# 7.1 Entity Framework 6 Support

MySQL Connector/NET integrates support for Entity Framework 6 (EF6), which now includes support for cross-platform application deployment with the EF 6.4 version. This chapter describes how to configure and use the EF6 features that are implemented in Connector/NET.

**In this section:**

## Minimum Requirements for EF6 on Windows Only

- Connector/NET 6.10 or 8.0.11
- MySQL Server 5.6
- Entity Framework 6 assemblies
- .NET Framework 4.5.1 (.NET Framework 4.5.2 for Connector/NET 8.0.22)

## Minimum Requirements for EF 6.4 with Cross-Platform Support

- Connector/NET 8.0.22
- MySQL Server 5.6
- Entity Framework 6.4 assemblies
- .NET Standard 2.1 (.NET Core SDK 3.1 and Visual Studio 2019 version 16.5)

## Configuration

**Note**

> The MySQL Connector/NET 8.0 release series has a naming scheme for EF6 assemblies and NuGet packages that differs from the scheme used with previous release series, such as 6.9 and 6.10. To configure Connector/NET 6.9 or 6.10 for use with EF6, substitute the assembly and package names in this section with the following:
>
> - Assembly: `MySql.Data.Entity.EF6`
> - NuGet package: `MySql.Data.Entity`
>
>   For more information about the `MySql.Data.Entity` NuGet package and its uses, see https://www.nuget.org/packages/MySql.Data.Entity/.

To configure Connector/NET support for EF6:

1. Edit the configuration sections in the `app.config` file to add the connection string and the Connector/NET provider.

```
<connectionStrings>
    <add name="MyContext" providerName="MySql.Data.MySqlClient"
        connectionString="server=localhost;port=3306;database=mycontext;uid=root;
</connectionStrings>
<entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlConnect
    <providers>
        <provider invariantName="MySql.Data.MySqlClient"
            type="MySql.Data.MySqlClient.MySqlProviderServices, MySql.Data.Entity
        <provider invariantName="System.Data.SqlClient"
            type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramewo
    </providers>
</entityFramework>
```

2. Apply the assembly reference using one of the following techniques:

   - **NuGet package.** Install the NuGet package to add this reference automatically to the `app.config` or `web.config` file during the installation. For example, to install the package for Connector/NET 8.0.22, use one of the following installation options:

     - Command Line Interface (CLI)

```
dotnet add package MySql.Data.EntityFramework -Version 8.0.22
```

- Package Manager Console (PMC)

```
Install-Package MySql.Data.EntityFramework -Version 8.0.22
```

- Visual Studio with NuGet Package Manager. For this option, select `nuget.org` as the package source, search for `mysql.data`, and install a stable version of `MySql.Data.EntityFramework`.

- **MySQL Installer or the MySQL Connector/NET MSI file.** Install MySQL Connector/NET and then add a reference for the `MySql.Data.EntityFramework` assembly to your project. Depending on the .NET Framework version used, the assembly is taken from the `v4.0`, `v4.5`, or `v4.8` folder.

- **MySQL Connector/NET source code.** Build Connector/NET from source and then insert the following data provider information into the `app.config` or `web.config` file:

```
<system.data>
   <DbProviderFactories>
     <remove invariant="MySql.Data.MySqlClient" />
     <add name="MySQL Data Provider" invariant="MySql.Data.MySqlClient" descr
          type="MySql.Data.MySqlClient.MySqlClientFactory, MySql.Data, Versic
   </DbProviderFactories>
</system.data>
```

> **Important**
>
> Always update the version number to match the one in the `MySql.Data.dll` assembly.

3. Set the new `DbConfiguration` class for MySQL. This step is optional but highly recommended, because it adds all the dependency resolvers for MySQL classes. This can be done in three ways:

   - Adding the `DbConfigurationTypeAttribute` on the context class:

```
[DbConfigurationType(typeof(MySqlEFConfiguration))]
```

   - Calling `DbConfiguration.SetConfiguration(new MySqlEFConfiguration())` at the application start up.

  - Set the `DbConfiguration` type in the configuration file:

    ```
    <entityFramework codeConfigurationType="MySql.Data.Entity.MySqlEFConfiguratic
    ```

  It is also possible to create a custom `DbConfiguration` class and add the dependency resolvers needed.

## EF6 Features

Following are the new features in Entity Framework 6 implemented in Connector/NET:

- *Cross-platform support* in Connector/NET 8.0.22 implements EF 6.4 as the initial provider version to include Linux and macOS compatibility with .NET Standard 2.1 from Microsoft.
- *Async Query and Save* adds support for the task-based asynchronous patterns that have been available since .NET 4.5. The new asynchronous methods supported by Connector/NET are:
  - `ExecuteNonQueryAsync`
  - `ExecuteScalarAsync`
  - `PrepareAsync`
- *Connection Resiliency / Retry Logic* enables automatic recovery from transient connection failures. To use this feature, add to the `OnCreateModel` method:

  ```
  SetExecutionStrategy(MySqlProviderInvariantName.ProviderName, () => new MySqlExec
  ```

- *Code-Based Configuration* gives you the option of performing configuration in code, instead of performing it in a configuration file, as it has been done traditionally.
- *Dependency Resolution* introduces support for the Service Locator. Some pieces of functionality that can be replaced with custom implementations have been factored out. To add a dependency resolver, use:

  ```
  AddDependencyResolver(new MySqlDependencyResolver());
  ```

  The following resolvers can be added:

  - `DbProviderFactory -> MySqlClientFactory`

  - `IDbConnectionFactory -> MySqlConnectionFactory`
  - `MigrationSqlGenerator -> MySqlMigrationSqlGenerator`
  - `DbProviderServices -> MySqlProviderServices`
  - `IProviderInvariantName -> MySqlProviderInvariantName`
  - `IDbProviderFactoryResolver -> MySqlProviderFactoryResolver`
  - `IManifestTokenResolver -> MySqlManifestTokenResolver`
  - `IDbModelCacheKey -> MySqlModelCacheKeyFactory`
  - `IDbExecutionStrategy -> MySqlExecutionStrategy`
- *Interception/SQL logging* provides low-level building blocks for interception of Entity Framework operations with simple SQL logging built on top:

```
myContext.Database.Log = delegate(string message) { Console.Write(message); };
```

- *DbContext can now be created with a DbConnection that is already opened*, which enables scenarios where it would be helpful if the connection could be open when creating the context (such as sharing a connection between components when you cannot guarantee the state of the connection)

```
  [DbConfigurationType(typeof(MySqlEFConfiguration))]
  class JourneyContext : DbContext
  {
    public DbSet<MyPlace> MyPlaces { get; set; }

    public JourneyContext()
      : base()
    {

    }

    public JourneyContext(DbConnection existingConnection, bool contextOwnsConnec
      : base(existingConnection, contextOwnsConnection)
    {

    }
  }

  using (MySqlConnection conn = new MySqlConnection("<connectionString>"))
  {
    conn.Open();
    ...

    using (var context = new JourneyContext(conn, false))
    {
      ...
    }
  }
```

- *Imp* as improved ways of creating a transaction within the Entity Framework. Starting with Entity Framework 6, `Database.ExecuteSqlCommand()` will wrap by default the command in a transaction if one was not already present. There are overloads of this method that allow users to override this behavior if wished. Execution of stored procedures included in the model through APIs such as `ObjectContext.ExecuteFunction()` does the same. It is also possible to pass an existing transaction to the context.

- *DbSet.AddRange/RemoveRange* provides an optimized way to add or remove multiple entities from a set.

## Code First Features

Following are new Code First features supported by Connector/NET:

- *Code First Mapping to Insert/Update/Delete Stored Procedures* supported:

```
modelBuilder.Entity<EntityType>().MapToStoredProcedures();
```

- *Idempotent migrations scripts* allow you to generate an SQL script that can upgrade a database at any version up to the latest version. To do so, run the **`Update-Database -Script -SourceMigration: $InitialDatabase`** command in Package Manager Console.

- *Configurable Migrations History Table* allows you to customize the definition of the migrations history table.

## Example for Using EF6

The following C# code example represents the structure of an Entity Framework 6 model.

```
using MySql.Data.Entity;
using System.Data.Common;
using System.Data.Entity;
```

```
namespace EF6
{
  // Code-Based Configuration and Dependency resolution
  [DbConfigurationType(typeof(MySqlEFConfiguration))]
  public class Parking : DbContext
  {
    public DbSet<Car> Cars { get; set; }

    public Parking()
      : base()
    {

    }

    // Constructor to use on a DbConnection that is already opened
    public Parking(DbConnection existingConnection, bool contextOwnsConnection)
      : base(existingConnection, contextOwnsConnection)
    {

    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
      base.OnModelCreating(modelBuilder);
      modelBuilder.Entity<Car>().MapToStoredProcedures();
    }
  }

  public class Car
  {
    public int CarId { get; set; }

    public string Model { get; set; }

    public int Year { get; set; }

    public string Manufacturer { get; set; }
  }
}
```

The C# code example that follows shows how to use the entities from the previous model in an application that stores the data within a MySQL table.

```
using MySql.Data.MySqlClient;
using System;
using System.Collections.Generic;
```

```
namespace EF6
{
  class Example
  {
    public static void ExecuteExample()
    {
      string connectionString = "server=localhost;port=3305;database=parking;uid=roo

      using (MySqlConnection connection = new MySqlConnection(connectionString))
      {
        // Create database if not exists
        using (Parking contextDB = new Parking(connection, false))
        {
          contextDB.Database.CreateIfNotExists();
        }

        connection.Open();
        MySqlTransaction transaction = connection.BeginTransaction();

        try
        {
          // DbConnection that is already opened
          using (Parking context = new Parking(connection, false))
          {

            // Interception/SQL logging
            context.Database.Log = (string message) => { Console.WriteLine(message);

            // Passing an existing transaction to the context
            context.Database.UseTransaction(transaction);

            // DbSet.AddRange
            List<Car> cars = new List<Car>();

            cars.Add(new Car { Manufacturer = "Nissan", Model = "370Z", Year = 2012 
            cars.Add(new Car { Manufacturer = "Ford", Model = "Mustang", Year = 2013
            cars.Add(new Car { Manufacturer = "Chevrolet", Model = "Camaro", Year = 
            cars.Add(new Car { Manufacturer = "Dodge", Model = "Charger", Year = 201

            context.Cars.AddRange(cars);

            context.SaveChanges();
          }

          transaction.Commit();
        }
        catch
        {
          transaction.Rollback();
          throw;
        }
      }
    }
  }
}
```