

Getting Started with Entity Framework 6 Code First using MVC 5

Tom Dykstra, Rick Anderson

Step By Step, Guide



Async and Stored Procedures with the Entity Framework in an ASP.NET MVC Application

In earlier tutorials you learned how to read and update data using the synchronous programming model. In this tutorial you see how to implement the asynchronous programming model. Asynchronous code can help an application perform better because it makes better use of server resources.

In this tutorial you'll also see how to use stored procedures for insert, update, and delete operations on an entity.

Finally, you'll redeploy the application to Windows Azure, along with all of the database changes that you've implemented since the first time you deployed.

The following illustrations show some of the pages that you'll work with.

←→

http://localhost:54112/Departm

Departments - Contoso Uni...

⌂★⚙

Contoso University

Departments

Create New

Name	Budget	Start Date	Administrator	
Temp	\$0.00	2013-10-29		Edit Details Delete
English	\$350,000.00	2007-09-01	Abercrombie, Kim	Edit Details Delete
Mathematics	\$100,000.00	2007-09-01	Fakhouri, Fadi	Edit Details Delete
Engineering	\$350,000.00	2007-09-01	Harui, Roger	Edit Details Delete
Economics	\$100,000.00	2007-09-01	Kapoor, Candace	Edit Details Delete
New	\$3.00	2013-01-01	Kapoor, Candace	Edit Details Delete
Another	\$1.00	2012-01-01	Harui, Roger	Edit Details Delete

© 2013 - Contoso University

The screenshot shows a web browser window with the address bar displaying `http://localhost:54112/Departm`. The browser tab is titled 'Create - Contoso University'. The page has a dark header with 'Contoso University' and a hamburger menu icon. The main content area is titled 'Create Department' and contains a form with the following fields:

- Name**: A text input field.
- Budget**: A text input field.
- Start Date**: A text input field.
- Administrator**: A dropdown menu with a downward arrow.

Below the form fields is a 'Create' button and a 'Back to List' link.

Why bother with asynchronous code

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server is enabled to handle more traffic without delays.

In earlier versions of .NET, writing and testing asynchronous code was complex, error prone, and hard to debug. In .NET 4.5, writing, testing, and debugging asynchronous code is so much

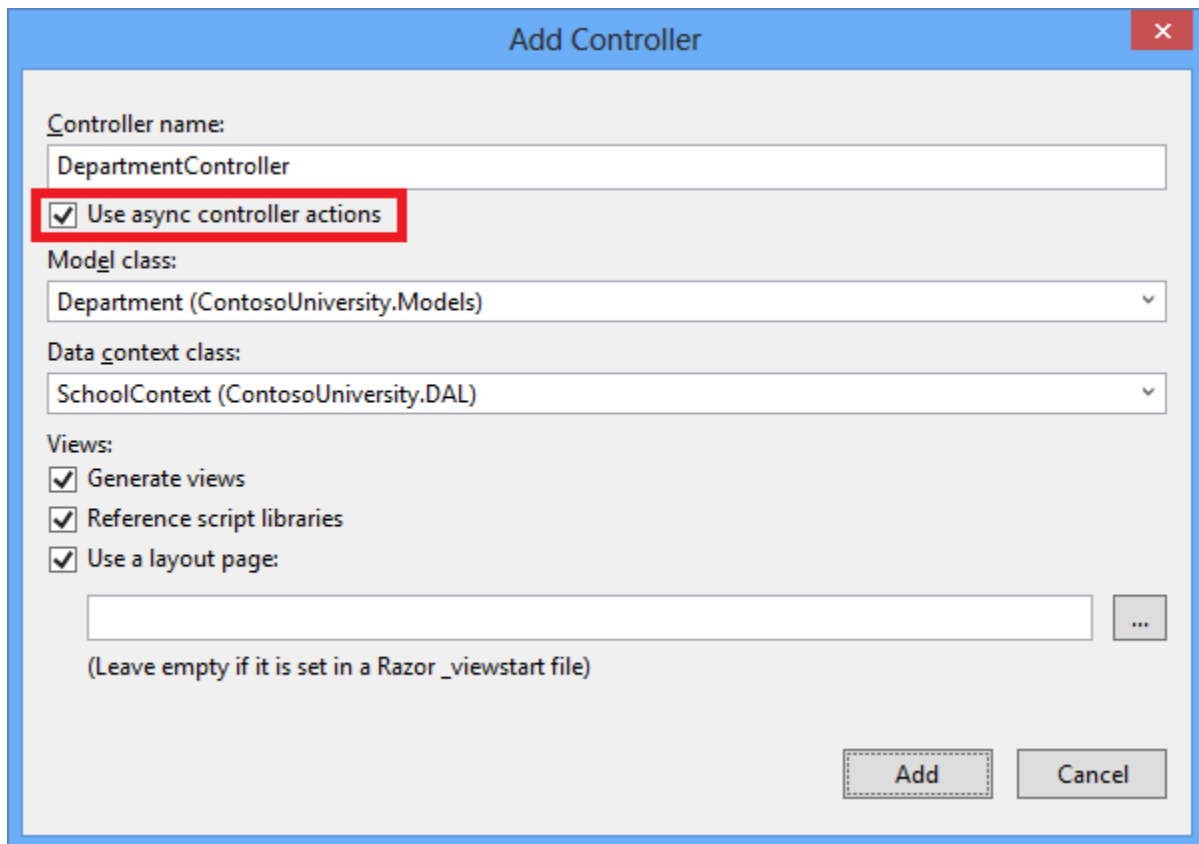
easier that you should generally write asynchronous code unless you have a reason not to. Asynchronous code does introduce a small amount of overhead, but for low traffic situations the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

For more information about asynchronous programming, see the following resources:

- [Entity Framework Async Query and Save](#)
- [Using Asynchronous Methods in ASP.NET MVC 4](#)
- [How to Build ASP./NET Web Applications Using Async](#) (Video)

Create the Department controller

Create a Department controller the same way you did the earlier controllers, except this time select the **Use async controller** actions check box.



The screenshot shows the 'Add Controller' dialog box. The 'Controller name' field contains 'DepartmentController'. The 'Use async controller actions' checkbox is checked and highlighted with a red box. The 'Model class' dropdown shows 'Department (ContosoUniversity.Models)' and the 'Data context class' dropdown shows 'SchoolContext (ContosoUniversity.DAL)'. Under the 'Views' section, three checkboxes are checked: 'Generate views', 'Reference script libraries', and 'Use a layout page:'. Below these is an empty text box and a button with three dots. At the bottom right are 'Add' and 'Cancel' buttons.

The following highlights show how what was added to the synchronous code for the `Index` method to make it asynchronous:

```
public async Task<ActionResult> Index()
{
    var departments = db.Departments.Include(d => d.Administrator);
    return View(await departments.ToListAsync());
}
```

```
}
```

Four changes were applied to enable the Entity Framework database query to execute asynchronously:

- The method is marked with the `async` keyword, which tells the compiler to generate callbacks for parts of the method body and to automatically create the `Task<ActionResult>` object that is returned.
- The return type was changed from `ActionResult` to `Task<ActionResult>`. The `Task<T>` type represents ongoing work with a result of type `T`.
- The `await` keyword was applied to the web service call. When the compiler sees this keyword, behind the scenes it splits the method into two parts. The first part ends with the operation that is started asynchronously. The second part is put into a callback method that is called when the operation completes.
- The asynchronous version of the `ToList` extension method was called.

Why is the `departments.ToList` statement modified but not the `departments = db.Departments` statement? The reason is that only statements that cause queries or commands to be sent to the database are executed asynchronously. The `departments = db.Departments` statement sets up a query but the query is not executed until the `ToList` method is called. Therefore, only the `ToList` method is executed asynchronously.

In the `Details` method and the `HttpGet Edit` and `Delete` methods, the `Find` method is the one that causes a query to be sent to the database, so that's the method that gets executed asynchronously:

```
public async Task<ActionResult> Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Department department = await db.Departments.FindAsync(id);
    if (department == null)
    {
        return HttpNotFound();
    }
    return View(department);
}
```

In the `Create`, `HttpPost Edit`, and `DeleteConfirmed` methods, it is the `SaveChanges` method call that causes a command to be executed, not statements such as `db.Departments.Add(department)` which only cause entities in memory to be modified.

```
public async Task<ActionResult> Create(Department department)
{
    if (ModelState.IsValid)
    {
        db.Departments.Add(department);
        await db.SaveChangesAsync();
    }
}
```

```

        return RedirectToAction("Index");
    }

```

Open *Views\Department\Index.cshtml*, and replace the template code with the following code:

```

@model IEnumerable<ContosoUniversity.Models.Department>
@{
    ViewBag.Title = "Departments";
}
<h2>Departments</h2>
<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Name)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Budget)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.StartDate)
        </th>
        <th>
            Administrator
        </th>
    </tr>
    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Budget)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.StartDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Administrator.FullName)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.DepartmentID }) |
                @Html.ActionLink("Details", "Details", new { id=item.DepartmentID }) |
                @Html.ActionLink("Delete", "Delete", new { id=item.DepartmentID })
            </td>
        </tr>
    }
</table>

```

This code changes the title from Index to Departments, moves the Administrator name to the right, and provides the full name of the administrator.

In the Create, Delete, Details, and Edit views, change the caption for the `InstructorID` field to "Administrator" the same way you changed the department name field to "Department" in the Course views.

In the Create and Edit views use the following code:

```
<label class="control-label col-md-2"
for="InstructorID">Administrator</label>
```

In the Delete and Details views use the following code:

```
<dt>
    Administrator
</dt>
```

Run the application, and click the **Departments** tab.

Name	Budget	Start Date	Administrator	
Temp	\$0.00	2013-10-29		Edit Details Delete
English	\$350,000.00	2007-09-01	Abercrombie, Kim	Edit Details Delete
Mathematics	\$100,000.00	2007-09-01	Fakhouri, Fadi	Edit Details Delete
Engineering	\$350,000.00	2007-09-01	Harui, Roger	Edit Details Delete
Economics	\$100,000.00	2007-09-01	Kapoor, Candace	Edit Details Delete
New	\$3.00	2013-01-01	Kapoor, Candace	Edit Details Delete
Another	\$1.00	2012-01-01	Harui, Roger	Edit Details Delete

© 2013 - Contoso University

Everything works the same as in the other controllers, but in this controller all of the SQL queries are executing asynchronously.

Some things to be aware of when you are using asynchronous programming with the Entity Framework:

- The async code is not thread safe. In other words, in other words, don't try to do multiple operations in parallel using the same context instance.
- If you want to take advantage of the performance benefits of async code, make sure that any library packages that you're using (such as for paging), also use async if they call any Entity Framework methods that cause queries to be sent to the database.

Use stored procedures for inserting, updating, and deleting

Some developers and DBAs prefer to use stored procedures for database access. In earlier versions of Entity Framework you can retrieve data using a stored procedure by [executing a raw SQL query](#), but you can't instruct EF to use stored procedures for update operations. In EF 6 it's easy to configure Code First to use stored procedures.

1. In *DAL\SchoolContext.cs*, add the highlighted code to the `OnModelCreating` method.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    modelBuilder.Entity<Course>()
        .HasMany(c => c.Instructors).WithMany(i => i.Courses)
        .Map(t => t.MapLeftKey("CourseID"))
        .MapRightKey("InstructorID")
        .ToTable("CourseInstructor");
    modelBuilder.Entity<Department>().MapToStoredProcedures();
}
```

This code instructs Entity Framework to use stored procedures for insert, update, and delete operations on the `Department` entity.

2. In Package Manage Console, enter the following command:

```
add-migration DepartmentSP
```

Open *Migrations\<timestamp>_DepartmentSP.cs* to see the code in the `Up` method that creates Insert, Update, and Delete stored procedures:

```
public override void Up()
{
    CreateStoredProcedure(
        "dbo.Department_Insert",
        p => new
        {
            Name = p.String(maxLength: 50),
            Budget = p.Decimal(precision: 19, scale: 4, storeType:
"money"),
            StartDate = p.DateTime(),
            InstructorID = p.Int(),
        },
        body:
        @"INSERT [dbo].[Department] ([Name], [Budget], [StartDate],
[InstructorID])
VALUES (@Name, @Budget, @StartDate, @InstructorID)

DECLARE @DepartmentID int
SELECT @DepartmentID = [DepartmentID]
FROM [dbo].[Department]
WHERE @@ROWCOUNT > 0 AND [DepartmentID] =
scope_identity()

SELECT t0.[DepartmentID]
FROM [dbo].[Department] AS t0
```

```

WHERE @@ROWCOUNT > 0 AND t0.[DepartmentID] =
@DepartmentID"
);

CreateStoredProcedure(
    "dbo.Department_Update",
    p => new
    {
        DepartmentID = p.Int(),
        Name = p.String(maxLength: 50),
        Budget = p.Decimal(precision: 19, scale: 4, storeType:
"money"),
        StartDate = p.DateTime(),
        InstructorID = p.Int(),
    },
    body:
    @"UPDATE [dbo].[Department]
        SET [Name] = @Name, [Budget] = @Budget, [StartDate] =
@StartDate, [InstructorID] = @InstructorID
        WHERE ([DepartmentID] = @DepartmentID)"
);

CreateStoredProcedure(
    "dbo.Department_Delete",
    p => new
    {
        DepartmentID = p.Int(),
    },
    body:
    @"DELETE [dbo].[Department]
        WHERE ([DepartmentID] = @DepartmentID)"
);
}

```

3. In Package Manage Console, enter the following command:

```
update-database
```

4. Run the application in debug mode, click the **Departments** tab, and then click **Create New**.
5. Enter data for a new department, and then click **Create**.

The screenshot shows a web browser window with the address bar displaying `http://localhost:54112/Departm` and a tab titled "Create - Contoso University". The browser's address bar also shows navigation icons (back, forward, search, refresh) and a home icon. The application's header is dark blue with the text "Contoso University" and a hamburger menu icon. The main content area is white and titled "Create Department". It contains four input fields: "Name", "Budget", "Start Date", and "Administrator". The "Administrator" field is a dropdown menu with a downward arrow. Below the input fields is a "Create" button and a "Back to List" link.

Contoso University

Create

Department

Name

Budget

Start Date

Administrator

Create

[Back to List](#)

6. In Visual Studio, look at the logs in the **Output** window to see that a stored procedure was used to insert the new Department row.

```
Output
Show output from: Debug
FROM [dbo].[Instructor] AS [Extent1]:
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ReaderExecuting;Timespan:00:00:00.0
000047;Properties:Command: [dbo].[Department_Insert];
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ReaderExecuting;Timespan:00:00:00.0
000041;Properties:Command: SELECT
[Extent1].[DepartmentID] AS [DepartmentID],
[Extent1].[Name] AS [Name],
[Extent1].[Budget] AS [Budget],
[Extent1].[StartDate] AS [StartDate],
[Extent1].[InstructorID] AS [InstructorID],
[Extent2].[ID] AS [ID],
[Extent2].[LastName] AS [LastName],
[Extent2].[FirstName] AS [FirstName],
[Extent2].[HireDate] AS [HireDate]
FROM [dbo].[Department] AS [Extent1]
LEFT OUTER JOIN [dbo].[Instructor] AS [Extent2] ON [Extent1].
[InstructorID] = [Extent2].[ID];
```

Code First creates default stored procedure names. If you are using an existing database, you might need to customize the stored procedure names in order to use stored procedures already defined in the database. For information about how to do that, see [Entity Framework Code First Insert/Update/Delete Stored Procedures](#).

If you want to customize what generated stored procedures do, you can edit the scaffolded code for the migrations `Up` method that creates the stored procedure. That way your changes are reflected whenever that migration is run and will be applied to your production database when migrations runs automatically in production after deployment.

If you want to change an existing stored procedure that was created in a previous migration, you can use the `Add-Migration` command to generate a blank migration, and then manually write code that calls the [AlterStoredProcedure](#) method.

Deploy to Windows Azure

This section requires you to have completed the optional **Deploying the app to Windows Azure** section in the [Migrations and Deployment](#) tutorial of this series. If you had migrations errors that you resolved by deleting the database in your local project, skip this section.

1. In Visual Studio, right-click the project in **Solution Explorer** and select **Publish** from the context menu.
2. Click **Publish**.

Visual Studio deploys the application to Windows Azure, and the application opens in your default browser, running in Windows Azure.

3. Test the application to verify it's working.

The first time you run a page that accesses the database, the Entity Framework runs all of the migrations `Up` methods required to bring the database up to date with the current data model. You can now use all of the web pages that you added since the last time you deployed, including the Department pages that you added in this tutorial.

Summary

In this tutorial you saw how to improve server efficiency by writing code that executes asynchronously, and how to use stored procedures for insert, update, and delete operations. In the next tutorial, you'll see how to prevent data loss when multiple users try to edit the same record at the same time.