ADO

Objective

- Introduction to ADO.NET
- Working with Connected Architecture
- Working with Disconnected Architecture

TIME FOR CASE STUDY

Case Study (continued)

- As a Shopon customer, I want to view all the products which are available in the store.

Thought

- We already have data in the database, we have to connect to the database and retrieve the values from the database.

- To retrieve data from database, we can use ADO.NET.

Knowledge Byte

Solution

Introduction to ADO.NET

- **ADO.NET** is the new database technology of the .NET platform, and it builds on **Microsoft ActiveX Data Objects** (ADO).
- It is an integral part of the .NET Compact Framework, providing access to relational data, XML documents, and application data
- Provides data access services in the Microsoft .NET platform
- An object-oriented set of libraries that allows you to interact with data sources
- Commonly, the data source is a data base, but it could also be a text file, an Excel spread sheet, or an XML file

Command

- A command is, in its simplest form, a string of text containing SQL statements that is to be issued to the database

- A command could also be a stored procedure, or the name of a table that will return all columns and all rows from that table

- A command can be constructed by passing the SQL clause as a parameter to the constructor of the Command class

```
string Source = “Data Source=server-test\sqlexpress; Integrated security=SSPI;
Initial Catalog=SalesOrder”;
string Query = “SELECT CustomerId, CustName FROM Customers”;
SqlConnection Conn = new SqlConnection(source);
Conn.Open();
SqlCommand Cmd = new SqlCommand(Query, Conn);
```

Command

- The Command class has a property called CommandType , which is used to define whether the command is a SQL clause, a call to a stored procedure, or a full table statement (which simply selects all columns and rows from a given table)

Command Type	Example
Text (default)	String Query = “SELECT CustName FROM Customers”; SqlCommand Cmd = new SqlCommand(Query , conn); cmd.CommandType = ***CommandType.Text; [Optional]***
Stored Procedure	SqlCommand cmd = new SqlCommand(“CustOrderHist”, conn); cmd.CommandType = ***CommandType.StoredProcedure;***
TableDirect ***[Only for OleDb provider]***	OleDbCommand cmd = new OleDbCommand(“Categories”, conn); cmd.CommandType = **CommandType.TableDirect;**

Command

- Once a command is defined, we need to execute the command
- Different ways exist to issue the statement, depending on what you expect to be returned (if anything) from that command
- Command classes provide the following execute methods
  - **ExecuteNonQuery()** — Executes the command but does not return any output
  - **ExecuteReader()** — Executes the command and returns a typed IDataReader
  - **ExecuteScalar()** — Executes the command and returns a single value
  - **ExecuteXmlReader()** — Executes the command and returns an XmlReader object, which can be used to traverse the XML fragment returned from the database ***[Only SqlCommand Class]***

SqlCommand

- **ExecuteNonQuery()**
  - This method is commonly used for INSERT, UPDATE, DELETE statements
  - The only returned value is the number of records affected as an int

```
string source = “Data Source=server-test\sqlexpress; Integrated Security=SSPI; Initial
Catalog=SalesOrder”;
string Query = “UPDATE Customers SET CustName = ‘KK’ WHERE CustomerId = 776”;
SqlConnection conn = new SqlConnection(source);
conn.Open();
SqlCommand cmd = new SqlCommand(Query, conn);
int rowsReturned = cmd.ExecuteNonQuery();
Console.WriteLine(“{0} rows returned.”, rowsReturned);
conn.Close();
```

Please refer CommandDemo.cs – ExecuteNonQueryDemo()

SqlCommand (continued)

- **ExecuteScalar()**
  - On many occasions, it is necessary to return a single result from a SQL statement, such as the count of records in a given table, or the current date/time on the server
  - Returns the first value from the select statement. Return type is object, hence we need to cast based on the data type of the value

```
string source = “Data Source=server-test\sqlexpress; Integrated Security=SSPI; Initial
Catalog=SalesOrder”;
string Query = “SELECT COUNT(*) FROM Customers”;
SqlConnection conn = new SqlConnection(source);
conn.Open();
SqlCommand cmd = new SqlCommand(Query, conn);
int CustomerCount = (int) cmd.ExecuteScalar();
Console.WriteLine(“Total Customers : {0}”, CustomerCount);
conn.Close();
```

Please refer CommandDemo.cs – ExecuteScalarDemo()

SqlCommand (continued)

- **ExecuteReader()**
  - This method executes the command and returns a typed data reader object, depending on the provider in use.
  - The object returned can be used to iterate through the record(s) returned

```
string source = “Data Source=server-test\sqlexpress; Integrated Security=SSPI; Initial
Catalog=SalesOrder”;
string Query=“SELECT CustomerId,CustName FROM Customers”;
SqlConnection conn = new SqlConnection(source);
conn.Open();
SqlCommand cmd = new SqlCommand(Query, conn);
IDataReader reader = cmd.ExecuteReader();
while(reader.Read()) {
        Console.WriteLine(“ID : {0} Name : {1}”, reader[0] , reader[1]);
}
reader.Close();
        conn.Close();
```

Please refer CommandDemo.cs – ExecuteReaderDemo()

SqlCommand (continued)

- **ExecuteXmlReader()** ***(SqlClient Provider Only)***
  - This method executes the command and returns an XmlReader object to the caller
  - SQL Server permits a SQL SELECT statement to be extended with a FOR XML clause
  - This clause can take one of three options:
    - FOR XML AUTO — Builds a tree based on the tables in the FROM clause
    - FOR XML RAW — Maps result set rows to elements, with columns mapped to attributes
    - FOR XML EXPLICIT – Requires that you specify the shape of the XML tree to be returned

SqlCommand (continued)

- **ExecuteXmlReader()** ***(SqlClient Provider Only)***

```
string source = “Data Source=server-test\sqlexpress; Integrated Security=SSPI; Initial Catalog=SalesOrder”;
string Query=“SELECT CustomerId, CustName FROM Customers FOR XML AUTO”;
SqlConnection conn = new SqlConnection(source);
conn.Open();
SqlCommand cmd = new SqlCommand(Query, conn);
XmlReader xr = cmd.ExecuteXmlReader();
xr.Read();
string data;
Do
{
data = xr.ReadOuterXml();
if (!string.IsNullOrEmpty(data))
                Console.WriteLine(data);
} while (!string.IsNullOrEmpty(data));
xr.Close(); conn.Close();
```

[Please refer CommandDemo.cs - ExecuteXmlReaderDemo()]

Reader

- A data reader is the simplest and fastest way of selecting some data from a data source
- It can only read data and cannot write
- You cannot directly instantiate a data reader object
- An instance is returned from the appropriate database ’ s command object (such as SqlCommand) after having called the ExecuteReader() method
- DataReaders are often described as fast-forward firehose-like streams of data

Reader

- Once you've read some data, you must save it because you will not be able to go back and read it again
- The forward only design of the DataReader is what enables it to be fast
- It doesn't have overhead associated with traversing the data or writing it back to the data source
- Therefore, if your only requirement for a group of data is for reading one time and you want the fastest method possible, the DataReader is the best choice
- The typical method of reading from the data stream returned by the SqlDataReader is to iterate through each row with a while loop

Reader

- The database connection used is kept open until the data reader has been closed
- The DataReader class has an indexer that permits access to any
  field using the familiar array style syntax
  object o = aReader[0]; or object o = aReader[“CustomerId”];
  Accessing through indexer is going to be faster however the latter is more readable
- Regardless of the type of the indexer parameter, a DataReader indexer will return type object
- We can convert to anything after that and proceed with our requirement
- Reader has to be closed at the end

Reader

- The database connection used is kept open until the data reader has been closed

```
string source = “Data Source=server-test\sqlexpress; Integrated Security=SSPI; Initial
Catalog=SalesOrder”;
string Query=“SELECT CustomerId, CustName FROM Customers”;
SqlConnection conn = new SqlConnection(source);
conn.Open();
SqlCommand cmd = new SqlCommand(Query, conn);
IDataReader reader = cmd.ExecuteReader();
while(reader.Read())
{
Console.WriteLine(“ID : {0} Name : {1}”, reader[0] , reader[1]); or
 Console.WriteLine(“ID : {0} Name : {1}”, reader[“CustomerId”] , reader[“CustomerName”]);
}
reader.Close();
  conn.Close();
```

Reader

- The same code can be written as below mentioned

```
while(reader.Read())
{
    Console.WriteLine("ID : " + reader.GetValue(0));
    Console.WriteLine(" : Name" + reader.GetValue(1));

}
reader.Close();
     conn.Close();
```

- *Please refer ReaderDemo.cs – ExecuteReaderDemo1()*
- *Please refer ReaderDemo.cs – ExecuteReaderDemo2()*

Solution

```
public class ProductRepoDBImpl
{
    private string connectionString = "Data Source=(local);Initial Catalog=db_Shopon;Integrated Security=True";
    1 reference
    public IEnumerable<Product> GetProducts()
    {
        List<Product> products = new List<Product>();
        string connectionString = "Data Source=(local);Initial Catalog=db_Shopon;Integrated Security=True";
        string sqlSt = "SELECT pid as ProductId, productname, price, companyid, categoryid, availablestatus, " +
                       "imageUrl FROM Product WHERE isDeleted = 0";
        try
        {
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();
                SqlCommand command = new SqlCommand(sqlSt, connection);
                SqlDataReader reader = command.ExecuteReader();

                while (reader.Read())
                {
                    Product product = new Product();
                    product.ProductId = Convert.ToInt32(reader["ProductId"]);
                    product.ProductName = reader["productname"].ToString();
                    product.Price = Convert.ToDouble(reader["price"]);
                    product.ImgUrl = reader["imageUrl"].ToString();
                    product.CompanyId = Convert.ToInt32(reader["companyid"]);
                    product.CategoryId = Convert.ToInt32(reader["categoryid"]);
                    product.Availability = reader["availablestatus"].ToString();
                        products.Add(product);
                    }
                }
            }
            catch (Exception)
            {
                throw;
            }

            return products;
        }
}
```

Next Step

Case Study (continued)

- As a Shopon developer, I want get product details from stored procedure.

Thought

- We must create stored procedure if it does not exists.
- Set command type to System.Data.CommandType.StoredProcedure to fetch data from stored procedure.

Knowledge Byte | Solution

Invoking Stored Procedure

- A stored procedure is a pre-defined, reusable routine that is stored in a database
- SQL Server compiles stored procedures, which makes them more efficient to use
- Therefore, rather than dynamically building queries in your code, you can take advantage of the reuse and performance benefits of stored procedures
- Calling a stored procedure with a command object is just a matter of
  - defining the name of the stored procedure
  - adding a definition for each parameter of the procedure [If any]
  - setting the command type to *StoredProcedure*
  - then executing the command

Invoking Stored Procedure (continued)

- A procedure that doesn't accept any parameters

```
string source = "Data Source=server-test\sqlexpress; Integrated
Security=SSPI; Initial Catalog=SalesOrder";
SqlConnection conn = new SqlConnection(source);
conn.Open();
SqlCommand cmd = new SqlCommand("GetCustomers", conn);
cmd.CommandType = CommandType.StoredProcedure;
IDataReader reader = cmd.ExecuteReader();
```

"GetCustomers" is the name of the procedure, which returns the list of customers in the database

CommandType is set to StoredProcedure before executing the command

Refer StoredPrecedureDemo.cs – ExecuteSPWithoutParametersDemo()

Invoking Stored Procedure (continued)

- A procedure that accepts parameters but doesn't return any data

(Conn object as defined in the previous slide)

```
SqlCommand cmd = new SqlCommand("UpdateCustomer", conn);
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.AddWithValue ("@CustomerID", 775 );
cmd.Parameters.AddWithValue ("@CustomerName", "AAI" );
cmd.Parameters.AddWithValue ("@RegionCode", "DE" );
cmd.ExecuteNonQuery();
```

"UpdateCustomer" is the name of the procedure, which updates the name based on the customer id

CommandType is set to StoredProcedure before executing the command

Parameters.AddWithValue sets its value — you can also manually construct SqlParameter instances and add these to the Parameters collection if appropriate

Refer StoredPrecedureDemo.cs – ExecuteSPWithParametersDemo1()

Invoking Stored Procedure (continued)

- A procedure that accepts parameters but doesn't return any data. [Through SqlParameter instance]

(Conn object as defined in the previous slide)

```
SqlCommand cmd = new SqlCommand("DeleteCustomer", conn);
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.Add(new SqlParameter("@CustomerID",
                SqlDbType.Int , 0 ));
cmd.Parameters["@CustomerID"].Value= 775;
cmd.ExecuteNonQuery();
```

"DeleteCustomer" is the name of the procedure, which deletes the row based on the customer id

CommandType is set to StoredProcedure before executing the command

Parameters.Add constructs SqlParameter instance and sets the value here. may lead to better performance than re – constructing the entire SqlCommand for each call, when repeated calls have to be made

Refer StoredPrecedureDemo.cs – ExecuteSPWithParametersDemo2()

Invoking Stored Procedure (continued)

- A procedure that returns output parameter

(Conn object as defined in the previous slide)

```
SqlCommand cmd = new SqlCommand("InsertCustomer", conn);
cmd.CommandType = CommandType.StoredProcedure;

cmd.Parameters.AddWithValue("@CustName", "SKK");
cmd.Parameters.AddWithValue("@RegionCode", "BA");
SqlParameter CustomerId = new SqlParameter("@CustomerId",
                                           SqlDbType.Int, 0);

CustomerId.Direction = ParameterDirection.Output;
cmd.Parameters.Add(CustomerId);
cmd.UpdatedRowSource = UpdateRowSource.OutputParameters;

cmd.ExecuteNonQuery();

 int NewCustID = (int) cmd.Parameters["@CustomerID"].Value;
```

Invoking Stored Procedure (continued)

- A procedure that returns output parameter

"InsertCustomer" is the name of the procedure, which inserts a row based on the customer name given

CommandType is set to StoredProcedure before executing the command

ParametersWithValue are added to the command

SqlParameter instance is created and we are defining as an output parameter which would store the value returned from the procedure

The UpdateRowSource enumeration is used to indicate that data will be returned from this stored procedure via output parameters

After executing the command, the value of @CustomerID parameter is read and cast to an integer

Note: We can also obtain the value through return value option, from a stored procedure

Refer StoredPrecedureDemo.cs – ExecuteSPWithParametersDemo3()

Solution – Stored Procedure

```
USE [db_Shopon]
GO

SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE PROCEDURE [dbo].[sp_GetProductDetails]
AS
BEGIN
    SELECT  c.CompanyId, c.CompanyName, p.Pid, p.ProductName,
            p.Price, p.ImageUrl, ca.CategoryId, ca.Category
        FROM Company c
        INNER JOIN
            Product p ON c.companyid = p.companyid
        INNER JOIN
            category ca ON p.categoryid = ca.categoryid
    WHERE
        p.isDeleted = 0
END

GO
```

Solution – ADO.NET Code

```
public IEnumerable<Product> GetProducts()
{
    List<Product> products = new List<Product>();
    string connectionString = "Data Source=(local);Initial Catalog=db_Shopon;Integrated Security=True";
    //string sqlSt = "SELECT pid as ProductId, productname, price, companyid, categoryid, availablestatus,
    //                imageUrl FROM Product WHERE isDeleted = 0";
    string sqlSt = "sp_GetProductDetails";
    try
    {
        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            connection.Open();
            SqlCommand command = new SqlCommand(sqlSt, connection);
            command.CommandType = System.Data.CommandType.StoredProcedure;
            SqlDataReader reader = command.ExecuteReader();
            while (reader.Read())
            {
                Product product = new Product();
                product.ProductId = Convert.ToInt32(reader["PId"]);
                product.ProductName = reader["productname"].ToString();
                product.Price = Convert.ToDouble(reader["price"]);
                product.ImageUrl = reader["imageUrl"].ToString();
                product.CompanyId = Convert.ToInt32(reader["companyid"]);
                product.CategoryId = Convert.ToInt32(reader["categoryid"]);
                product.AvailabilityStatus = reader["availablestatus"].ToString();
                product.Company = companyRepo.GetCompanyById(product.CompanyId);
                products.Add(product);
            }
        }
    }
    catch (Exception)
    {
        throw;
    }
    return products;
}
```

Transactions

- A transaction is a set of operations that must either succeed or fail as a unit
- Goal of a transaction is to ensure that data is always in a valid and consistent state
- For Ex: Consider a transaction that transfers funds from one account A to another account B.
  - Should deduct amount from Account A
  - Should add amount to Account B
- If the application completes step 1 successfully and while executing step2 suppose there occurs an error, then there would exist inconsistent data
- Transactions help avoid these problems by ensuring that changes are committed only if all steps are successful

Version 1.0

Transactions

- So a system stays in one of two valid states, initial state with no money transferred and the final state with money debited from one account and credited into another account
- Transactions are characterized by four properties called ACID
- Atomic:
  - All steps in the transaction should succeed or fail together. Unless all the steps of a transaction is completed, transaction is not considered complete
- Consistent
  - The transaction takes the underlying database from one stable state to another
- Isolated
  - Every transaction is an independent entity. One transaction should not affect any other transaction running at the same time
- Durable
  - Changes that occur during the transaction are permanently stored on some media, typically a hard disk, before the transaction is declared successful. Logs are maintained to restore database if a hard[illegible] or network failure occurs

Transactions

- Most ADO.NET providers include support for database transactions
- Transactions are started through the connection object by calling the BeginTransaction() method
- This method returns a provider specific Transaction object, that would manage the transaction
  - All Transaction classes implement the IDbTransaction interface.
    For Ex: SqlTransaction : IDbTransaction
- Transaction class provides two key methods
  - Commit() – Identifies that the transaction is complete and that pending changes should be stored permanently
  - RollBack() – Indicates that the transaction was unsuccessful. Pending changes are discarded and the database state remains unchanged

Transactions

```
(Conn object as defined in the previous slides)
SqlTransaction NewTransaction = null;
try
{
        NewTransaction = Conn.BeginTransaction();
        string Sql1= “Insert into Customer (CustName, RegionCode)        values (‘SKK’, ‘BA’)”;
        string Sql2= “Insert into Customer (CustName, RegionCode)        values (‘SKK’, ‘BA’)”;
        SqlCommand cmd1 = new SqlCommand(Sql1, Conn);
        SqlCommand cmd2 = new SqlCommand(Sql2, Conn);
        cmd1.Transaction = NewTransaction;
        cmd2.Transaction = NewTransaction;
        cmd1.ExecuteNonQuery();
        cmd2.ExecuteNonQuery();
        NewTransaction.Commit();
}
catch(SqlException Ex) {
        NewTransaction.RollBack();
}
```

Transactions

- Executing transaction in the previous example
  - Declare Transaction object
  - Begin the transaction in the try block
  - Assign the transaction reference to the commands
  - Commit the transaction
  - If exception, then rollback

- *Refer TransactionDemo.cs – ExecuteTransactionDemo1()*

Transactions

- When implementing a transaction, you can follow these practices to achieve the best results
  - Keep transactions as short as possible
  - Avoid returning data with a select query in the middle of a transaction. Ideally you should return the data before a transaction starts. This reduces the amount of data your transaction will lock
  - If you do retrieve records, fetch only the rows that are required so as to reduce the number of locks
  - Whenever possible write transactions within stored procedures instead of using ADO.NET transactions. It is faster as your application need not talk to the database server
  - Avoid transactions that combine multiple independent batches of work. Put separate batches into separate transactions
  - Avoid updates to large range of records if possible

Transactions

- **Savepoint**
  - When roll back happens in a transaction, all the commands in the transaction are nullified
  - If we want to nullify only certain commands among all of those, then we have an option called *Savepoint*
  - You can mark a certain point in the flow of transaction and roll back to that point
  - Savepoint is set using Transaction.Save() method
  - Save() method is available only for SqlTransaction class
  - It is not part of the IDbTransaction interface

Refer TransactionDemo.cs – ExecuteTransactionDemo2()

Transactions

- **Transactions** don’t necessarily meet the strictest definition of Isolated
- This is because the isolation level of a transaction is configurable by you, when you create the transaction
- Exclusively locking a set of rows while a transaction works on them may be unfeasible due to performance
- .NET (through SQL Server), provides the ability to specify isolation levels when you create a transaction
- Simply requires supplying a System.Data.IsolationLevel value to the BeginTransaction() method
- There are found isolation levels that can be used

Transactions

Isolation Level	Description
ReadCommitted	The default for SQL Server. This level ensures that data written by one transaction will be accessible in a second transaction only after the first transaction is committed
ReadUncommitted	This is, essentially, no isolation. Anyone can read the data placed in a table or updated immediately after the SQL statement causes the change no commit is required. This could lead to a process having out-of-date data: it may be using a version of the data that was then rolled back out of the table
RepeatableRead	In this case, a shared lock is applied on all data queried within a transaction. This means that no other transaction can alter the data used in your transaction. This prevents the case where data you had queried once changes on subsequent queries. It does not, though, prevent the case where rows are added to the table that may be returned in subsequent queries
Serializable	This is the most “exclusive” transaction level, which in effect serializes access to data within the database. With this isolation level, phantom rows can never show up, so a SQL statement issued within a serializable transaction will always retrieve the same data. The negative performance impact of a Serializable transaction should not be underestimated — if you don’t absolutely need to use this level of isolation, stay away from it.

DataSet

- A DataSet is an in-memory data store that can hold numerous tables
- It is the heart of your disconnected data model
- DataSets only hold data and do not interact with a data source
- The DataSet class has been designed as an offline container of data. It has no notion of database connections
- The DataSet object is made up of two objects:
  - DataTableCollection object containing null or multiple DataTable objects (Columns, Rows, Constraints).
  - DataRelationCollection object containing null or multiple DataRelation objects which establish a parent/child relation between two DataTable objects

DataSet

- You just create a new instance, just like any other object
- DataSet NewDS = new DataSet();
- A DataTable is very similar to your physical database table
  - It consists of several columns with particular properties
  - Might have zero or more rows of data
  - Zero or more constriants
- DataTable consists of
  - DataRowCollection which is a collection of DataRow instances
  - DataColumnCollection which is a collection of DataColumn instances
  - The Constraints collection can be populated with either unique or primary key constraints

DataSet

- A DataColumn object defines properties of a column within the DataTable , such as the data type of that column, whether the column is read - only, and various other facts
- A column can be created in code, or it can be automatically generated by the runtime
- The data type of the column can be set either by supplying it in the constructor or by setting the DataType property
- Data columns can be created to hold the following .NET Framework data types
  - Boolean, Decimal, Int64, TimeSpan, Byte, Double, Sbyte, UInt16, Char, Int16, Single, UInt32, DateTime,Int32,String , UInt64

DataSet

- The following table shows the properties that can be set on a DataColumn object

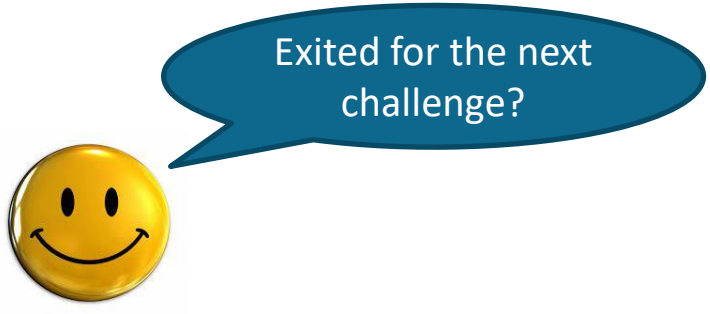
Property	Description
AllowDBNull	If true, permits the column to be set to DBNull
AutoIncrement	Defines that this column value is automatically generated as an incrementing number
AutoIncrementSeed	Defines the initial seed value for an AutoIncrement column
AutoIncrementStep	Defines the step between automatically generated column values, with a default of one
Caption	Can be used for displaying the name of the column onscreen
ColumnMapping	Defines how a column is mapped into XML when a DataSet class is saved by calling DataSet.WriteXml
ColumnName	The name of the column; this is auto-generated by the runtime if not setin the constructor

DataSet (continued)

- The following table shows the properties that can be set on a DataColumn object

Property	Description
DataType	Defines the System.Type value of the column
DefaultValue	Can define a default value for a column
Expression	Defines the expression to be used in a computed column

Next Step



Recap

Useful links

Thank you

Recap

Useful Links

- Pls paste links from all the slides

US – CORPORATE HEADQUARTERS

1248 Reamwood Avenue,
Sunnyvale, CA 94089
Phone: (408) 743 4400

343 Thornall St 720
Edison, NJ 08837
Phone: (732) 395 6900

UK

57 Rathbone Place,
4th Floor, Holden House,
London, W1T 1JU , UK

89 Worship Street Shoreditch,
London EC2A 2BF, UK
Phone: (44) 2079 938 955

INDIA

Mumbai
4th Floor, Nomura
Powai , Mumbai 400 076
Phone: +91 (22) 3051 1000

Pune
5th floor, Amar Paradigm Baner Road
Baner, Pune 411 045
Phone: +91 (20) 6604 6000

Bangalore
4th Floor, Kabra Excelsior,
80 Feet Main Road, Koramangala 1st Block,
Bengaluru (Bangalore) 560034
Phone: +91 (80) 4666 1666

www.xoriant.com