Getting Started with Entity Framework 6 Code First using MVC 5

Tom Dykstra, Rick Anderson

Step By Step, Guide

Microsoft

Connection Resiliency and Command Interception with the Entity Framework in an ASP.NET MVC Application

So far the application has been running locally in IIS Express on your development computer. To make a real application available for other people to use over the Internet, you have to deploy it to a web hosting provider, and you have to deploy the database to a database server.

In this tutorial you'll learn how to use two features of Entity Framework 6 that are especially valuable when you are deploying to the cloud environment: connection resiliency (automatic retries for transient errors) and command interception (catch all SQL queries sent to the database in order to log or change them).

This connection resiliency and command interception tutorial is optional. If you skip this tutorial, a few minor adjustments will have to be made in subsequent tutorials.

Enable connection resiliency

When you deploy the application to Windows Azure, you'll deploy the database to Windows Azure SQL Database, a cloud database service. Transient connection errors are typically more frequent when you connect to a cloud database service than when your web server and your database server are directly connected together in the same data center. Even if a cloud web server and a cloud database service are hosted in the same data center, there are more network connections between them that can have problems, such as load balancers.

Also a cloud service is typically shared by other users, which means its responsiveness can be affected by them. And your access to the database might be subject to throttling. Throttling means the database service throws exceptions when you try to access it more frequently than is allowed in your Service Level Agreement (SLA).

Many or most connection problems when you're accessing a cloud service are transient, that is, they resolve themselves in a short period of time. So when you try a database operation and get a type of error that is typically transient, you could try the operation again after a short wait, and the operation might be successful. You can provide a much better experience for your users if you handle transient errors by automatically trying again, making most of them invisible to the customer. The connection resiliency feature in Entity Framework 6 automates that process of retrying failed SQL queries.

The connection resiliency feature must be configured appropriately for a particular database service:

- It has to know which exceptions are likely to be transient. You want to retry errors caused by a temporary loss in network connectivity, not errors caused by program bugs, for example.
- It has to wait an appropriate amount of time between retries of a failed operation. You can wait longer between retries for a batch process than you can for an online web page where a user is waiting for a response.
- It has to retry an appropriate number of times before it gives up. You might want to retry more times in a batch process that you would in an online application.

You can configure these settings manually for any database environment supported by an Entity Framework provider, but default values that typically work well for an online application that uses Windows Azure SQL Database have already been configured for you, and those are the settings you'll implement for the Contoso University application.

All you have to do to enable connection resiliency is create a class in your assembly that derives from the DbConfiguration class, and in that class set the SQL Database *execution strategy*, which in EF is another term for *retry policy*.

1. In the DAL folder, add a class file named *SchoolConfiguration.cs*.
2. Replace the template code with the following code:

```
using System.Data.Entity;
using System.Data.Entity.SqlServer;

namespace ContosoUniversity.DAL
{
    public class SchoolConfiguration : DbConfiguration
    {
        public SchoolConfiguration()
        {
            SetExecutionStrategy("System.Data.SqlClient", () => new
SqlAzureExecutionStrategy());
        }
    }
}
```

   The Entity Framework automatically runs the code it finds in a class that derives from `DbConfiguration`. You can use the `DbConfiguration` class to do configuration tasks in code that you would otherwise do in the *Web.config* file. For more information, see EntityFramework Code-Based Configuration.

3. In *StudentController.cs*, add a `using` statement for `System.Data.Entity.Infrastructure`.

```
using System.Data.Entity.Infrastructure;
```

4. Change all of the `catch` blocks that catch `DataException` exceptions so that they catch `RetryLimitExceededException` exceptions instead. For example:

```
catch (RetryLimitExceededException /* dex */)
{
    //Log the error (uncomment dex variable name and add a line here to
write a log.
    ModelState.AddModelError("", "Unable to save changes. Try again,
and if the problem persists see your system administrator.");
}
```

You were using `DataException` to try to identify errors that might be transient in order to give a friendly "try again" message. But now that you've turned on a retry policy, the only errors likely to be transient will already have been tried and failed several times and the actual exception returned will be wrapped in the `RetryLimitExceededException` exception.

For more information, see Entity Framework Connection Resiliency / Retry Logic.

Enable Command Interception

Now that you've turned on a retry policy, how do you test to verify that it is working as expected? It's not so easy to force a transient error to happen, especially when you're running locally, and it would be especially difficult to integrate actual transient errors into an automated unit test. To test the connection resiliency feature, you need a way to intercept queries that Entity Framework sends to SQL Server and replace the SQL Server response with an exception type that is typically transient.

You can also use query interception in order to implement a best practice for cloud applications: log the latency and success or failure of all calls to external services such as database services. EF6 provides a dedicated logging API that can make it easier to do logging, but in this section of the tutorial you'll learn how to use the Entity Framework's interception feature directly, both for logging and for simulating transient errors.

Create a logging interface and class

A best practice for logging is to do it by using an interface rather than hard-coding calls to System.Diagnostics.Trace or a logging class. That makes it easier to change your logging mechanism later if you ever need to do that. So in this section you'll create the logging interface and a class to implement it./p>

1. Create a folder in the project and name it *Logging*.
2. In the *Logging* folder, create a class file named *ILogger.cs*, and replace the template code with the following code:

```
using System;

namespace ContosoUniversity.Logging
{
    public interface ILogger
    {
```

```
        void Information(string message);
        void Information(string fmt, params object[] vars);
        void Information(Exception exception, string fmt, params
object[] vars);

        void Warning(string message);
        void Warning(string fmt, params object[] vars);
        void Warning(Exception exception, string fmt, params object[]
vars);

        void Error(string message);
        void Error(string fmt, params object[] vars);
        void Error(Exception exception, string fmt, params object[]
vars);

        void TraceApi(string componentName, string method, TimeSpan
timespan);
        void TraceApi(string componentName, string method, TimeSpan
timespan, string properties);
        void TraceApi(string componentName, string method, TimeSpan
timespan, string fmt, params object[] vars);

    }
}
```

The interface provides three tracing levels to indicate the relative importance of logs, and one designed to provide latency information for external service calls such as database queries. The logging methods have overloads that let you pass in an exception. This is so that exception information including stack trace and inner exceptions is reliably logged by the class that implements the interface, instead of relying on that being done in each logging method call throughout the application.

The TraceApi methods enable you to track the latency of each call to an external service such as SQL Database.

3. In the *Logging* folder, create a class file named *Logger.cs*, and replace the template code with the following code:

```
using System;
using System.Diagnostics;
using System.Text;

namespace ContosoUniversity.Logging
{
    public class Logger : ILogger
    {

        public void Information(string message)
        {
            Trace.TraceInformation(message);
        }

        public void Information(string fmt, params object[] vars)
        {
```

```
            Trace.TraceInformation(fmt, vars);
        }

        public void Information(Exception exception, string fmt, params
object[] vars)
        {
            Trace.TraceInformation(FormatExceptionMessage(exception,
fmt, vars));
        }

        public void Warning(string message)
        {
            Trace.TraceWarning(message);
        }

        public void Warning(string fmt, params object[] vars)
        {
            Trace.TraceWarning(fmt, vars);
        }

        public void Warning(Exception exception, string fmt, params
object[] vars)
        {
            Trace.TraceWarning(FormatExceptionMessage(exception, fmt,
vars));
        }

        public void Error(string message)
        {
            Trace.TraceError(message);
        }

        public void Error(string fmt, params object[] vars)
        {
            Trace.TraceError(fmt, vars);
        }

        public void Error(Exception exception, string fmt, params
object[] vars)
        {
            Trace.TraceError(FormatExceptionMessage(exception, fmt,
vars));
        }

        public void TraceApi(string componentName, string method,
TimeSpan timespan)
        {
            TraceApi(componentName, method, timespan, "");
        }

        public void TraceApi(string componentName, string method,
TimeSpan timespan, string fmt, params object[] vars)
        {
            TraceApi(componentName, method, timespan,
string.Format(fmt, vars));
        }
```

```
        public void TraceApi(string componentName, string method,
TimeSpan timespan, string properties)
        {
            string message = String.Concat("Component:", componentName,
";Method:", method, ";Timespan:", timespan.ToString(), ";Properties:",
properties);
            Trace.TraceInformation(message);
        }

        private static string FormatExceptionMessage(Exception
exception, string fmt, object[] vars)
        {
            // Simple exception formatting: for a more comprehensive
version see
            // http://code.msdn.microsoft.com/windowsazure/Fix-It-app-
for-Building-cdd80df4
            var sb = new StringBuilder();
            sb.Append(string.Format(fmt, vars));
            sb.Append(" Exception: ");
            sb.Append(exception.ToString());
            return  sb.ToString();
        }
    }
}
```

The implementation uses System.Diagnostics to do the tracing. This is a built-in feature of .NET which makes it easy to generate and use tracing information. There are many "listeners" you can use with System.Diagnostics tracing, to write logs to files, for example, or to write them to blob storage in Windows Azure. See some of the options, and links to other resources for more information, in Troubleshooting Windows Azure Web Sites in Visual Studio. For this tutorial you'll only look at logs in the Visual Studio **Output** window.

In a production application you might want to consider tracing packages other than System.Diagnostics, and the ILogger interface makes it relatively easy to switch to a different tracing mechanism if you decide to do that.

Create interceptor classes

Next you'll create the classes that the Entity Framework will call into every time it is going to send a query to the database, one to simulate transient errors and one to do logging. These interceptor classes must derive from the `DbCommandInterceptor` class. In them you write method overrides that are automatically called when query is about to be executed. In these methods you can examine or log the query that is being sent to the database, and you can change the query before it's sent to the database or return something to Entity Framework yourself without even passing the query to the database.

1. To create the interceptor class that will log every SQL query that is sent to the database, create a class file named *SchoolInterceptorLogging.cs* in the *DAL* folder, and replace the template code with the following code:

```
using System;
using System.Data.Common;
using System.Data.Entity;
using System.Data.Entity.Infrastructure.Interception;
using System.Data.Entity.SqlServer;
using System.Data.SqlClient;
using System.Diagnostics;
using System.Reflection;
using System.Linq;
using ContosoUniversity.Logging;

namespace ContosoUniversity.DAL
{
    public class SchoolInterceptorLogging : DbCommandInterceptor
    {
        private ILogger _logger = new Logger();
        private readonly Stopwatch _stopwatch = new Stopwatch();

        public override void ScalarExecuting(DbCommand command,
DbCommandInterceptionContext<object> interceptionContext)
        {
            base.ScalarExecuting(command, interceptionContext);
            _stopwatch.Restart();
        }

        public override void ScalarExecuted(DbCommand command,
DbCommandInterceptionContext<object> interceptionContext)
        {
            _stopwatch.Stop();
            if (interceptionContext.Exception != null)
            {
                _logger.Error(interceptionContext.Exception, "Error
executing command: {0}", command.CommandText);
            }
            else
            {
                _logger.TraceApi("SQL Database",
"SchoolInterceptor.ScalarExecuted", _stopwatch.Elapsed, "Command: {0}:
", command.CommandText);
            }
            base.ScalarExecuted(command, interceptionContext);
        }

        public override void NonQueryExecuting(DbCommand command,
DbCommandInterceptionContext<int> interceptionContext)
        {
            base.NonQueryExecuting(command, interceptionContext);
            _stopwatch.Restart();
        }

        public override void NonQueryExecuted(DbCommand command,
DbCommandInterceptionContext<int> interceptionContext)
        {
            _stopwatch.Stop();
            if (interceptionContext.Exception != null)
            {
```

```
                _logger.Error(interceptionContext.Exception, "Error
executing command: {0}", command.CommandText);
            }
            else
            {
                _logger.TraceApi("SQL Database",
"SchoolInterceptor.NonQueryExecuted", _stopwatch.Elapsed, "Command:
{0}: ", command.CommandText);
            }
            base.NonQueryExecuted(command, interceptionContext);
        }

        public override void ReaderExecuting(DbCommand command,
DbCommandInterceptionContext<DbDataReader> interceptionContext)
        {
            base.ReaderExecuting(command, interceptionContext);
            _stopwatch.Restart();
        }
        public override void ReaderExecuted(DbCommand command,
DbCommandInterceptionContext<DbDataReader> interceptionContext)
        {
            _stopwatch.Stop();
            if (interceptionContext.Exception != null)
            {
                _logger.Error(interceptionContext.Exception, "Error
executing command: {0}", command.CommandText);
            }
            else
            {
                _logger.TraceApi("SQL Database",
"SchoolInterceptor.ReaderExecuted", _stopwatch.Elapsed, "Command: {0}:
", command.CommandText);
            }
            base.ReaderExecuted(command, interceptionContext);
        }
    }
}
```

For successful queries or commands, this code writes an Information log with latency information. For exceptions, it creates an Error log.

2. To create the interceptor class that will generate dummy transient errors when you enter "Throw" in the **Search** box, create a class file named *SchoolInterceptorTransientErrors.cs* in the *DAL* folder, and replace the template code with the following code:

```
using System;
using System.Data.Common;
using System.Data.Entity;
using System.Data.Entity.Infrastructure.Interception;
using System.Data.Entity.SqlServer;
using System.Data.SqlClient;
using System.Diagnostics;
using System.Reflection;
using System.Linq;
```

```
using ContosoUniversity.Logging;

namespace ContosoUniversity.DAL
{
    public class SchoolInterceptorTransientErrors :
DbCommandInterceptor
    {
        private int _counter = 0;
        private ILogger _logger = new Logger();

        public override void ReaderExecuting(DbCommand command,
DbCommandInterceptionContext<DbDataReader> interceptionContext)
        {
            bool throwTransientErrors = false;
            if (command.Parameters.Count > 0 &&
command.Parameters[0].Value.ToString() == "Throw")
            {
                throwTransientErrors = true;
                command.Parameters[0].Value = "an";
                command.Parameters[1].Value = "an";
            }

            if (throwTransientErrors && _counter < 4)
            {
                _logger.Information("Returning transient error for
command: {0}", command.CommandText);
                _counter++;
                interceptionContext.Exception =
CreateDummySqlException();
            }
        }

        private SqlException CreateDummySqlException()
        {
            // The instance of SQL Server you attempted to connect to
does not support encryption
            var sqlErrorNumber = 20;

            var sqlErrorCtor =
typeof(SqlError).GetConstructors(BindingFlags.Instance |
BindingFlags.NonPublic).Where(c => c.GetParameters().Count() ==
7).Single();
            var sqlError = sqlErrorCtor.Invoke(new object[] {
sqlErrorNumber, (byte)0, (byte)0, "", "", "", 1 });

            var errorCollection =
Activator.CreateInstance(typeof(SqlErrorCollection), true);
            var addMethod = typeof(SqlErrorCollection).GetMethod("Add",
BindingFlags.Instance | BindingFlags.NonPublic);
            addMethod.Invoke(errorCollection, new[] { sqlError });

            var sqlExceptionCtor =
typeof(SqlException).GetConstructors(BindingFlags.Instance |
BindingFlags.NonPublic).Where(c => c.GetParameters().Count() ==
4).Single();
```

```
            var sqlException =
(SqlException)sqlExceptionCtor.Invoke(new object[] { "Dummy",
errorCollection, null, Guid.NewGuid() });

            return sqlException;
        }
    }
}
```

This code only overrides the `ReaderExecuting` method, which is called for queries that can return multiple rows of data. If you wanted to check connection resiliency for other types of queries, you could also override the `NonQueryExecuting` and `ScalarExecuting` methods, as the logging interceptor does.

When you run the Student page and enter "Throw" as the search string, this code creates a dummy SQL Database exception for error number 20, a type known to be typically transient. Other error numbers currently recognized as transient are 64, 233, 10053, 10054, 10060, 10928, 10929, 40197, 40501, abd 40613, but these are subject to change in new versions of SQL Database.

The code returns the exception to Entity Framework instead of running the query and passing back query results. The transient exception is returned four times, and then the code reverts to the normal procedure of passing the query to the database.

Because everything is logged, you'll be able to see that Entity Framework tries to execute the query four times before finally succeeding, and the only difference in the application is that it takes longer to render a page with query results.

The number of times the Entity Framework will retry is configurable; the code specifies four times because that's the default value for the SQL Database execution policy. If you change the execution policy, you'd also change the code here that specifies how many times transient errors are generated. You could also change the code to generate more exceptions so that Entity Framework will throw the `RetryLimitExceededException` exception.

The value you enter in the Search box will be in `command.Parameters[0]` and `command.Parameters[1]` (one is used for the first name and one for the last name). When the value "Throw" is found, it is replaced in those parameters by "an" so that some students will be found and returned.

This is just a convenient way to test connection resiliency based on changing some input to the application UI. You can also write code that generates transient errors for all queries or updates, as explained later in the comments about the *DbInterception.Add* method.

3. In *Global.asax*, add the following `using` statements:

```
using ContosoUniversity.DAL;
```

```
using System.Data.Entity.Infrastructure.Interception;
```

4. Add the highlighted line to the `Application_Start` method:

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);
    DbInterception.Add(new SchoolInterceptorTransientErrors());
    DbInterception.Add(new SchoolInterceptorLogging());
}
```

These lines of code are what causes your interceptor code to be run when Entity Framework sends queries to the database. Notice that because you created separate interceptor classes for transient error simulation and logging, you can independently enable and disable them.

You can add interceptors using the `DbInterception.Add` method anywhere in your code; it doesn't have to be in the `Application_Start` method. Another option is to put this code in the DbConfiguration class that you created earlier to configure the execution policy.

```
public class SchoolConfiguration : DbConfiguration
{
    public SchoolConfiguration()
    {
        SetExecutionStrategy("System.Data.SqlClient", () => new
SqlAzureExecutionStrategy());
        DbInterception.Add(new SchoolInterceptorTransientErrors());
        DbInterception.Add(new SchoolInterceptorLogging());
    }
}
```

Wherever you put this code, be careful not to execute `DbInterception.Add` for the same interceptor more than once, or you'll get additional interceptor instances. For example, if you add the logging interceptor twice, you'll see two logs for every SQL query.

Interceptors are executed in the order of registration (the order in which the `DbInterception.Add` method is called). The order might matter depending on what you're doing in the interceptor. For example, an interceptor might change the SQL command that it gets in the `CommandText` property. If it does change the SQL command, the next interceptor will get the changed SQL command, not the original SQL command.

You've written the transient error simulation code in a way that lets you cause transient errors by entering a different value in the UI. As an alternative, you could write the interceptor code to always generate the sequence of transient exceptions without checking for a particular parameter value. You could then add the interceptor only when you want to generate transient errors. If you do this, however, don't add the interceptor

until after database initialization has completed. In other words, do at least one database operation such as a query on one of your entity sets before you start generating transient errors. The Entity Framework executes several queries during database initialization, and they aren't executed in a transaction, so errors during initialization could cause the context to get into an inconsistent state.

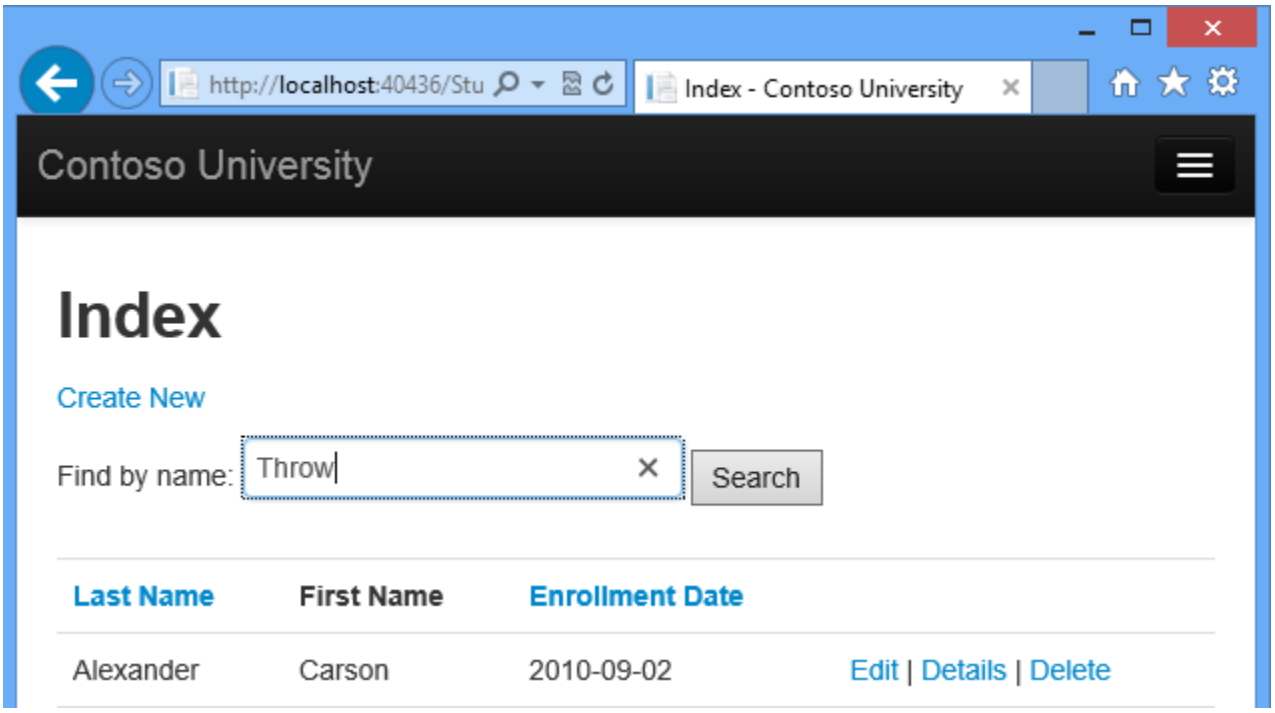
Test logging and connection resiliency

1. Press F5 to run the application in debug mode, and then click the **Students** tab.
2. Look at the Visual Studio **Output** window to see the tracing output. You might have to scroll up past some JavaScript errors to get to the logs written by your logger.

   Notice that you can see the actual SQL queries sent to the database. You see some initial queries and commands that Entity Framework does to get started, checking the database version and migration history table (you'll learn about migrations in the next tutorial). And you see a query for paging, to find out how many students there are, and finally you see the query that gets the student data.

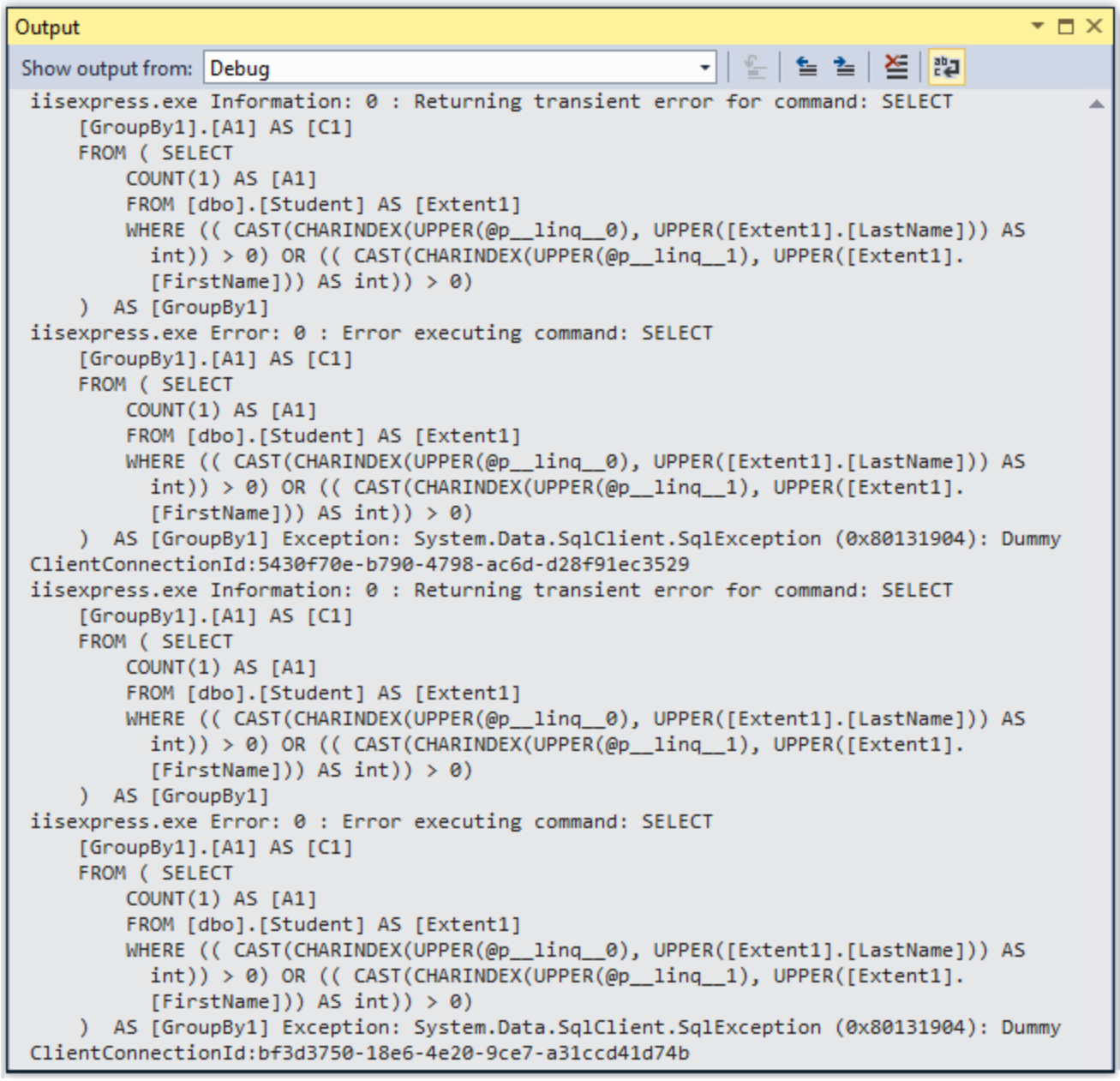

Output
Show output from: Debug
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ReaderExecuted;Timespan:00:00:00.0276705;Properties
:Command: select serverproperty('EngineEdition'):
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ScalarExecuted;Timespan:00:00:00.0424968;Properties
:Command: SELECT Count(*) FROM sys.databases WHERE [name]=N'ContosoUniversity2':
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ScalarExecuted;Timespan:00:00:00.0450531;Properties
:Command:
SELECT Count(*)
FROM INFORMATION_SCHEMA.TABLES AS t
WHERE t.TABLE_TYPE = 'BASE TABLE'
AND (t.TABLE_SCHEMA + '.' + t.TABLE_NAME IN
('dbo.Course','dbo.Department','dbo.Instructor','dbo.OfficeAssignment','dbo.Enrol
lment','dbo.Student','dbo.CourseInstructor')
OR t.TABLE_NAME = 'EdmMetadata'):
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ScalarExecuted;Timespan:00:00:00.0262633;Properties
:Command: SELECT Count(*) FROM sys.databases WHERE [name]=N'ContosoUniversity2':
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ReaderExecuted;Timespan:00:00:00.0373571;Properties
:Command: SELECT
[GroupBy1].[A1] AS [C1]
FROM (SELECT
COUNT(1) AS [A1]
FROM [dbo].[__MigrationHistory] AS [Extent1]
) AS [GroupBy1]:
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ReaderExecuted;Timespan:00:00:00.0227459;Properties
:Command: SELECT
[GroupBy1].[A1] AS [C1]
FROM (SELECT
COUNT(1) AS [A1]
FROM [dbo].[__MigrationHistory] AS [Extent1]
WHERE ([Extent1].[ContextKey] = @p__linq__0) AND (@p__linq__0 IS NOT NULL)
) AS [GroupBy1]:
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ReaderExecuted;Timespan:00:00:00.0298657;Properties
:Command: SELECT TOP (1)
[Project1].[C1] AS [C1],
[Project1].[MigrationId] AS [MigrationId],
[Project1].[Model] AS [Model]
FROM (SELECT
[Extent1].[MigrationId] AS [MigrationId],
[Extent1].[Model] AS [Model],
1 AS [C1]
FROM [dbo].[__MigrationHistory] AS [Extent1]
WHERE ([Extent1].[ContextKey] = @p__linq__0) AND (@p__linq__0 IS NOT NULL)
) AS [Project1]
ORDER BY [Project1].[MigrationId] DESC:
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ReaderExecuted;Timespan:00:00:00.0246607;Properties
:Command: SELECT
[GroupBy1].[A1] AS [C1]
FROM (SELECT
COUNT(1) AS [A1]
FROM [dbo].[Student] AS [Extent1]
) AS [GroupBy1]:
'iisexpress.exe' (CLR v4.0.30319: /LM/W3SVC/12/ROOT-1-130329870303366864): Loaded
'EntityFrameworkDynamicProxies-ContosoUniversity'.

3. In the **Students** page, enter "Throw" as the search string, and click **Search**.



You'll notice that the browser seems to hang for several seconds while Entity Framework is retrying the query several times. The first retry happens very quickly, then the wait before increases before each additional retry. This process of waiting longer before each retry is called *exponential backoff*.

When the page displays, showing students who have "an" in their names, look at the output window, and you'll see that the same query was attempted five times, the first four times returning transient exceptions. For each transient error you'll see the log that you write when generating the transient error in the `SchoolInterceptorTransientErrors` class ("Returning transient error for command...") and you'll see the log written when `SchoolInterceptorLogging` gets the exception.



Since you entered a search string, the query that returns student data is parameterized:

```
SELECT TOP (3)
    [Project1].[ID] AS [ID],
    [Project1].[LastName] AS [LastName],
    [Project1].[FirstMidName] AS [FirstMidName],
    [Project1].[EnrollmentDate] AS [EnrollmentDate]
    FROM ( SELECT [Project1].[ID] AS [ID], [Project1].[LastName] AS
[LastName], [Project1].[FirstMidName] AS [FirstMidName],
[Project1].[EnrollmentDate] AS [EnrollmentDate], row_number() OVER
(ORDER BY [Project1].[LastName] ASC) AS [row_number]
        FROM ( SELECT
            [Extent1].[ID] AS [ID],
```

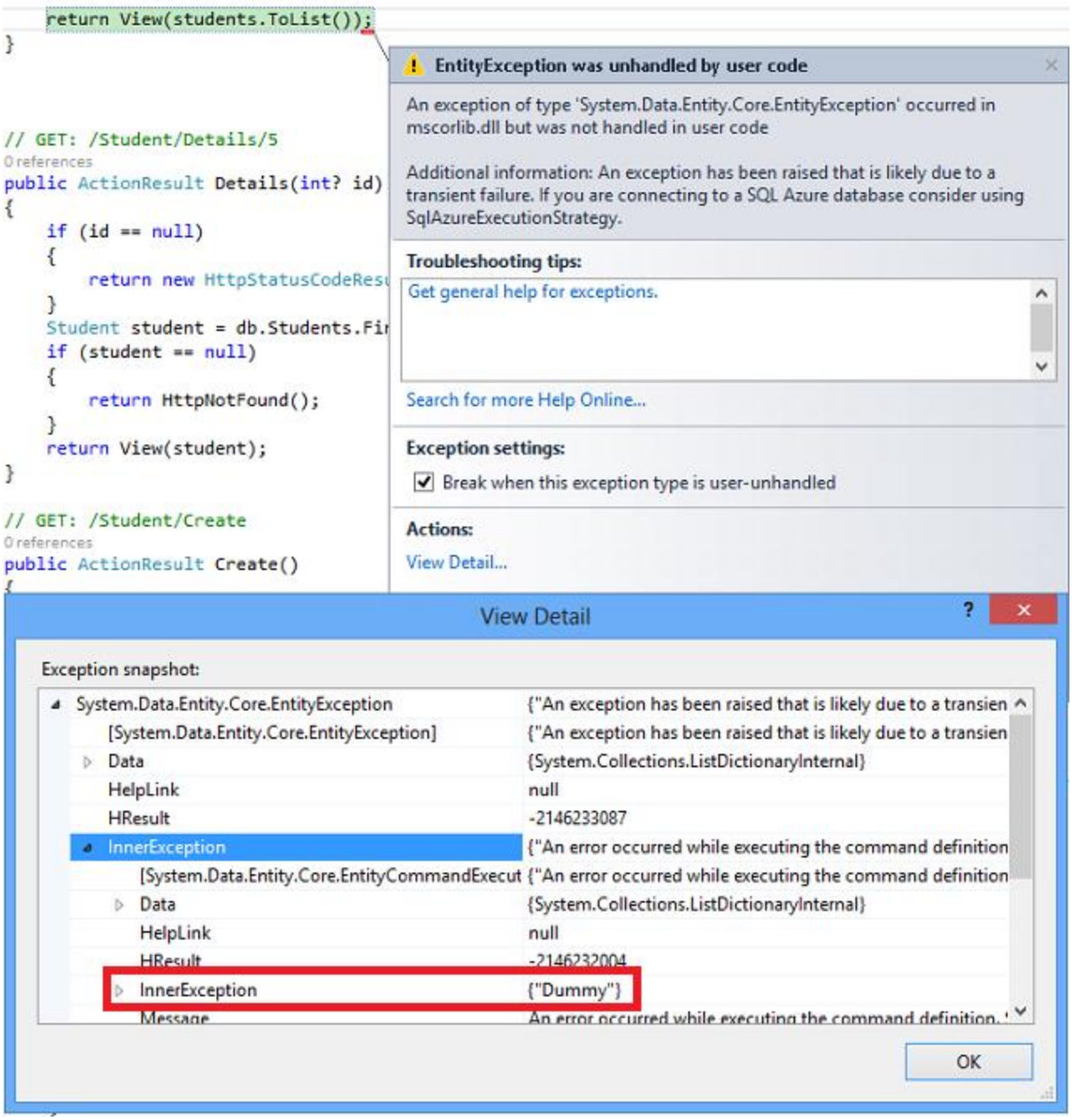
```
            [Extent1].[LastName] AS [LastName],
            [Extent1].[FirstMidName] AS [FirstMidName],
            [Extent1].[EnrollmentDate] AS [EnrollmentDate]
            FROM [dbo].[Student] AS [Extent1]
            WHERE (( CAST(CHARINDEX(UPPER(@p__linq__0),
UPPER([Extent1].[LastName])) AS int)) > 0) OR ((
CAST(CHARINDEX(UPPER(@p__linq__1), UPPER([Extent1].[FirstMidName])) AS
int)) > 0)
        )  AS [Project1]
    )  AS [Project1]
    WHERE [Project1].[row_number] > 0
    ORDER BY [Project1].[LastName] ASC
```

You're not logging the value of the parameters, but you could do that. If you want to see the parameter values, you can write logging code to get parameter values from the `Parameters` property of the `DbCommand` object that you get in the interceptor methods.

Note that you can't repeat this test unless you stop the application and restart it. If you wanted to be able to test connection resiliency multiple times in a single run of the application, you could write code to reset the error counter in `SchoolInterceptorTransientErrors`.

4. To see the difference the execution strategy (retry policy) makes, comment out the `SetExecutionStrategy` line in *SchoolConfiguration.cs*, run the Students page in debug mode again, and search for "Throw" again.

   This time the debugger stops on the first generated exception immediately when it tries to execute the query the first time.



5. Uncomment the *SetExecutionStrategy* line in *SchoolConfiguration.cs*.

Summary

In this tutorial you've seen how to enable connection resiliency and log SQL commands that Entity Framework composes and sends to the database. In the next tutorial you'll deploy the application to the Internet, using Code First Migrations to deploy the database.