObject and %DynamicArray**

Many of the methods described in this section accept or return instances of %DynamicObject and %DynamicArray. These classes allow you to work with JSON strings and data structures in ObjectScript. See Using JSON for complete details and examples.

4.6.1 Using getServer() to Retrieve Configuration Settings

The **getServer()** method of the $system.external interface returns a %DynamicObject containing all configuration settings for the named external server definition. The method signature is:

```
getServer(serverName As %RawString) as %Library.DynamicObject
```

For example, the following code returns the settings for `%Java Server`, the default Java configuration:

```
set srv = $system.external.getServer("%Java Server")
write srv

   2@%Library.DynamicObject
```

Since each field is a property of the dynamic object, you can use standard property notation to display the fields you want to see:

```
write "Name:"_srv.Name_", Type:"_srv.Type_", Port:"_srv.Port,!

   Name:%Java Server, Type:Java, Port:4015
```

You can iterate over dynamic object properties with %DynamicObject methods **%GetIterator()** and **%GetNext()** (see "Iterating over a Dynamic Entity with %GetNext()" in *Using JSON* for details).

Alternatively, you can use dynamic object method **%ToJSON()** to display the entire list of settings at once. The following examples use **getServer()** to retrieve all settings for the three default external server configurations (`%Java Server`, `%Python Server`, or `%DotNet Server`) and use **%ToJSON()** to display the information.

Displaying an existing Java External Server definition

This example returns and displays a %DynamicObject containing all fields for the default Java server configuration (line breaks added for clarity):

```
write $system.external.getServer("%Java Server").%ToJSON()

   {"Name":"%Java Server",
   "FullName":"user3:IRIS_DS2:%Java Server","Type":"Java",
   "Port":4015,"LogFile":"","AllowedIPAddresses":"127.0.0.1",
   "ConnectionTimeout":5,"HeartbeatFailureAction":"R","HeartbeatFailureRetry":300,
   "HeartbeatFailureTimeout":30,"HeartbeatInterval":10,"InitializationTimeout":5,
   "UsePassphrase":0,"UseSharedMemory":1,"passphraseList":"",
   "SSLConfigurationServer":"","SSLConfigurationClient":"",
   "JVMArgs":"","JavaHome":"","ClassPath":""}
```

The last line in this example displays fields used only by the Java External Server (see "Defining External Server Configurations for Java" for detailed information on each field).

Displaying an existing .NET External Server definition

This example returns and displays a %DynamicObject containing all fields for the default .NET server configuration (line breaks added for clarity):

```
write $system.external.getServer("%DotNet Server").%ToJSON()

   {"Name":"%DotNet Server",
   "FullName":"user3:IRIS_DS2:%DotNet Server","Type":".NET",
   "Port":4115,"LogFile":"","AllowedIPAddresses":"127.0.0.1",
   "ConnectionTimeout":5,"HeartbeatFailureAction":"R","HeartbeatFailureRetry":300,
   "HeartbeatFailureTimeout":30,"HeartbeatInterval":10,"InitializationTimeout":5,
   "UsePassphrase":0,"UseSharedMemory":1,"passphraseList":"",
   "SSLConfigurationServer":"","SSLConfigurationClient":"",
   "DotNetVersion":"C2.1","Exec32":0,"FilePath":""}
```

The last line in this example displays fields used only by the .NET External Server (see "Defining External Server Configurations for .NET" for detailed information on each field).

Displaying an existing Python External Server definition

This example returns and displays a %DynamicObject containing all fields for the default Python server configuration (line breaks added for clarity):

```
write $system.external.getServer("%Python Server").%ToJSON()

   {"Name":"%Python Server",
   "FullName":"user3:IRIS_DS2:%Python Server","Type":"Python",
   "Port":4215,"LogFile":"","AllowedIPAddresses":"127.0.0.1",
   "ConnectionTimeout":5,"HeartbeatFailureAction":"R","HeartbeatFailureRetry":300,
   "HeartbeatFailureTimeout":30,"HeartbeatInterval":10,"InitializationTimeout":5,
   "UsePassphrase":0,"UseSharedMemory":1,"passphraseList":"",
   "SSLConfigurationServer":"","SSLConfigurationClient":"",
   "PythonOptions":"","PythonPath":""}
```

The last line in this example displays fields used only by the Python External Server (see "Defining External Server Configurations for Python" for detailed information on each field).

4.6.2 Getting Other Information about Existing Definitions

The **serverExists()**, **getServers()**, and **getServerLanguageVersion()** methods of the $system.external interface return information about external server definitions stored in the database. Since these are local database queries, none of the methods require an external server connection.

serverExists()

The **serverExists()** method returns true (1) if the external server definition exists, false (0) otherwise. The method signature is:

```
serverExists(serverName As %RawString) as %Boolean
```

For example:

```
write $system.external.serverExists("foo")

   0

write $system.external.serverExists("%Java Server")

   1
```

getServers()

The **getServers()** method returns a %DynamicArray containing the names of all external server definitions. The method signature is:

```
getServers() as %Library.DynamicArray
```

For example:

```
set srv = $system.external.getServers()
write srv.%ToJSON()

   ["%DotNet Server","%IntegratedML Server","%JDBC Server","%Java Server",
   "%Python Server","%R Server","%XSLT Server","JavaGate","NetGate2","PyGate2"]
```

In this example, the last two names are definitions created by the user (by convention, only the predefined default configurations should have names starting with "%").

getServerLanguageVersion()

The **getServerLanguageVersion()** method returns the external language version string for the specified external server definition. The method signature is:

```
getServerLanguageVersion(serverName As %RawString) as %String
```

For example

```
write $system.external.getServerLanguageVersion("%Java Server")

   8
```

We can get a more useful display by calling the **getServer()** method to get a dynamic object containing related information (see “Using getServer() to Retrieve Configuration Settings”):

```
set javaDef = $system.external.getServer("%Java Server")
set LanguageVersion = $system.external.getServerLanguageVersion(javaDef.Name)

write javaDef.Name_" language version is: "_javaDef.Type_" "_LanguageVersion

   %Java Server version is: Java 8
```

4.6.3 Creating and Modifying External Server Definitions

The $system.external interface allows you to manipulate external server definitions with the createServer(), modifyServer(), and deleteServer() methods. These methods work directly with definitions stored in the database, so all three require the %Admin_Manage resource for database access. A definition cannot be altered while the corresponding external server is running.

createServer()

The **createServer()** method accepts a list of settings in a %DynamicObject and returns a new server definition (including default values) as %DynamicObject. The method signature is:

```
createServer(serverDef As %DynamicObject) as %DynamicObject
```

The set of fields that can be defined in the input list is the same as the set returned by **getServer()** (see "Using getServer() to Retrieve Configuration Settings" for a complete list of default settings).

The input definition must specify values for all required fields (`Name`, `Type`, and `Port`) plus values for any other fields you want to customize. All input values are validated when **createServer()** is called. You can specify the input settings as dynamic object properties:

```
set def = {}
set def.Name = "AnotherServer"
set def.Type = "Java"
set def.Port = 5309
set def.ClassPath = "C:/dev/pathname/"
```

or as a JSON string:

```
set def = {"Name":"MyNewServer","Type":"Java","Port":5309, "ClassPath": "C:/dev/pathname/"}
```

Notice that the name in this example does not start with a % character. By convention, only the predefined default configurations should have names starting with "%".

The `Name` value must be unique, so this example calls **serverExists()** to test it before calling **createServer()**:

```
set badName = $system.external.serverExists(def.Name)
if ('badName) set srv = $system.external.createServer(def)
write srv.%ToJSON()

   {"Name":"MyNewServer","FullName":"user3:IRIS_DS2:MyNewServer","Type":"Java",
   "Port":5309,"LogFile":"","AllowedIPAddresses":"127.0.0.1",
   "ConnectionTimeout":5,"HeartbeatFailureAction":"R","HeartbeatFailureRetry":300,
   "HeartbeatFailureTimeout":30,"HeartbeatInterval":10,"InitializationTimeout":5,
   "UsePassphrase":0,"UseSharedMemory":1,"passphraseList":"",
   "SSLConfigurationServer":"","SSLConfigurationClient":"",
   "JVMArgs":"","JavaHome":"","ClassPath":"C:/dev/pathname/"}
```

Even though only the four required fields and `ClassPath` are specified in this example, the returned dynamic object includes all fields, assigning default values as necessary.

modifyServer()

The **modifyServer()** method accepts a list of settings contained in a %DynamicObject, and returns a %DynamicObject containing the modified definition. The method signature is:

```
modifyServer(serverDef As %DynamicObject) as %DynamicObject
```

The input dynamic object can be specified as either a JSON string or (as in this example) a set of property definitions:

```
set mod = {}
set mod.Name = "%Java Server"
set mod.JavaHome = "/Library/Java/Contents/Home/"
```

New values can be specified for any fields except `Name` and `Type`. Since this method operates on an existing definition, these properties cannot be modified.

modifyServer() will throw an error if you attempt to call it while the corresponding external server is running. You can use **isServerRunning()** to check on the server and **stopServer()** to stop it if necessary (see "Controlling Connections with the $system.external Interface" for more information):

```
set isRunning = $system.external.isServerRunning(mod.Name)
if (isRunning) $system.external.stopServer(mod.Name)
set modifiedserver = $system.external.modifyServer(mod)
```

The method returns a dynamic object containing all fields of the modified definition:

```
write modifiedserver.%ToJSON()

   {"Name":"%Java Server","FullName":"user3:IRIS_DS2:%Java Server","Type":"Java",
   "Port":4015,"LogFile":"","AllowedIPAddresses":"127.0.0.1",
   "ConnectionTimeout":5,"HeartbeatFailureAction":"R","HeartbeatFailureRetry":300,
   "HeartbeatFailureTimeout":30,"HeartbeatInterval":10,"InitializationTimeout":5,
   "UsePassphrase":0,"UseSharedMemory":1,"passphraseList":"",
   "SSLConfigurationServer":"","SSLConfigurationClient":"",
   "JVMArgs":"","JavaHome":"/Library/Java/somename/Contents/Home/","ClassPath":""}
```

deleteServer()

The **deleteServer()** method deletes an existing external server definition and returns a dynamic object containing the deleted settings. The method signature is:

```
deleteServer(serverName As %String) as %DynamicObject
```

deleteServer() will throw an error if you attempt to call it while the corresponding external server is running. You can use **isServerRunning()** to check on the server and **stopServer()** to stop it if necessary (see "Controlling Connections with the $system.external Interface" for more information):

```
set oldServer = "MyOldServer"
set isRunning = $system.external.isServerRunning(oldServer)
if (isRunning) $system.external.stopServer(oldServer)

set deletedDef = $system.external.deleteServer(oldServer)
write $system.external.serverExists(oldServer)

   0
```

The method returns a dynamic object containing all fields of the deleted definition:

```
write deletedDef.%ToJSON()

   {"Name":"MyOldServer","FullName":"user3:IRIS_DS2:MyOldServer","Type":"Java",
   "Port":5309,"LogFile":"","AllowedIPAddresses":"127.0.0.1",
   "ConnectionTimeout":5,"HeartbeatFailureAction":"R","HeartbeatFailureRetry":300,
   "HeartbeatFailureTimeout":30,"HeartbeatInterval":10,"InitializationTimeout":5,
   "UsePassphrase":0,"UseSharedMemory":1,"passphraseList":"",
   "SSLConfigurationServer":"","SSLConfigurationClient":"",
   "JVMArgs":"","JavaHome":"","ClassPath":""}
```

5

InterSystems External Server Requirements

All InterSystems External Servers are set to a predefined default configuration automatically on installation. If your language platforms are installed in default locations, no additional configuration should be necessary. If an external server fails to start with the predefined settings, the problem can almost always be corrected by setting the correct path to the supporting language platform (see "Troubleshooting External Server Definitions" at the end of this section for details).

This section provides information on the default location and supported versions for all Java, .NET, and Python external servers:

- InterSystems IRIS Requirements
- Java External Server Setup
- .NET External Server Setup
- Python External Server Setup
- Troubleshooting External Server Definitions
- Upgrading Object Gateway Code

5.1 InterSystems IRIS Requirements

To communicate with an InterSystems External Server, your instance of InterSystems IRIS must be version 2020.4 or higher. The Java, .NET, and Python External Servers are designed to work automatically if the default settings for those language platforms can be used. In some cases, minor changes to server configurations may be required (as described in the following sections).

Tip: **Locating *<install-dir>***

The path to your installation directory can be displayed at the Terminal by entering the following ObjectScript command:

```
write $SYSTEM.Util.InstallDirectory()
```

5.2 Java External Server Setup

A version of the Java External Server executable is provided for each supported version of Java. The correct version will usually be specified automatically for the default Java server configuration (`%Java Server`) during installation.

Each server version is located in a different subdirectory of *install-dir*\dev\java\bin where *install-dir* contains your preferred instance of InterSystems IRIS.

The following versions of the Java External Server are included in the standard installation:

- Java 8: *install-dir*\dev\java\lib\JDK18\intersystems-gateway-3.2.0.jar
- Java 11: *install-dir*\dev\java\lib\JDK11\intersystems-gateway-3.2.0.jar

If you wish to use a JVM other than the one specified in your `JAVA_HOME` setting, some extra configuration will be required to set the path to your desired language platform (see "Defining External Server Configurations for Java").

Note: The appropriate version of Java must be installed on your system in order to use these components. The InterSystems IRIS installation procedure does not install or upgrade any version of Java.

5.3 .NET External Server Setup

A version of the .NET External Server assembly is provided for each supported .NET version. The correct external server version will usually be specified automatically for the default .NET external server configuration (`%DotNet Server`) during installation.

All assemblies are supported on Windows, and .NET Core 2.1 is also supported on Linux and MacOS. Each version is located in a different subdirectory of *install-dir*\dev\dotnet\bin where *install-dir* contains your preferred instance of InterSystems IRIS. The following assemblies are available:

.NET Version 2.0:

- *install-dir*\dev\dotnet\bin\v2.0.50727\InterSystems.Data.Gateway.exe
- *install-dir*\dev\dotnet\bin\v2.0.50727\InterSystems.Data.Gateway64.exe

.NET Version 4.0:

- *install-dir*\dev\dotnet\bin\v4.0.30319\InterSystems.Data.Gateway.exe
- *install-dir*\dev\dotnet\bin\v4.0.30319\InterSystems.Data.Gateway64.exe

.NET Version 4.5:

- *install-dir*\dev\dotnet\bin\v4.5\InterSystems.Data.Gateway.exe
- *install-dir*\dev\dotnet\bin\v4.5\InterSystems.Data.Gateway64.exe

.NET Core 2.1:

- *install-dir*\dev\dotnet\bin\Core21\IRISGatewayCore21.dll

  See .NET Core 2 Installation and Configuration in *Using .NET with InterSystems Software* for Core 2.1 installation instructions.

In some applications, the .NET Framework assemblies may be used to load unmanaged code libraries. Both 32-bit and 64-bit assemblies are provided for each supported version, which makes it possible to create gateway applications for 64-bit Windows that can load 32-bit libraries.

If you wish to use a version other than the default for your system, some extra configuration will be required to set the path to your desired language platform (see "Defining External Server Configurations for .NET").

Note: A supported 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.

5.4 Python External Server Setup

The Python External Server requires Python 3.6.6 or higher. If a supported version of Python is available on your system, the correct path to the Python executable will usually be specified automatically for the default Python external server configuration (`Python Server`) during installation.

If a supported version of Python 3 is not the default for your system, some extra configuration will be required to set the path to your desired instance (see "Defining External Server Configurations for Python").

The installation file is intersystems_irispython-3.2.0-py3-none-any.whl, located in <install-dir>\dev\python\ (where <install-dir> is the root directory of your InterSystems IRIS instance). The directory may also contain other.whl files, which are not relevant to external servers and may be ignored. (You can also download the latest version of the installation file from the InterSystems IRIS Driver Packages page)

Install the Python External Server package with the following command:

```
python -m pip install --upgrade <path>\intersystems_irispython-3.2.0-py3-none-any.whl
```

Do not use the `--user` option.

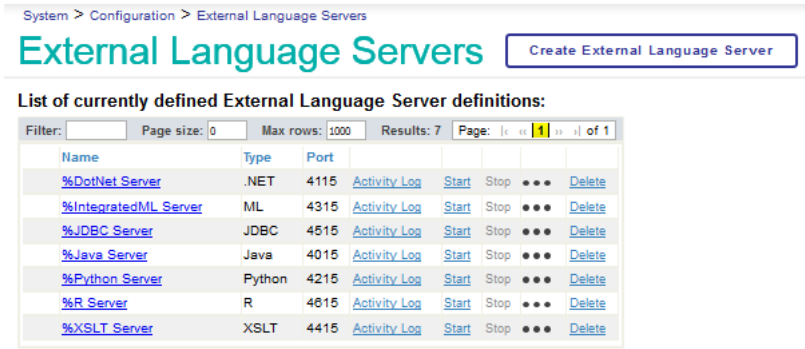
Note: A supported version of Python 3 must be installed on your system in order to use this package. The InterSystems IRIS installation procedure does not install or upgrade any version of Python.

5.5 Troubleshooting External Server Definitions

All supported languages require the language platform to be properly installed and available to InterSystems IRIS. If an external server does not start, error messages in the Activity Log will often point to the cause. The most common configuration problem is an invalid path to the language platform.

The *External Servers* page (System Administration > Configuration > Connectivity > External Servers) contains a listing for each default external server configuration:

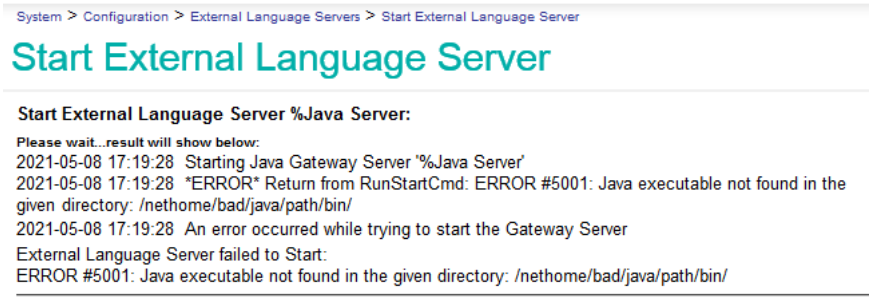
Figure 5–1: InterSystems External Servers (System Administration > Configuration > Connectivity)



The default external servers described in this document are `%DotNet Server`, `%Java Server`, and `%Python Server`. You can display all previous Start and Stop messages for an external server by clicking on the `Activity Log` link.

You can test an external server by clicking `Start`. This will display the startup commands, and will list error messages if the external server fails to start. For example, the following message indicates that the path to the Java executable is wrong:

Figure 5–2: Example of an error message on startup



To generate this example, the `%Java Server` Java Home setting was deliberately changed to \nethome\java\bad\path.

Note: The default configuration for the Java language is defined in `%Java Server`. Do not confuse it with `%JDBC Server`, which is a specialized configuration used only by certain older InterSystems interfaces.

Each language has an optional configuration setting that can be used to specify the language path (for details, see the language-specific Management Portal pages for Java, .NET, and Python in "Defining Configurations in the Management Portal").

5.6 Upgrading Object Gateway Code

External servers use an enhanced and simplified form of the older Dynamic Object Gateway technology. All Object Gateway features are still available, so upgrading your code is a simple matter of replacing certain class and method references. The following code demonstrates old and new ways to perform some common activities:

Starting the server and getting a Gateway object

The process of getting a Gateway object with an active server connection involved several calls to methods of two different ObjectScript classes:

```
  set status = ##class(%Net.Remote.Service).OpenGateway("JavaGate",.GatewayInfo)
  set name = GatewayInfo.Name
  set port = GatewayInfo.Port
  set server = GatewayInfo.Server
  if ('##class(%Net.Remote.Service).IsGatewayRunning(server,port,,.status)) {
    set status = ##class(%Net.Remote.Service).StartGateway(name)
  }
set gateway = ##class(%Net.Remote.Gateway).%New()
set status = gateway.%Connect(server, port, "USER")
```

With external servers, it's somewhat simpler:

```
set gateway = $system.external.getJavaGateway()
```

One call creates a Gateway object with all the appropriate settings, and automatically starts the connection to the external server. You can still access the external server manually (see "Starting and Stopping External Servers"), but this is unnecessary for most applications.

Specifying a class path and creating a proxy object

Object Gateway code required references to several different ObjectScript classes and a significant amount of server-specific information before a proxy could be created:

```
set path = ##class(%ListOfDataTypes).%New()
do path.Insert("C:\Dev\SomeClasses.jar")
do ##class(%Net.Remote.Service).OpenGateway("JavaGate",.GatewayInfo)
set gateway = ##class(%Net.Remote.Gateway).%New()
do gateway.%Connect(GatewayInfo.Server, GatewayInfo.Port, "USER",,path)
set proxy = ##class(%Net.Remote.Object).%New(gateway,"SomeClasses.ClassOne")
```

Once again, external server code is considerably simpler:

```
set gateway = getJavaGateway()
do gateway.addToPath("C:\Dev\SomeClasses.jar")
set proxy = gateway.new("SomeClasses.ClassOne")
```

The new **addToPath()** method also provides a simpler way to add multiple class paths.

Calling a class method

Instead of the old **%ClassMethod()**, external servers use the new Gateway **invoke()** method.

```
// Object Gateway
set num =
##class(%Net.Remote.Object).%ClassMethod(gateway,"Demo.ReverseGateway","factorial",num-1)

// external server
set num = gateway.invoke("Demo.ReverseGateway","factorial",num-1)
```

6

Quick Reference for the ObjectScript $system.external Interface

This is a quick reference to the ObjectScript $system.external interface, which provides ObjectScript with programmatic access to all InterSystems External Servers.

Note: This chapter is intended as a convenience for readers of this book, but it is not the definitive reference. For the most complete and up-to-date information, see the online Class Library documentation for %system.external.

You can get the same information at the command line by calling the $system.external.**Help()** method. For a list of all methods, call:

```
do $system.external.Help()
```

or for detailed information about a specific method *methodName*, call:

```
do $system.external.Help("methodName")
```

6.1 All methods by Usage

Gateway and Proxy Objects

See "Working with External Languages").

Creating Gateway objects

- getDotNetGateway() gets a connection to the default .NET External Server.
- getJavaGateway() gets a connection to the default Java External Server.
- getPythonGateway() gets a connection to the default Python External Server.

Creating proxy objects

- new() returns a new proxy object bound to an instance of the external class.
- addToPath() adds a path or paths to executables to the current language gateway path.

Accessing static methods and properties of external classes

- invoke() calls an external object method and gets any returned value.
- getProperty() gets the value of a static property from the external object.
- setProperty() sets the value of a static property in the external object.

Managing external servers

See "Managing External Server Connections"

- startServer() starts the server.
- stopServer() stops the server.
- isServerRunning() returns true if the requested server is running.
- getActivity() gets ActivityLog entries for the specified server.

Customizing external server configurations

See "Customizing External Server Definitions"

Getting information about external server definitions

- getServer() gets a dynamic object containing the server definition.
- getServers() gets the names of all existing external server definitions.
- getServerLanguageVersion() gets the configured external language version string for a server definition.
- serverExists() returns true if the server definition exists.

Managing external server definitions

- createServer() creates a new server definition.
- deleteServer() deletes an existing server definition.
- modifyServer() modifies an existing server definition.

6.2 $system.external Method Details

addToPath()

$system.external.**addToPath()** adds *path* to the current language gateway path, where the *path* string contains paths to one or more executables.

```
addToPath(path:%RawString)
```

returns: nothing

The *path* argument can be a simple string containing a single path (for Java, this can be a folder or a jar URL). Multiple paths can be added by passing a dynamic array or an instance of %Library.ListOfDataTypes containing the paths to be added.

This function throws an exception if an error is encountered.

parameters:

- `path` — string containing a path, or a %DynamicArray or %ListOfDataTypes containing multiple paths

createServer()

$system.external.**createServer()** creates a new external server definition. This function requires the %Admin_Manage resource.

```
createServer(serverDef As %DynamicObject) as %DynamicObject
```

parameters:

- `serverDef` — a %DynamicObject containing the new external server settings

see also: "Creating and Modifying External Server Definitions"

deleteServer()

$system.external.**deleteServer()** deletes an existing external server definition. This function requires the %Admin_Manage resource.

```
deleteServer(serverName As %String) as %DynamicObject
```

parameters:

- `serverName` — name of an existing external server definition

see also: "Creating and Modifying External Server Definitions"

getActivity()

$system.external.**getActivity()** returns a %DynamicArray containing the specified number of ActivityLog entries for the specified external server.

```
getActivity(serverName As %String, entryCount As %Integer = 10,
verbose As %Boolean = 0) as %Library.DynamicArray
```

parameters:

- `serverName` — name of an existing external server definition
- `entryCount` — number of ActivityLog entries to return
- `verbose` — if true, display entries on the current device

If verbose is true then those rows will also be displayed on the current device

see also: "Displaying the Activity Log"

getExternalLanguage()

$system.external.**getExternalLanguage()** returns the external language from the external server.

```
getExternalLanguage(externalServerName As %String) as %String
```

parameters:

- `externalServerName` — name of the currently connected server

see also: Gateway.**getExternalLanguage()**

getExternalLanguageVersion()

$system.external.**getExternalLanguageVersion()** returns the external language version from the external server.

```
getExternalLanguageVersion(externalServerName As %String) as %String
```

parameters:

- `externalServerName` — name of the currently connected server

see also: Gateway.**getExternalLanguageVersion()**

getGateway()

$system.external.**getGateway()** returns a new gateway connection to an external server.

```
getGateway(externalServer As %RawString) as %External.Gateway
```

parameters:

- `externalServer` — name of the server to be connected

This method does not retrieve an existing cached gateway connection. It always acquires a new gateway connection to the requested external server.

getDotNetGateway()

$system.external.**getDotNetGateway()** returns a new gateway connection to the default .NET External Server. Equivalent to `getGateway("%DotNet Server")` (see getGateway()).

```
getDotNetGateway() As %External.Gateway
```

This method does not retrieve an existing cached gateway connection. It always acquires a new gateway connection to the default .NET External Server.

see also: "Creating a Gateway and Using a Proxy Object"

getJavaGateway()

$system.external.**getJavaGateway()** returns a new gateway connection to the default Java External Server. Equivalent to `getGateway("%Java Server")` (see getGateway()).

```
getJavaGateway() As %External.Gateway
```

This method does not retrieve an existing cached gateway connection. It always acquires a new gateway connection to the default Java External Server.

see also: "Creating a Gateway and Using a Proxy Object"

getPythonGateway()

$system.external.**getPythonGateway()** returns a new gateway connection to the default Python External Server. Equivalent to `getGateway("%Python Server")` (see getGateway()).

```
getPythonGateway() As %External.Gateway
```

This method does not retrieve an existing cached gateway connection. It always acquires a new gateway connection to the default Python External Server.

see also: "Creating a Gateway and Using a Proxy Object"

getProperty()

$system.external.**getProperty()** returns the value of a static property from the external class.

```
getProperty(externalServerName As %String, externalClass As %String, propertyName As %String)
as %ObjectHandle
```

parameters:

- `externalServerName` — name of the currently connected server
- `externalClass` — name of the external class
- `propertyName` — name of the external property

see also: Gateway.**getProperty()**

getServer()

$system.external.**getServer()** returns a %DynamicObject containing the configuration settings from the specified external server definition.

```
getServer(serverName As %RawString) as %Library.DynamicObject
```

parameters:

- `serverName` — name of an existing external server definition

Note: the similarly named **getServers()** method returns the names of all defined external server configurations.

see also: "Using getServer() to Retrieve Configuration Settings"

getServerLanguageVersion()

$system.external.**getServerLanguageVersion()** returns the external language version string for an external server configuration.

```
getServerLanguageVersion(serverName As %RawString) as %String
```

parameters:

- `serverName` — name of an existing external server definition

This function does not establish a connection to the external server in most cases. It simply returns the language version from the configuration. This function may execute an external command but it does not start the external server.

see also: "Getting Other Information about Existing Definitions"

getServers()

$system.external.**getServers()** returns a %DynamicArray containing the names of all defined external server configurations.

```
getServers() as %Library.DynamicArray
```

Note: the similarly named **getServer()** method returns detailed configuration settings for one external server.

see also: "Getting Other Information about Existing Definitions"

Help()

$system.external.**Help()** returns a string containing a list of $system.external methods. If the optional *method* argument is specified, it returns detailed information for that method.

```
Help(method As %String = "") as %String
```

parameters:

- `methodName` — optional string containing the name of a method for which a detailed description should be returned.

see also: Gateway.**Help()**

invoke()

$system.external.**invoke()** invokes external code. If this method is called as an expression then it returns any value returned by the external code.

```
invoke(externalServerName As %String,externalClass As %String, externalMethod As %String,
args... As %RawString) as %String
```

parameters:

- `externalServerName` — name of the currently connected server
- `externalClass` — name of the external class
- `externalMethod` — name of the external method
- `args...` — zero or more arguments to the specified class constructor

If no value is returned by the external code then a `<COMMAND>` exception is thrown. The *externalClass* is the name of the external code container (Java or .NET class name, Python class or module name). The *externalMethod* is the name of the external unit of code (function, method, or other language-specific unit) to invoke from the *externalClass*. The return value is the value returned by the external code. If the external method does not return a value then **invoke()** does not return a value.

see also: Gateway.**invoke()**

isServerRunning()

$system.external.**isServerRunning()** returns `true` if the requested server is running.

```
isServerRunning(arg As %RawString) as %Boolean
```

parameters:

- `arg` — the external server name; either a string, or a dynamic object containing the server definition (as returned by **getServer()**).

see also: Gateway.**isServerRunning()**, "Starting and Stopping External Servers"

modifyServer()

$system.external.**modifyServer()** modifies settings in an existing external server definition. This function requires the %Admin_Manage resource.

```
modifyServer(serverDef As %DynamicObject) as %DynamicObject
```

parameters:

- `serverDef` — a %DynamicObject containing the modified external server settings

see also: "Creating and Modifying External Server Definitions"

new()

$system.external.**new()** returns a new proxy object bound to an instance of the external class. Pass *externalClass* and any additional constructor arguments necessary.

```
new(externalServerName As %String, externalClass As %String, args... As %RawString) as
%Net.Remote.Object
```

parameters:

- `externalServerName` — name of the currently connected server
- `externalClass` — name of the external class
- `args...` — zero or more arguments to the specified class constructor

see also: Gateway.**new()**

serverExists()

$system.external.**serverExists()** returns `true` if *serverName* is an existing external server definition.

```
serverExists(serverName As %RawString) as %Boolean
```

parameters:

- `serverName` — name of an existing external server definition

see also: "Getting Other Information about Existing Definitions"

setProperty()

$system.external.**setProperty()** sets the value of a static property in the external class.

```
setProperty(externalServerName As %String, externalClass As %String, propertyName As %String,
value As %RawString)
```

parameters:

- `externalServerName` — name of the currently connected server
- `externalClass` — name of the external class
- `propertyName` — name of the external property
- `value` — new value for the specified property

see also: Gateway.**setProperty()**

startServer()

$system.external.**startServer()** starts the external server.

```
startServer(serverName As %String) as %Boolean
```

parameters:

- `serverName` — name of an existing external server definition

see also: "Starting and Stopping External Servers"

stopServer()

$system.external.**stopServer()** stops the external server.

```
stopServer(serverName As %String, verbose As %Boolean = 0) as %Boolean
```

parameters:

- `serverName` — name of an existing external server definition
- `verbose` — if true, display entries on the current device

see also: "Starting and Stopping External Servers"

6.3 Gateway Methods

Gateway objects have a set of methods that work exactly like $system.external methods of the same name, except that they do not take a server name as the first argument. For example, the following statement invokes the Java **Date()** method by calling $system.external.**new()** with `"%Java_Server"` as the first argument:

```
set date = $system.external.new("%Java_Server","java.util.Date")
```

The following almost identical statement actually uses $system.external.**getGateway()** to create a Java Gateway object, and then chains a call to the Gateway version of **new()**:

```
set date = $system.external.getGateway("%Java Server").new("java.util.Date")
```

This example can be simplified by getting the Gateway object with **getJavaGateway()**, which doesn't require a server name argument:

```
set date = $system.external.getJavaGateway().new("java.util.Date")
```

All three of these examples are functionally identical.

6.3.1 Gateway Method Details

addToPath()

Gateway.**addToPath()** adds *path* to the current language gateway path, where the *path* string contains paths to one or more executables.

```
addToPath(path As %RawString)
```

parameters:

- `path` — string containing a path, or a %DynamicArray or %ListOfDataTypes containing multiple paths

The *path* argument can be a simple string containing a single path (for Java, this can be a folder or a jar URL). Multiple paths can be added by passing a dynamic array or an instance of %Library.ListOfDataTypes containing the paths to be added.

This function throws an exception if an error is encountered.

see also: $system.external.**addToPath()**

disconnect()

Gateway.**disconnect()** disconnects the connection to the external server if it exists.

```
disconnect()
```

see also: $system.external.**disconnect()**

getExternalLanguage()

Gateway.**getExternalLanguage()** returns the external language from the external server.

```
getExternalLanguage() as %String
```

see also: $system.external.**getExternalLanguage()**

getExternalLanguageVersion()

Gateway.**getExternalLanguageVersion()** returns the external language version from the external server.

```
getExternalLanguageVersion() as %String
```

see also: $system.external.**getExternalLanguageVersion()**

getProperty()

Gateway.**getProperty()** returns the value of a static property from the external class.

```
getProperty(externalClass As %String, propertyName As %String) as %ObjectHandle
```

parameters:

- `externalClass` — name of the external class
- `propertyName` — name of the external property

see also: $system.external.**getProperty()**

Help()

Gateway.**Help()** returns a string containing a list of Gateway methods. If the optional *method* argument is specified, it returns detailed information for that method.

```
Help(method As %String = "") as %String
```

parameters:

- `methodName` — optional string containing the name of a method for which a detailed description should be returned.

see also: $system.external.**Help()**

invoke()

Gateway.**invoke()** invokes external code. If this method is called as an expression then it returns any value returned by the external code.

```
invoke(externalClass As %String, externalMethod As %String,    args... As %RawString) as %String
```

parameters:

- `externalClass` — name of the external class
- `externalMethod` — name of the external method
- `args...` — zero or more arguments to the specified class constructor

If no value is returned by the external code then a `<COMMAND>` exception is thrown. The *externalClass* is expected to be passed as the name of the external code container. For Java or .NET this is the class name. For Python this can be the name of a class or module. The *externalMethod* is the name of the external unit of code (function, method, or other language-specific unit) to invoke in the *externalClass* container. The return value is the value returned by the external code. If the external method does not return a value then **invoke()** does not return a value.

see also: $system.external.**invoke()**

isServerRunning()

Gateway.**isServerRunning()** returns `true` if the server for this Gateway is running.

```
isServerRunning() as %Boolean
```

see also: $system.external.**isServerRunning()**, "Starting and Stopping External Servers"

new()

Gateway.**new()** returns a new instance of %Net.Remote.Object that is bound to an instance of the external class. Pass *externalClass* and any additional constructor arguments necessary.

```
new(externalClass As %String, args... As %RawString) as %Net.Remote.Object
```

parameters:

- `externalClass` — name of the external class
- `args...` — zero or more arguments to the specified class constructor

see also: $system.external.**new()**, "Creating a Gateway and Using a Proxy Object"

setProperty()

Gateway.**setProperty()** sets the value of a static property in the external class.

```
setProperty(externalClass As %String, propertyName As %String, value As %RawString)
```

parameters:

- `externalClass` — name of the external class
- `propertyName` — name of the external property
- `value` — new value for the specified property

see also: $system.external.**setProperty()**