Using .NET with InterSystems Software

Version 2024.1
2024-07-02

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

Using .NET with InterSystems Software
InterSystems IRIS Data Platform Version 2024.1 2024-07-02

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 Using .NET with InterSystems Software

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

InterSystems IRIS® provides a wide variety of robust .NET connectivity options, including lightweight SDKs that provide database access via .NET ADO, .NET objects, or InterSystems multidimensional storage, and gateways that give InterSystems IRIS server applications direct access to .NET applications and external databases.

This document covers the following topics:

- Connecting to the Database with .NET provides detailed information about database connections (including connection pooling).
- .NET Requirements and Configuration provides information on setup and configuration for all InterSystems .NET solutions.
- Using the ADO.NET Managed Provider gives concrete examples using the InterSystems implementation of the ADO.NET Managed Provider API.
- Quick Reference for the .NET Managed Provider — lists and describes all methods and properties discussed in these topics.

Related Documents

The following documents contain detailed information on other .NET solutions provided by InterSystems IRIS:

- *Using the Native SDK for .NET* describes how to use the .NET Native SDK to access resources formerly available only through ObjectScript.
- *Persisting .NET Objects with InterSystems XEP* describes how to use the Event Persistence SDK (XEP) for rapid .NET object persistence.
- *Using the InterSystems ODBC Driver* describes how to use the ODBC driver to access InterSystems databases from external applications or to access external ODBC data sources from InterSystems products.

2

Connecting to the Database with .NET

This section describes how to create a connection between your .NET client application and an InterSystems server using an IRISConnection object.

2.1 Establishing Connections with .NET

The code below establishes a connection to a namespace named USER. See "Connection Parameter Options" for a complete list of parameters that can be set when instantiating a connection object.

The following simple method could be called to start a connection:

Add code to Instantiate the connection

```
public IRISConnection Conn;
private void CreateConnection(){
  try {
    Conn = new IRISConnection();
    Conn.ConnectionString =
      "Server=localhost; Port=51773; Namespace=USER;"
      + "Password=SYS; User ID=_SYSTEM;";
    Conn.Open();
  }
  catch (Exception eConn){
    MessageBox.Show("CreateConnection error: " + eConn.Message);
  }
}
```

Once the object has been created, it can be shared among all the classes that need it. The connection object can be opened and closed as necessary. You can do this explicitly by using **Conn.Open()** and **Conn.Close()**. If you are using an ADO.NET Dataset, instances of DataAdapter will open and close the connection automatically, as needed.

2.1.1 Troubleshooting .NET Client Connections

A .NET client that attempts to connect to an InterSystems server needs the following information:

- A URL that provides the server IP address, port number, and namespace.
- If the connection uses passwords, you must specify a case-sensitive username and password.

Check the following points if you have any problems:

- Make sure that the server process is installed and running.
- Make sure that you know the IP address of the machine on which the server process is running.

- Make sure that you know the TCP/IP port number on which the server is listening.
- Make sure that you have a valid username and password to use to establish a connection. (You can manage usernames and passwords using the Management Portal: System Administration > Security > Users).
- Make sure that your connection URL includes a valid namespace. This should be the namespace containing the classes and data your program uses.

2.2 .NET Shared Memory Connections

The standard ADO .NET connection to a remote InterSystems IRIS instance is over TCP/IP. To maximize performance, InterSystems IRIS also offers a shared memory connection for .NET applications running on the same machine as an InterSystems IRIS instance. This connection avoids potentially expensive calls into the kernel network stack, providing optimal low latency and high throughput for .NET operations.

If a connection specifies server address `localhost` or `127.0.0.1`, shared memory will be used by default. TCP/IP will be used if the actual machine address is specified. The connection will automatically fall back to TCP/IP if the shared memory device fails or is not available.

Shared memory can be disabled in the connection string by setting the *SharedMemory* property to `false`. For example, the following connection string will not use shared memory, even though the server address is specified as `localhost`.

```
"Server=localhost;Port=51774;Namespace=user;Password = SYS;User ID = _system;SharedMemory=false"
```

Shared memory is not used for TLS connections. The log will include information on whether a shared memory connection was attempted and if it was successful.

Note: **Shared memory connections do not work across container boundaries**

InterSystems does not currently support shared memory connections between two different containers. If a client tries to connect across container boundaries using `localhost` or `127.0.0.1`, the connection mode will default to shared memory, causing it to fail. This applies regardless of whether the Docker `--network host` option is specified. You can guarantee a TCP/IP connection between containers either by specifying the actual hostname for the server address, or by disabling shared memory in the connection string (as demonstrated above).

Shared memory connections can be used without problems when the server and client are in the same container.

2.3 Connection Pooling with .NET

Connection pooling is on by default. The following connection string parameters can be used to control various aspects of connection pooling:

- `Pooling` — Defaults to `true`. Set `Pooling` to `false` to create a connection with no connection pooling.
- `Min Pool Size` and `Max Pool Size` — Default values are `0` and `100`. Set these parameters to specify the maximum and minimum (initial) size of the connection pool for this specific connection string.
- `Connection Reset` and `Connection Lifetime` — Set `Connection Reset` to `true` to turn on the pooled connection reset mechanism. `Connection Lifetime` specifies the number of seconds to wait before resetting an idle pooled connection. The default value is `0`.

For example, the following connect string sets the initial size of the connection pool to 2 and the maximum number of connections to 5, and activates connection reset with a maximum connection idle time of 3 seconds:

```
Conn.ConnectionString =
  "Server = localhost;"
  + " Port = 51774;"
  + " Namespace = USER;"
  + " Password = SYS;"
  + " User ID = _SYSTEM;"
  + " Min Pool Size = 2;"
  + " Max Pool Size = 5;"
  + " Connection Reset = true;"
  + " Connection Lifetime = 3;";
```

See the “Quick Reference for the .NET Managed Provider” for more details on the various connection pooling methods and properties.

3

.NET Requirements and Configuration

This section provides information on the InterSystems .NET client assemblies. It also provides instructions for Setting Up the Entity Framework Provider.

3.1 Managed Providers in the IRISClient assembly

The IRISClient .NET assembly provides two different but complementary Providers that support access to InterSystems databases from a .NET application:

- *The ADO.NET Managed Provider* — is InterSystems implementation of the ADO.NET data access interface. It provides easy relational access to data using the standard ADO.NET Managed Provider classes (see "Using ADO.NET Managed Provider Classes").
- *The Entity Framework Provider* — is the InterSystems implementation of the object-relational mapping (ORM) framework for ADO.NET. It enables .NET developers to work with relational data using domain-specific objects (see Setting Up the Entity Framework Provider).

  For more information, see the course on Using Entity Framework with InterSystems IRIS Data Platform.

The IRISClient assembly is implemented using .NET managed code throughout, making it easy to deploy within a .NET environment. It is thread-safe and can be used within multithreaded .NET applications.

3.2 Supported .NET Versions

For a list of supported .NET versions, see Supported .NET Frameworks in *InterSystems Supported Platforms*.

3.2.1 Unsupported IRISClient Assemblies

InterSystems IRIS has dropped support for several versions of .NET that are no longer in support by Microsoft (.NET Framework 2.0, 4.0 and 4.5, and .NET Core 1.0 and 2.1). Older projects that use the path to the dll location will need to update the path to correspond to the new assembly file location. For example, a previous path would be:

```
\<IRIS install location>\dev\dotnet\bin\v4.5\InterSystems.Data.IRISClient.dll
```

That location will no longer exist under the new installation, and should be changed to:

```
\<IRIS install location>\dev\dotnet\bin\v4.6.2\InterSystems.Data.IRISClient.dll
```

In terms of compatibility between versions, the newer 4.6.2 version is backwards compatible and the applications will run on systems that have any .NET Framework 4.x installed.

However, .NET Framework is not forwards-compatible, so if your application targets .NET Framework 4.5 specifically, it cannot use .NET Framework 4.6.2 client libraries as a dependency. In this case, your options are:

- Change the target framework of the application to be at least 4.6.2. .NET Framework 4.5 has been out of support by Microsoft since 2016, so this will also ensure users are using a supported language version.
- Use the .NET Framework 3.5 version of the library. You may lose access to certain features or functionality introduced in version 4.0.
- Continue to use an older version of the client library that targets 4.5. Older versions will not contain the latest bug fixes or functionality, but you will not need to modify the dependencies of your application. This is a temporary solution, since upgrades to future versions the InterSystems IRIS server will eventually make them incompatible with the older clients.

3.3 Configuring the IRISClient Assembly

Support is implemented in the IRISClient assembly, using .NET managed code throughout, making it easy to deploy within a .NET environment. IRISClient is thread-safe and can be used within multithreaded .NET applications. This section provides specifies requirements, and provides instructions for installing the IRISClient assembly and configuring Visual Studio.

3.3.1 Requirements

- A supported version of .NET or .NET Framework
- Visual Studio 2013 or higher.

InterSystems IRIS is not required on computers that run your .NET client applications, but the clients must have a TCP/IP connection to an InterSystems server and must be running a supported version of .NET or .NET Framework.

3.3.2 IRISClient Assembly Setup

The IRISClient assembly (InterSystems.Data.IRISClient.dll) is installed along with the rest of InterSystems IRIS, and requires no special preparation.

- When installing InterSystems IRIS in Windows, select the `Setup Type: Development` option.
- If InterSystems IRIS has been installed with security option 2, open the Management Portal and go to System Administration > Security > Services, select `%Service_CallIn`, and make sure the `Service Enabled` box is checked. If you installed InterSystems IRIS with security option 1 (minimal) it should already be checked.

Important: **Setup for Cloud Service Installations**

If you are not running InterSystems IRIS on a local installation, you may have to download and install the client driver manually. See Connecting Your Application to InterSystems IRIS for information on this option.

3.3.3 Declaring IRISClient in a .NET Application

To use the IRISClient assembly in a .NET application, you must add a `Using` statement for the InterSystems.Data.IRISClient.dll assembly before the beginning of your application's namespace.

Both the using statement and the following namespace are required:

```
using InterSystems.Data.IRISClient;
namespace YourNameSpace {
  ...
}
```

There is a separate version of InterSystems.Data.IRISClient.dll for each supported version of .NET and .NET Framework. See "Supported .NET Frameworks" for details.

3.3.4 Configuring Visual Studio

To add a IRISClient assembly reference to a project:

1. From the Visual Studio main menu, select `Project > Add Reference`
2. In the `Add Reference` window, click on `Browse...`
3. Browse to the subdirectory of <iris-install-dir>\dev\dotnet\bin that contains the assembly for the version of .NET used in your project (as listed in the previous section), select InterSystems.Data.IRISClient.dll, and click `OK`.
4. In the Visual Studio Solution Explorer, the InterSystems.Data.IRISClient assembly should now be listed under References.

3.4 Setting Up the Entity Framework Provider

Entity Framework is an object-relational mapper that enables .NET developers to work with relational data using domain-specific objects. The InterSystems Entity Framework Provider enables you to use Entity Framework 6 technology to access an InterSystems database. See Using Entity Framework with InterSystems IRIS Data Platform for a tutorial with working examples using the Entity Framework.

Follow the instructions in this section to configure the InterSystems Entity Framework Provider.

3.4.1 System Requirements

To use Entity Framework Provider with InterSystems IRIS, the following software is required:

- Visual Studio 2013 or later (first supported release is VS 2013 Professional/Ultimate with update 5).
- A supported version of .NET Framework (.NET Core and later .NET versions are not supported).
- InterSystems IRIS Entity Framework Provider distribution, described in the following section.

3.4.2 Creating the IrisEF Directory

The InterSystems IRIS Entity Framework Provider distribution file is IrisEF.zip, located in *install-dir*\dev\dotnet\bin\v4.0.30309.

1. Create a new directory named *install-dir*\dev\dotnet\bin\v4.0.30309\IrisEF.
2. Extract the contents of IrisEF.zip to the new directory.

This .zip file contains the following files, which you use in the setup instructions:

- setup.cmd, which installs the DLLs InterSystems.Data.IRISClient.dll and InterSystems.Data.IRISVSTools.dll.
- Nuget\InterSystems.Data.Entity6.4.5.0.0.nupkg which installs the Entity Framework Provider.
- CreateNorthwindEFDB.sql which is used to create a sample database (see Using Entity Framework with InterSystems IRIS Data Platform for more information).

3.4.3 Configure Visual Studio and install EF Provider

Important: If you are running VS 2013 or 2015, reverse steps 2 and 3 below: first run setup.cmd, then run `devenv /setup`.

1. Move to the new IrisEF directory. The following instructions assume that IrisEF is the current directory.
2. Set up the Visual Studio development environment:
   - In Windows, select All Programs > Visual Studio 201x > Visual Studio Tools.
   - In the displayed Windows Explorer folder, right-click Developer Command Prompt for VS201x > Run as Administrator and enter:

     ```
     devenv /setup
     ```

   This command repopulates the environment setting from the registry key that specifies the path to your version of Visual Studio.
3. At the command prompt, run setup.cmd. This installs InterSystems Entity Framework Provider files InterSystems.Data.IRISClient.dll and InterSystems.Data.IRISVSTools.dll.

3.4.4 Copy Files to Visual Studio

Copy the following files from IrisEF subdirectory IrisEF\Templates to Visual Studio:

- SSDLToIrisSQL.tt
- GenerateIrisSQL.Utility.ttinclude

Copy from <*iris-install-dir*>\dev\dotnet\bin\v4.0.30319\IrisEF\Templates

to <*VisualStudio-install-dir*>\Common7\IDE\Extensions\Microsoft\Entity Framework Tools\DBGen

3.4.5 Connect Visual Studio to the Server

To connect Visual Studio to an InterSystems database instance, follow the steps below:

1. Open Visual Studio and select View > Server Explorer.
2. Right-click **Data Connections** and select **Add Connection**. In the Add Connection Dialog:
   a. Select **Data source** as `InterSystems IRIS Data Source (.Net Framework Data Provider for InterSystems IRIS)`
   b. Select **Server**
   c. Enter **Username** and **password**. Click **Connect**.
   d. Select a namespace from the list. Click **OK**.

3.4.6 Configure the NuGet Local Repository

Follow these steps to configure the Package Manager to find the local NuGet repository:

1. Create a directory as a NuGet repository if you have not already done so. You can use any name and location. For example, you could create directory `NuGet Repository` in the default Visual Studio project directory (*<yourdoclibraryVS201x>*\Projects).
2. Copy the InterSystems.Data.Entity6.4.5.0.0.nupkg file from IrisEF subdirectory IrisEF\Nuget\ to your NuGet repository directory. Click **OK**.
3. In Visual Studio, select Project > Manage Nuget Packages > Settings > Package Manager > Package Sources.
4. Click the plus sign**+**. Enter the path that contains InterSystems.Data.Entity6.4.5.0.0.nupkg. Click **OK**

4

Using the ADO.NET Managed Provider

ADO.NET needs no introduction for experienced .NET database developers, but it can be useful even if you only use it occasionally for small utility applications. This section is a quick overview of ADO.NET that demonstrates how to do simple database queries and work with the results.

The InterSystems ADO.NET Managed Provider allows your .NET projects to access InterSystems databases with fully compliant versions of generic ADO.NET Managed Provider classes such as Connection, Command, CommandBuilder, DataReader, and DataAdapter. See "Connecting Your Application to InterSystems IRIS" for a complete description of how to connect your .NET application with InterSystems IRIS. The following classes are InterSystems-specific implementations of the standard ADO.NET Managed Provider classes:

- IRISConnection — Represents the connection between your application and the databases in a specified InterSystems namespace. See "Connecting to the InterSystems Database" for more information on how to use IRISConnection.
- IRISCommand — Encapsulates an SQL statement or stored procedure to be executed against databases in the namespace specified by a IRISConnection.
- IRISCommandBuilder — Automatically generates SQL commands that reconcile a database with changes made by objects that encapsulate a single-table query.
- IRISDataReader — Provides the means to fetch the result set specified by a IRISCommand. A IRISDataReader object provides quick forward-only access to the result set, but is not designed for random access.
- IRISDataAdapter — Encapsulates a result set that is mapped to data in the namespace specified by a IRISConnection. It is used to fill an ADO.NET DataSet and to update the database, providing an effective random access connection to the resultset.

This chapter gives some concrete examples of code using InterSystems ADO.NET Managed Provider classes. The following subjects are discussed:

- Introduction to ADO.NET Managed Provider Classes — provides a simple demonstration of how InterSystems ADO.NET Managed Provider classes are used.
- Using IRISCommand and IRISDataReader — demonstrates how to execute a simple read-only query.
- Using SQL Queries with IRISParameter — demonstrates passing a parameter to a query.
- Using IRISDataAdapter and IRISCommandBuilder — changing and updating data.
- Using Transactions — demonstrates how to commit or rollback transactions.

4.1 Introduction to ADO.NET Managed Provider Classes

A project using the InterSystems implementations of ADO.NET Managed Provider classes can be quite simple. Here is a complete, working console program that opens and reads an item from the `Sample.Person` database:

```
using System;
using InterSystems.Data.IRISClient;

namespace TinySpace {
  class TinyProvider {
    [STAThread]
    static void Main(string[] args) {

      string connectionString = "Server = localhost; Port = 51783; " +
        "Namespace = USER; Password = SYS; User ID = _SYSTEM;";
      using IRISConnection conn = new IRISConnection(connectionString);
      conn.Open();

      using IRISCommand command = conn.CreateCommand();
      command.CommandText = "SELECT * FROM Sample.Person WHERE ID = 1";
      IRISDataReader reader = command.ExecuteReader();
      while (reader.Read()) {
        Console.WriteLine($"TinyProvider output:\r\n " +
          $"{reader[reader.GetOrdinal("ID")]}: {reader[reader.GetOrdinal("Name")]}");
      }
      reader.Close();

    } // end Main()
  } // end class TinyProvider
}
```

This project contains the following important features:

- The `Using` statements provide access to the IRISClient assembly. A namespace must be declared for the client code:

  ```
  using InterSystems.Data.IRISClient;
  ```

  ```
  namespace TinySpace {
  ```

- The IRISConnection *conn* object is used to create and open a connection to the USER namespace. The *conn* object is created with a `using` declaration to ensure that it will always be properly closed and disposed:

  ```
  string connectionString = "Server = localhost; Port = 51783; " +
    "Namespace = USER; Password = SYS; User ID = _SYSTEM;";
  using IRISConnection conn = new IRISConnection(connectionString);
  conn.Open();
  ```

- The IRISCommand *command* object uses the IRISConnection object and an SQL statement to open the instance of `Sample.Person` that has an `ID` equal to `1`.

  ```
  using IRISCommand command = conn.CreateCommand();
  command.CommandText = "SELECT * FROM Sample.Person WHERE ID = 1";
  ```

- The IRISDataReader object is used to access the data items in the row:

  ```
  IRISDataReader reader = command.ExecuteReader();
  while (reader.Read()) {
    Console.WriteLine($"TinyProvider output:\r\n " +
      $"{reader[reader.GetOrdinal("ID")]}: {reader[reader.GetOrdinal("Name")]}");
  }
  reader.Close();
  ```

4.2 Using IRISCommand and IRISDataReader

Simple read-only queries can be performed using IRISCommand and IRISDataReader. Like all database transactions, such queries also require an open IRISConnection object.

In this example, an SQL query string is passed to a new IRISCommand object, which will use the existing connection:

```
string SQLtext = "SELECT * FROM Sample.Person WHERE ID < 10";
IRISCommand Command = new IRISCommand(SQLtext, Conn);
```

Results of the query are returned in a IRISDataReader object. Properties are accessed by referring to the names of columns specified in the SQL statement.

```
IRISDataReader reader = Command.ExecuteReader();
while (reader.Read()) {
  Console.WriteLine(
    reader[reader.GetOrdinal("ID")] + "\t"
  + reader[reader.GetOrdinal("Name")] + "\r\n\t"
  + reader[reader.GetOrdinal("Home_City")] + " "
  + reader[reader.GetOrdinal("Home_State")] + "\r\n");
};
```

The same report could be generated using column numbers instead of names. Since IRISDataReader objects can only read forward, the only way to return to beginning of the data stream is to close the reader and reopen it by executing the query again.

```
reader.Close();
reader = Command.ExecuteReader();
while (reader.Read()) {
  Console.WriteLine(
    reader[0] + "\t"
  + reader[4] + "\r\n\t"
  + reader[7] + " "
  + reader[8] + "\n");
}
```

4.3 Using SQL Queries with IRISParameter

The IRISParameter object is required for more complex SQL queries. The following example selects data from all rows where `Name` starts with a string specified by the IRISParameter value:

```
string SQLtext =
    "SELECT ID, Name, DOB, SSN "
  + "FROM Sample.Person "
  + "WHERE Name %STARTSWITH ?"
  + "ORDER BY Name";
IRISCommand Command = new IRISCommand(SQLtext, Conn);
```

The parameter value is set to get all rows where `Name` starts with A, and the parameter is passed to the IRISCommand object:

```
IRISParameter Name_param =
  new IRISParameter("Name_col", IRISDbType.NVarChar);
Name_param.Value = "A";
Command.Parameters.Add(Name_param);
```

Note: Be default, the SQL statement is not validated before being executed on the Server, since this would require two calls to the Server for each query. If validation is desirable, call IRISCommand**.Prepare()** to validate the syntax for the SQL statement against the server.

A IRISDataReader object can access the resulting data stream just as it did in the previous example:

```
IRISDataReader reader = Command.ExecuteReader();
while (reader.Read()) {
  Console.WriteLine(
    reader[reader.GetOrdinal("ID")] + "\t"
  + reader[reader.GetOrdinal("Name")] + "\r\n\t"
  + reader[reader.GetOrdinal("DOB")] + " "
  + reader[reader.GetOrdinal("SSN")] + "\r\n");
};
```

4.4 Using IRISDataAdapter and IRISCommandBuilder

The IRISCommand and IRISDataReader classes are inadequate when your application requires anything more than sequential, read-only data access. In such cases, the IRISDataAdapter and IRISCommandBuilder classes can provide full random read/write access. The following example uses these classes to get a set of `Sample.Person` rows, read and change one of the rows, delete a row and add a new one, and then save the changes to the database.

The IRISDataAdapter constructor accepts an SQL command and a IRISConnection object as parameters, just like a IRISCommand. In this example, the resultset will contain data from all Sample.Person rows where `Name` starts with `A` or `B`. The `Adapter` object will map the resultset to a table named `Person`:

```
string SQLtext =
    " SELECT ID, Name, SSN "
  + " FROM Sample.Person "
  + " WHERE Name < 'C' "
  + " ORDER BY Name ";
IRISDataAdapter Adapter = new IRISDataAdapter(SQLtext, Conn);
Adapter.TableMappings.Add("Table", "Person");
```

A IRISCommandBuilder object is created for the `Adapter` object. When changes are made to the data mapped by the `Adapter` object, `Adapter` can use SQL statements generated by `Builder` to update corresponding items in the database:

```
IRISCommandBuilder Builder = new IRISCommandBuilder(Adapter);
```

An ADO DataSet object is created and filled by `Adapter`. It contains only one table, which is used to define the `PersonTable` object.

```
System.Data.DataSet DataSet = new System.Data.DataSet();
Adapter.Fill(DataSet);
System.Data.DataTable PersonTable = DataSet.Tables["Person"];
```

A simple **foreach** command can be used to read each row in `PersonTable`. In this example, we save `Name` in the first row and change it to `"Fudd, Elmer"`. When the data is printed, all names will be in alphabetical order except the first, which now starts with `F`. After the data has been printed, the first `Name` is reset to its original value. Both changes were made only to the data in `DataSet`. The original data in the database has not yet been touched.

```
if (PersonTable.Rows.Count > 0) {
  System.Data.DataRow FirstPerson = PersonTable.Rows[0];
  string OldName = FirstPerson["Name"].ToString();
  FirstPerson["Name"] = "Fudd, Elmer";

  foreach (System.Data.DataRow PersonRow in PersonTable.Rows) {
    Console.WriteLine("\t"
      + PersonRow["ID"] + ":\t"
      + PersonRow["Name"] + "\t"
      + PersonRow["SSN"]);
  }
  FirstPerson["Name"] = OldName;
}
```

The following code marks the first row for deletion, and then creates and adds a new row. Once again, these changes are made only to the `DataSet` object.

```
FirstPerson.Delete();

System.Data.DataRow NewPerson = PersonTable.NewRow();
NewPerson["Name"] = "Budd, Billy";
NewPerson["SSN"] = "555-65-4321";
PersonTable.Rows.Add(NewPerson);
```

Finally, the **Update()** method is called. `Adapter` now uses the IRISCommandBuilder code to update the database with the current data in the `DataSet` object's `Person` table.

```
Adapter.Update(DataSet, "Person");
```

4.5 Using Transactions

The Transaction class is used to specify an SQL transaction (see "Transaction Processing" in *Using InterSystems SQL* for an overview of how to use transactions). In the following example, transaction `Trans` will fail and be rolled back if `SSN` is not unique.

```
IRISTransaction Trans =
  Conn.BeginTransaction(System.Data.IsolationLevel.ReadCommitted);
try {
  string SQLtext = "INSERT into Sample.Person(Name, SSN) Values(?,?)";
  IRISCommand Command = new IRISCommand(SQLtext, Conn, Trans);

  IRISParameter Name_param =
    new IRISParameter("name", IRISDbType.NVarChar);
  Name_param.Value = "Rowe, Richard";
  Command.Parameters.Add(Name_param);

  IRISParameter SSN_param =
    new IRISParameter("ssn", IRISDbType.NVarChar);
  SSN_param.Value = "234-56-3454";
  Command.Parameters.Add(SSN_param);

  int rows = Command.ExecuteNonQuery();
  Trans.Commit();
  Console.WriteLine("Added record for " + SSN_param.Value.ToString());
}
catch (Exception eInsert) {
  Trans.Rollback();
  WriteErrorMessage("TransFail", eInsert);
}
```

5

Quick Reference for the .NET Managed Provider

This chapter is a quick reference for the following extended classes and options:

- Class IRISPoolManager — methods related to InterSystems connection pooling.
- Class IRISConnection — methods for clearing connection pools.
- Connection Parameter Options — lists all supported connection parameters.

5.1 Class IRISPoolManager

The IRISClient.IRISPoolManager class can be used to monitor and control connection pooling programmatically. The following static methods are available:

ActiveConnectionCount()

```
int count = IRISPoolManager.ActiveConnectionCount();
```

Total number of established connections in all pools. Count includes both idle and in-use connections.

IdleCount()

```
int count = IRISPoolManager.IdleCount();
```

Total number of idle connections in all the pools.

```
int count = IRISPoolManager.IdleCount(conn);
```

Total number of idle connections in the pool associated with connection object *conn*.

InUseCount()

```
int count = IRISPoolManager.InUseCount();
```

Total number of in-use connections in all pools.

```
int count = IRISPoolManager.InUseCount(conn);
```

Total number of in-use connections in the pool associated with connection object *conn*.

RecycleAllConnections()

```
IRISPoolManager.RecycleAllConnections(bool remove);
```

Recycles connections in all pools

```
IRISPoolManager.RecycleConnections(conn,bool remove)
```

Recycles connections in the pool associated with connection object *conn*.

RemoveAllIdleConnections()

```
IRISPoolManager.RemoveAllIdleConnections();
```

Removes idle connections from all connection pools.

RemoveAllPoolConnections()

```
IRISPoolManager.RemoveAllPoolConnections();
```

Deletes all connections and removes all pools, regardless of what state the connections are in.

5.2 Class IRISConnection

ClearPool()

```
IRISConnection.ClearPool(conn);
```

Clears the connection pool associated with connection `conn`.

ClearAllPools()

```
IRISConnection.ClearAllPools();
```

Removes all connections in the connection pools and clears the pools.

5.3 Connection Parameter Options

The following tables describe all parameters that can be used in a connection string.

- Required Parameters
- Connection Pooling Parameters
- Other Connection Parameters

5.3.1 Required Parameters

The following parameters are required for all connection strings (see “Creating a Connection”).

server

alternate names: ADDR, ADDRESS, DATA SOURCE, DATASOURCE, HOST, NETWORK ADDRESS, NETWORKADDRESS

IP address or host name. For example: `Server = localhost`

port

Specifies the TCP/IP port number for the connection. For example: `Port = 51774`

namespace

alternate names: DATABASE, INITIAL CATALOG

Specifies the namespace to connect to. For example: `Namespace = USER`

password

alternate name: PWD

User's password. For example: `Password = SYS`

user id

alternate names: USERID, UID, USER, USERNAME, USR

Set user login name. For example: `User ID = _SYSTEM`

5.3.2 Connection Pooling Parameters

The following parameters define various aspects of connection pooling (see "Connection Pooling").

connection lifetime

alternate name: CONNECTIONLIFETIME

The length of time in seconds to wait before resetting an idle Pooled connection when the connection reset mechanism is on. Default is `0`.

connection reset

alternate name: CONNECTIONRESET

Turn on Pooled connection reset mechanism (used with CONNECTION LIFETIME). Default is `false`.

max pool size

alternate name: MAXPOOLSIZE

Maximum size of connection pool for this specific connection string. Default is `100`.

min pool size

alternate name: MINPOOLSIZE

Minimum or initial size of the connection pool, for this specific connection string. Default is `0`.

pooling

Turn on connection pooling. Default is `true`.

5.3.3 Other Connection Parameters

The following optional parameters can be set if required.

application name

Sets the application name.

connection timeout

alternate name: CONNECT TIMEOUT

Sets the length of time in seconds to try and establish a connection before failure. Default is `30`. This setting will be ignored if it is greater than the default maximum defined by your OS. The OS-specific value is usually around 20 to 40 seconds.

current language

Sets the language for this process.

logfile

Turns on logging and sets the log file location.

packet size

Sets the TCP Packet size. Default is `1024`.

PREPARSE CACHE SIZE

Sets an upper limit to the number of SQL commands that will be held in the preparse cache before recycling is applied. Default is `200`.

sharedmemory

Enables or disables shared memory connections on `localhost` or `127.0.0.1`. For example: `SharedMemory=false` disables shared memory. Default is `true`.

so rcvbuf

Sets the TCP receive buffer size. Default is `0` (use system default value).

so sndbuf

Sets the TCP send buffer size. Default is `0` (use system default value).

ssl

Specifies whether SSL/TLS secures the client-server connection (see Configuring .NET Clients to Use SSL/TLS with InterSystems IRIS). Default is `false`.

tcp nodelay

Sets the TCP *nodelay* option. Default is `true`.

transaction isolation level

Sets the System.Data.IsolationLevel value for the connection.

workstation id

Sets the Workstation name for process identification.