**Performing Asynchronous Operations Using Entity Framework**

**Introduction**

Asynchronous programming involves executing operations in the background so that the main thread can continue its own operations. This way the main thread can keep the user interface responsive while the background thread is processing the task at hand. .NET framework introduced the async and await keywords that simplify asynchronous programming. Entity Framework 6.0 also supports asynchronous operations for querying and saving of the data. This article discusses the basics of using asynchronous operations of Entity Framework in desktop as well as web applications.

**The Async/Await Pattern**

Asynchronous operations can help your application in two possible ways: first, they make your application more responsive to user interactions and second, they can improve the overall performance of your application. Suppose that you wish to execute a lengthy database operation in a desktop application. If the processing is executed on the main thread of the application, it will block the user interaction until the operation is running. That means the user cannot perform actions such as clicking on some controls, moving the application window etc. This is synchronous processing. For web applications synchronous processing can hamper the ability of a resource to handle multiple requests. If the said operation is made to run in an asynchronous manner, the database processing can be run on a background thread whereas the main thread will continue to serve its purpose (user interface rendering or request processing).

There are several ways in which to execute asynchronous operations. However, async/await keywords introduced in .NET Framework 4.5 make your job simple. All you need to follow is the async/await pattern as illustrated by the following code fragment.

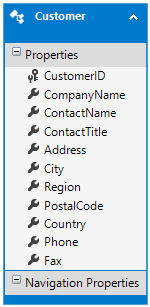
1. public async Task<return\_type> Method1Async()
2. {
3. return await Method2Async();
4. }

The above code shows a function Method1Async() that is supposed to initiate some task in the background. This function has keyword async added to its declaration and returns an instance of Task. The Task instance wraps the actual return value of Method1Async(). By convention asynchronous method names end in "Async". Inside, the Method1Async() calls another method Method2Async(). This method does the actual lengthy processing and returns its value to the calling function (Method1Async() in this case). The await keyword added to the call waits for Method2Async to complete the task. Note that async and await keywords always go hand in hand. It is mandatory for a method using await to be marked as async. .NET framework uses asynchronous operations in many places and now even Entity Framework supports asynchronous querying and saving of data.

Now that you know basics of the async/await pattern, let's develop a simple example that illustrates how Entity Framework supports asynchronous operations.

**Creating a Helper Class to Perform Asynchronous CRUD Operations**

To begin developing this example, create a new Console application using Visual Studio 2013. Then add an ADO.NET Entity Framework Data Model (.edmx) to the project and configure it to use the Customers table of the Northwind database. The following figure shows how the Customer entity class looks in the designer:

  
Customer Entity

Now let's develop a helper class that performs CRUD operations on the Customers table. So, add a new class to the project and name it CustomerHelper. For beginners, the simplest way to create asynchronous operations is to first create them as synchronous and then convert them to asynchronous. This way you can easily understand what a method is supposed to do and then you can migrate the same method to its asynchronous form.

The CustomerHelper class consists of five methods as shown below:

1. public static List<Customer> SelectAll()
2. {
3. }
5. public static Customer SelectByID(string id)
6. {
7. }
9. public static string Insert(Customer obj)
10. {
11. }
13. public static string Update(Customer obj)
14. {
15. }
17. public static string Delete(string id)
18. {
19. }

The above methods are quite straightforward in terms of syntax. The SelectAll() method is supposed to retrieve all the records from the Customers table and return it as a List of Customer entities. The SelectByID() method accepts a CustomerID and returns a single Customer entity for that value. The Insert(), Update() and Delete() methods perform the respective operations and return a string message indicating the result of the operation.

We won't discuss the actual implementation of all these methods here for the want of space. Just to give you an idea SelectAll() and Insert() method implementation is shown below:

1. public static List<Customer> SelectAll()
2. {
3. NorthwindEntities db=new NorthwindEntities();
4. var query = from c in db.Customers
5. orderby c.CustomerID ascending
6. select c;
7. return query.ToList();
8. }
10. public static string Insert(Customer obj)
11. {
12. NorthwindEntities db = new NorthwindEntities();
13. db.Customers.Add(obj);
14. await db.SaveChanges();
15. return "Customer added successfully!";
16. }

The code shown above uses LINQ to Entities to query the data and uses EF objects and their methods to add the data to the Customers table.

Now, that you know the synchronous versions of the methods, let's code the asynchronous versions. The following code shows the SelectAllAsync() method.

1. public **async** static **Task<List<Customer>>** **SelectAllAsync**()
2. {
3. NorthwindEntities db=new NorthwindEntities();
4. var query = from c in db.Customers
5. orderby c.CustomerID ascending
6. select c;
7. return **await** query.**ToListAsync**();
8. }

Notice the following about this method:

* The name of the method has been changed from SelectAll to SelectAllAsync to reflect  the asynchronous method naming convention.
* The SelectAllAsync() method is marked with async keyword.
* The method returns a Task that wraps a List of Customer entities.
* The return keyword is followed by the await keyword.
* Instead of calling the ToList() method the return statement calls the ToListAsync() method.

The ToListAsync() is the asynchronous method of Entity Framework that realizes the data in asynchronous manner.

On the same lines the following code represents the asynchronous versions of the remaining four methods:

1. public static **async** **Task<Customer>** **SelectByIDAsync**(string id)
2. {
3. NorthwindEntities db = new NorthwindEntities();
4. var query = from c in db.Customers
5. where c.CustomerID == id
6. select c;
7. Customer obj = **await** query.**SingleOrDefaultAsync()**;
8. return obj;
9. }
11. public static **async Task<string> InsertAsync**(Customer obj)
12. {
13. NorthwindEntities db = new NorthwindEntities();
14. db.Customers.Add(obj);
15. **await db.SaveChangesAsync();**
16. return "Customer added successfully!";
17. }
19. public static **async** **Task<string> UpdateAsync**(Customer obj)
20. {
21. NorthwindEntities db = new NorthwindEntities();
22. Customer existing = await db.Customers.**FindAsync**(obj.CustomerID);
23. existing.CompanyName = obj.CompanyName;
24. existing.ContactName = obj.ContactName;
25. existing.Country = obj.Country;
26. **await db.SaveChangesAsync();**
27. return "Customer updated successfully!";
28. }
30. public static **async** **Task<string> DeleteAsync**(string id)
31. {
32. NorthwindEntities db = new NorthwindEntities();
33. Customer existing = await db.Customers.**FindAsync**(id);
34. db.Customers.Remove(existing);
35. **await db.SaveChangesAsync();**
36. return "Customer deleted successfully!";
37. }

Notice that the above code uses the following asynchronous methods of Entity Framework:

* **SingleOrDefaultAsync() :**Fetches a single object as a result of the query.
* **FindAsync() :**Searches for a specified key in the DbSet in asynchronous fashion and returns the entity if found.
* **SaveChangesAsync() :**Persists the changes to the database in asynchronous manner.

 This completes the CustomerHelper class. Now, it's time to use it in a Console application.

**Using the Helper Class in a Console Application**

As an example of consuming the CustomerHelper you created in the previous section, let's call the SelectAllAsync() method in the Main() method:

1. static void Main(string[] args)
2. {
3. var task = CustomerHelper.SelectAllAsync();
4. task.Wait();
5. List<Customer> data = task.Result;
6. Console.WriteLine(data.Count);
7. }

The Main() method calls the SelectAllAsync() method and stores its return value (a Task instance) in a variable. The Wait() method of the Task instance waits for all the background processing to complete. The actual return value of the method (List of Customer) is obtained by using the Result method of the Task object. The total number of customers is then outputted on the Console.

On the same lines you can call any other method of the CustomerHelper class.

**Using Asynchronous EF Operations in ASP.NET MVC**

In the previous section you used the CustomerHelper class in a Console application. If you wish to use asynchronous methods of Entity Framework inside a controller action then you can do the following:

1. public async Task<ActionResult> Index()
2. {
3. List<Customer> model = await CustomerHelper.SelectAllAsync();
4. return View(model);
5. }

The above code shows the Index() method of a controller. The Index() method itself is marked with an async keyword and it returns a Task object wrapping the ActionResult. Inside, it calls the SelectAllAsync() method of the CustomerHelper. This time SelectAllAsync() is called with an await keyword and return value collected in a List of Customer. The List is then passed to the Index view as its model.

**Summary**

Entity Framework 6 introduced support for asynchronous operations. Using these asynchronous methods along with the async/await pattern you can query the database asynchronously. You can also save the changes to the database in asynchronous manner. Using the asynchronous methods - ToListAsync(), SingleOrDefaultAsync(),FindAsync(), SaveChangesAsync() - you created a class that performs CRUD operations on the database asynchronously.