**CRUD (Create, Read, Update, Delete)**

The IIS Administration API provides direct access to resources on the system. Many of these resources allow create, read, update and delete operations. The REST API maps CRUD operations to HTTP methods. The following table specifies which HTTP method maps to which operation.

| **CRUD (CREATE, READ, UPDATE, DELETE)** | |
| --- | --- |
| **CRUD Operation** | **HTTP Method** |
| Create | POST |
| Read | GET |
| Update | PATCH / PUT |
| Delete | DELETE |

**Create (POST)**

Resources are created by sending HTTP POST requests to the API. The type of resource is determined by the URL of the request. The body of the request should contain a JSON object describing the resource to create. The object in the request body determines the initial state of the resource will be when it is created. Some resources require certain properties be provided when they are created, others can be created with an empty JSON object.

Creating a resource while setting the name property. **POST**

Copy

{

"name": "Example Resource Name"

}

**Creating a resource that belongs to another**

Sometimes resources are created that are meant to belong to another resource. For example, if *applications* must belong to a web site and someone wanted to create a new *application* then they would specify that relationship during the creation of the application as shown below.

Creating an application for a web site. **POST**

Copy

{

"path": "/MyApp",

"physical\_path": "c:/sites/mysite/myapp",

"website": {

"id": {website\_id}

}

}

**Read (GET)**

Resources are retrieved by performing HTTP GET requests. There are two main methods to retrieve resources. The first method involves requesting a list of resources, the second method is when a single resource is requested. Requests to a single resource are marked by the presence of the resource **id** in the URL of the request. Sometimes, singular resources can also be specified through query string paremeters in the URL. This behavior depends on the individual API endpoint.

**Retrieving multiple resources**

Reading lists of resources is done by requesting a resource endpoint without specifying an individual resources **id**. Sometimes resources require query string parameters or else they cannot produce valid lists. For example IIS applications live at the */api/webserver/webapps* endpoint, but requesting that endpoint alone would produce no information. This is because a web site must be specified to tell the API which applications should be shown. So consumers would request */api/webserver/webapps?website.id={website\_id}* to see a list of applications.

Retrieving a list of resources. **GET** */api/websites*

Copy

{

"websites": [

{

"name": "Default Web Site",

"id": "{id}",

"status": "started",

"\_links": {

"self": {

"href": "/api/webserver/websites/{id}"

}

}

},

{

"name": "My Site",

"id": "{id\_1}",

"status": "started",

"\_links": {

"self": {

"href": "/api/webserver/websites/{id\_1}"

}

}

}

{

"name": "docs",

"id": "{id\_2}",

"status": "started",

"\_links": {

"self": {

"href": "/api/webserver/websites/{id\_2}"

}

}

}

]

}

**Retrieving individual resources**

Resources are retrieved on an individual basis by providing the **id** of the resource in the URL of the resource endpoint. Some API endpoints also allow specifying individual resources by providing uniquely identifying query string parameters. For example, a file can be retrieved by providing the **id** of the file in the URL or by providing the **physical\_path** of the file.

The file resource allows multiple methods to retrieve individual files:

* */api/files/{id}*
* */api/files?physical\_path={physical path of the file}*

The files endpoint provides this behavior because only one file can exist for any given physical path, so it is a **uniquely identifying** query string parameter.

**Update (PATCH / PUT)**

Updates are performed by issuing HTTP PATCH requests to the URL that the resource is located at. When a PATCH request is performed, the properties of the request body are read, and if the resource has a property with the same name the property of the resource will be set to the new value.

**Example resource before PATCH**

Copy

{

"name": "My Site",

"id": "12345",

"physical\_path": "c:\\sites\\mysite"

"\_links": {

"self": {

"href": "/api/webserver/websites/{12345}"

}

}

}

**Performing the PATCH request**

Patch request to update the name of the resource. **PATCH** */api/webserver/websites/12345*

Copy

{

"name": "My Site 2"

}

**Resource after PATCH**

Copy

{

"name": "My Site 2",

"id": "12345",

"physical\_path": "c:\\sites\\mysite"

"\_links": {

"self": {

"href": "/api/webserver/websites/{12345}"

}

}

}

**Delete (DELETE)**

Resources are deleted by sending an HTTP DELETE request to the URL that the resource is located at. This is the URL that contains the **id** of the resource

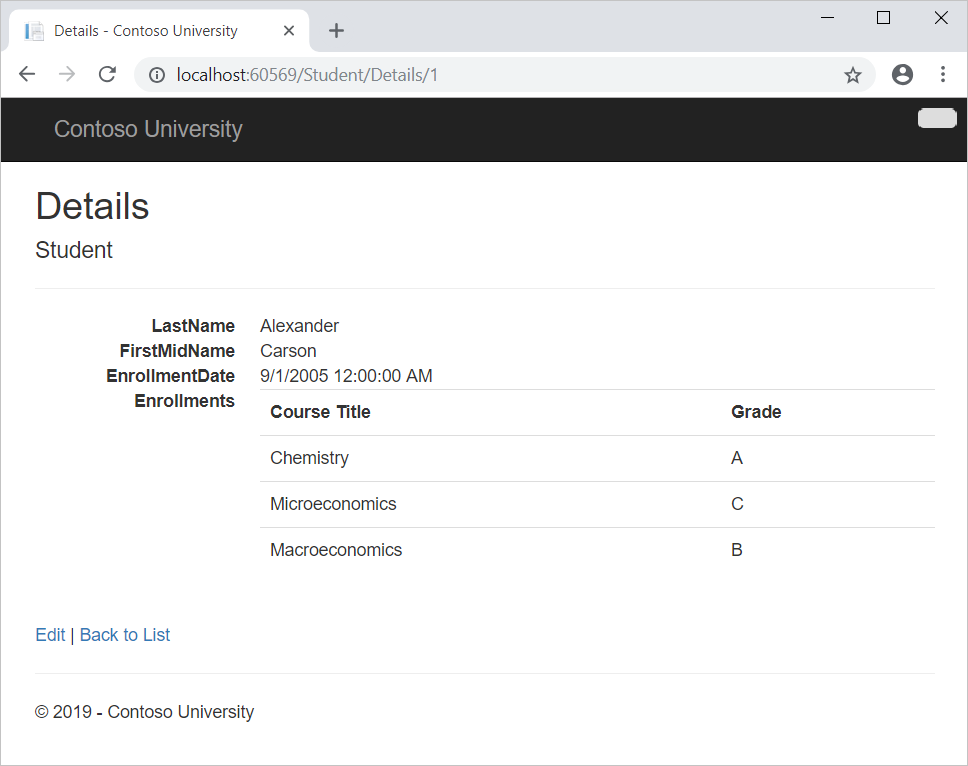
**Tutorial: Implement CRUD Functionality with the Entity Framework in ASP.NET MVC**

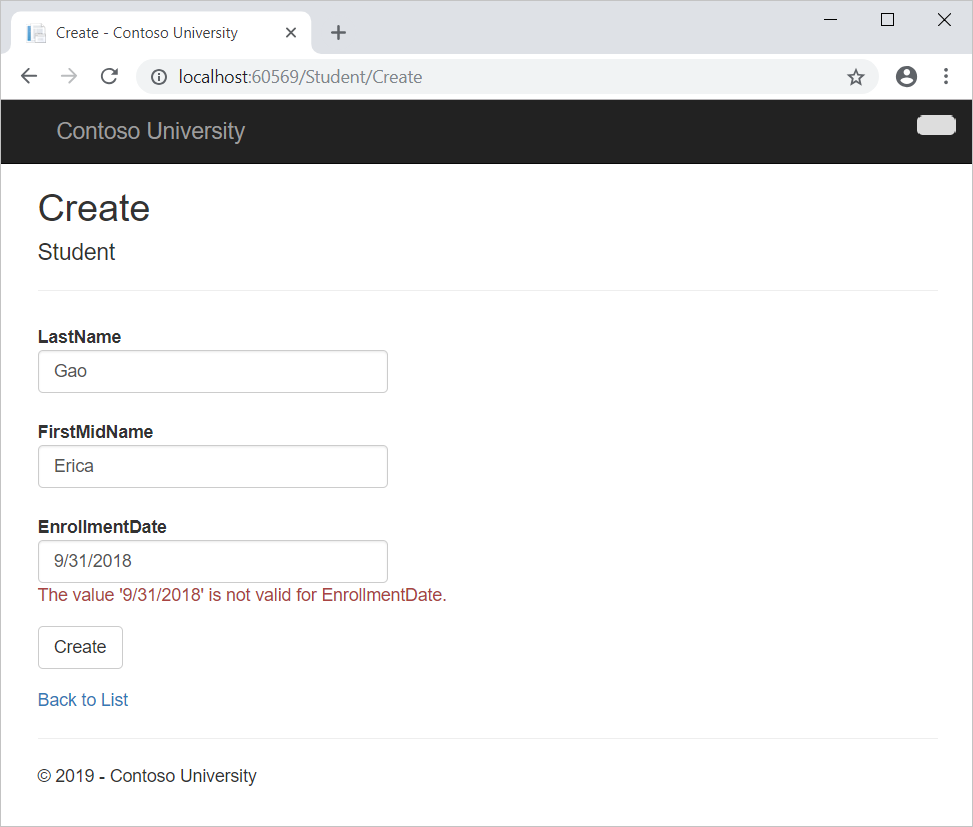
In the [previous tutorial](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application), you created an MVC application that stores and displays data using the Entity Framework (EF) 6 and SQL Server LocalDB. In this tutorial, you review and customize the create, read, update, delete (CRUD) code that the MVC scaffolding automatically creates for you in controllers and views.

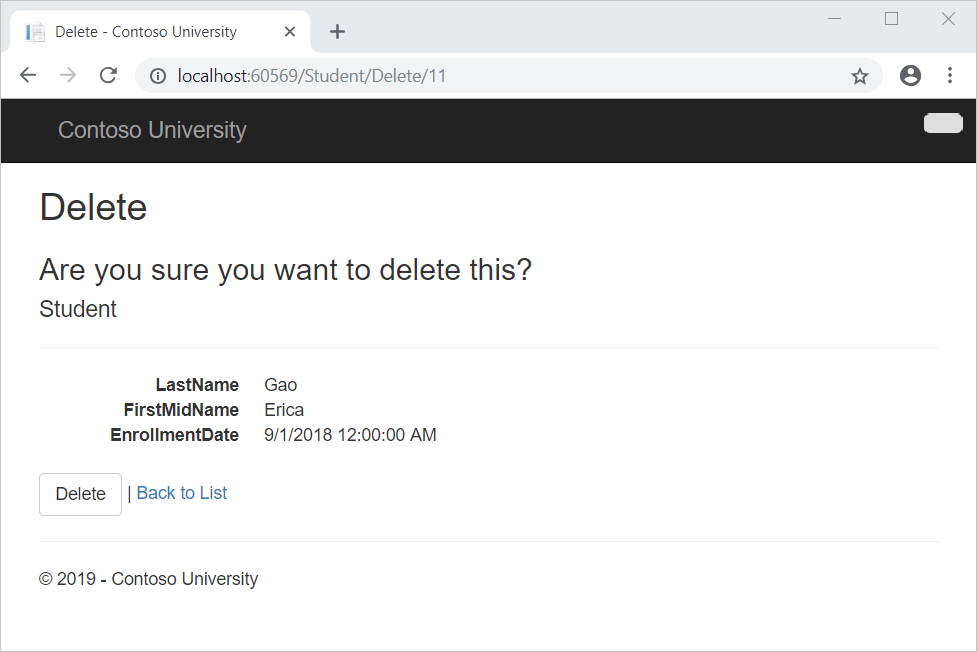
**Note**

It's a common practice to implement the repository pattern in order to create an abstraction layer between your controller and the data access layer. To keep these tutorials simple and focused on teaching how to use EF 6 itself, they don't use repositories. For info about how to implement repositories, see the [**ASP.NET Data Access Content Map**](https://docs.microsoft.com/en-us/aspnet/whitepapers/aspnet-data-access-content-map).

Here are examples of the web pages you create:







In this tutorial, you:

* Create a Details page
* Update the Create page
* Update the HttpPost Edit method
* Update the Delete page
* Close database connections
* Handle transactions

**Prerequisites**

* [Create the Entity Framework Data Model](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application)

**Create a Details page**

The scaffolded code for the Students Index page left out the Enrollments property, because that property holds a collection. In the Details page, you'll display the contents of the collection in an HTML table.

In *Controllers\StudentController.cs*, the action method for the Details view uses the [Find](https://msdn.microsoft.com/library/gg696418(v=VS.103).aspx) method to retrieve a single Student entity.

C#Copy

public ActionResult Details(int? id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

Student student = db.Students.Find(id);

if (student == null)

{

return HttpNotFound();

}

return View(student);

}

The key value is passed to the method as the id parameter and comes from *route data* in the **Details** hyperlink on the Index page.

**Tip: Route data**

Route data is data that the model binder found in a URL segment specified in the routing table. For example, the default route specifies controller, action, and id segments:

C#Copy

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

In the following URL, the default route maps Instructor as the controller, Index as the action and 1 as the id; these are route data values.

http://localhost:1230/Instructor/Index/1?courseID=2021

?courseID=2021 is a query string value. The model binder will also work if you pass the id as a query string value:

http://localhost:1230/Instructor/Index?id=1&CourseID=2021

The URLs are created by ActionLink statements in the Razor view. In the following code, the id parameter matches the default route, so id is added to the route data.

CSHTMLCopy

@Html.ActionLink("Select", "Index", new { id = item.PersonID })

In the following code, courseID doesn't match a parameter in the default route, so it's added as a query string.

CSHTMLCopy

@Html.ActionLink("Select", "Index", new { courseID = item.CourseID })

**To create the Details page**

1. Open *Views\Student\Details.cshtml*.

Each field is displayed using a DisplayFor helper, as shown in the following example:

CSHTMLCopy

<dt>

@Html.DisplayNameFor(model => model.LastName)

</dt>

<dd>

@Html.DisplayFor(model => model.LastName)

</dd>

1. After the EnrollmentDate field and immediately before the closing </dl> tag, add the highlighted code to display a list of enrollments, as shown in the following example:

CSHTMLCopy

<dt>

@Html.DisplayNameFor(model => model.EnrollmentDate)

</dt>

<dd>

@Html.DisplayFor(model => model.EnrollmentDate)

</dd>

<dt>

@Html.DisplayNameFor(model => model.Enrollments)

</dt>

<dd>

<table class="table">

<tr>

<th>Course Title</th>

<th>Grade</th>

</tr>

@foreach (var item in Model.Enrollments)

{

<tr>

<td>

@Html.DisplayFor(modelItem => item.Course.Title)

</td>

<td>

@Html.DisplayFor(modelItem => item.Grade)

</td>

</tr>

}

</table>

</dd>

</dl>

</div>

<p>

@Html.ActionLink("Edit", "Edit", new { id = Model.ID }) |

@Html.ActionLink("Back to List", "Index")

</p>

If code indentation is wrong after you paste the code, press **Ctrl**+**K**, **Ctrl**+**D** to format it.

This code loops through the entities in the Enrollments navigation property. For each Enrollment entity in the property, it displays the course title and the grade. The course title is retrieved from the Course entity that's stored in the Course navigation property of the Enrollments entity. All of this data is retrieved from the database automatically when it's needed. In other words, you are using lazy loading here. You did not specify *eager loading* for the Courses navigation property, so the enrollments were not retrieved in the same query that got the students. Instead, the first time you try to access the Enrollments navigation property, a new query is sent to the database to retrieve the data. You can read more about lazy loading and eager loading in the [Reading Related Data](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/reading-related-data-with-the-entity-framework-in-an-asp-net-mvc-application) tutorial later in this series.

1. Open the Details page by starting the program (**Ctrl**+**F5**), selecting the **Students** tab, and then clicking the **Details** link for Alexander Carson. (If you press **Ctrl**+**F5** while the *Details.cshtml* file is open, you get an HTTP 400 error. This is because Visual Studio tries to run the Details page, but it wasn't reached from a link that specifies the student to display. If that happens, remove "Student/Details" from the URL and try again, or, close the browser, right-click the project, and click **View** > **View in Browser**.)

You see the list of courses and grades for the selected student.

1. Close the browser.

**Update the Create page**

1. In *Controllers\StudentController.cs*, replace the [HttpPostAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.httppostattribute) Create action method with the following code. This code adds a try-catch block and removes ID from the [BindAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.bindattribute) attribute for the scaffolded method:

C#Copy

[HttpPost]

[ValidateAntiForgeryToken]

public ActionResult Create([Bind(Include = "LastName, FirstMidName, EnrollmentDate")]Student student)

{

try

{

if (ModelState.IsValid)

{

db.Students.Add(student);

db.SaveChanges();

return RedirectToAction("Index");

}

}

catch (DataException /\* 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.");

}

return View(student);

}

This code adds the Student entity created by the ASP.NET MVC model binder to the Students entity set and then saves the changes to the database. *Model binder* refers to the ASP.NET MVC functionality that makes it easier for you to work with data submitted by a form; a model binder converts posted form values to CLR types and passes them to the action method in parameters. In this case, the model binder instantiates a Student entity for you using property values from the Form collection.

You removed ID from the Bind attribute because ID is the primary key value which SQL Server will set automatically when the row is inserted. Input from the user does not set the ID value.

**Security warning - The ValidateAntiForgeryToken attribute helps prevent**[**cross-site request forgery**](https://docs.microsoft.com/en-us/aspnet/mvc/overview/security/xsrfcsrf-prevention-in-aspnet-mvc-and-web-pages)**attacks. It requires a corresponding Html.AntiForgeryToken() statement in the view, which you'll see later.**

The Bind attribute is one way to protect against *over-posting* in create scenarios. For example, suppose the Student entity includes a Secret property that you don't want this web page to set.

C#Copy

public class Student

{

public int ID { get; set; }

public string LastName { get; set; }

public string FirstMidName { get; set; }

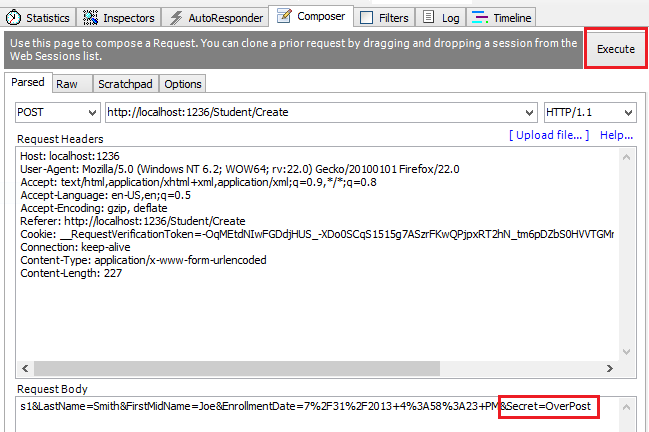
public DateTime EnrollmentDate { get; set; }

public string Secret { get; set; }

public virtual ICollection<Enrollment> Enrollments { get; set; }

}

Even if you don't have a Secret field on the web page, a hacker could use a tool such as [fiddler](http://fiddler2.com/home), or write some JavaScript, to post a Secret form value. Without the [BindAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.bindattribute) attribute limiting the fields that the model binder uses when it creates a Student instance*,* the model binder would pick up that Secret form value and use it to create the Student entity instance. Then whatever value the hacker specified for the Secret form field would be updated in your database. The following image shows the fiddler tool adding the Secret field (with the value "OverPost") to the posted form values.



The value "OverPost" would then be successfully added to the Secret property of the inserted row, although you never intended that the web page be able to set that property.

It's best to use the Include parameter with the Bind attribute to *whitelist* fields. It's also possible to use the Exclude parameter to *blacklist* fields you want to exclude. The reason Include is more secure is that when you add a new property to the entity, the new field is not automatically protected by an Exclude list.

You can prevent overposting in edit scenarios is by reading the entity from the database first and then calling TryUpdateModel, passing in an explicit allowed properties list. That is the method used in these tutorials.

An alternative way to prevent overposting that is preferred by many developers is to use view models rather than entity classes with model binding. Include only the properties you want to update in the view model. Once the MVC model binder has finished, copy the view model properties to the entity instance, optionally using a tool such as [AutoMapper](http://automapper.org/). Use db.Entry on the entity instance to set its state to Unchanged, and then set Property("PropertyName").IsModified to true on each entity property that is included in the view model. This method works in both edit and create scenarios.

Other than the Bind attribute, the try-catch block is the only change you've made to the scaffolded code. If an exception that derives from [DataException](https://docs.microsoft.com/en-us/dotnet/api/system.data.dataexception) is caught while the changes are being saved, a generic error message is displayed. [DataException](https://docs.microsoft.com/en-us/dotnet/api/system.data.dataexception) exceptions are sometimes caused by something external to the application rather than a programming error, so the user is advised to try again. Although not implemented in this sample, a production quality application would log the exception. For more information, see the **Log for insight** section in [Monitoring and Telemetry (Building Real-World Cloud Apps with Azure)](https://docs.microsoft.com/en-us/aspnet/aspnet/overview/developing-apps-with-windows-azure/building-real-world-cloud-apps-with-windows-azure/monitoring-and-telemetry#log).

The code in *Views\Student\Create.cshtml* is similar to what you saw in *Details.cshtml*, except that EditorFor and ValidationMessageFor helpers are used for each field instead of DisplayFor. Here is the relevant code:

CSHTMLCopy

<div class="form-group">

@Html.LabelFor(model => model.LastName, new { @class = "control-label col-md-2" })

<div class="col-md-10">

@Html.EditorFor(model => model.LastName)

@Html.ValidationMessageFor(model => model.LastName)

</div>

</div>

*Create.cshtml* also includes @Html.AntiForgeryToken(), which works with the ValidateAntiForgeryToken attribute in the controller to help prevent [cross-site request forgery](https://docs.microsoft.com/en-us/aspnet/mvc/overview/security/xsrfcsrf-prevention-in-aspnet-mvc-and-web-pages) attacks.

No changes are required in *Create.cshtml*.

1. Run the page by starting the program, selecting the **Students** tab, and then clicking **Create New**.
2. Enter names and an invalid date and click **Create** to see the error message.

This is server-side validation that you get by default. In a later tutorial, you'll see how to add attributes that generate code for client-side validation. The following highlighted code shows the model validation check in the **Create** method.

C#Copy

if (ModelState.IsValid)

{

db.Students.Add(student);

db.SaveChanges();

return RedirectToAction("Index");

}

1. Change the date to a valid value and click **Create** to see the new student appear in the **Index** page.
2. Close the browser.

**Update HttpPost Edit method**

1. Replace the [HttpPostAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.httppostattribute) Edit action method with the following code:

C#Copy

[HttpPost, ActionName("Edit")]

[ValidateAntiForgeryToken]

public ActionResult EditPost(int? id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

var studentToUpdate = db.Students.Find(id);

if (TryUpdateModel(studentToUpdate, "",

new string[] { "LastName", "FirstMidName", "EnrollmentDate" }))

{

try

{

db.SaveChanges();

return RedirectToAction("Index");

}

catch (DataException /\* 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.");

}

}

return View(studentToUpdate);

}

**Note**

In *Controllers\StudentController.cs*, the HttpGet Edit method (the one without the HttpPost attribute) uses the Find method to retrieve the selected Student entity, as you saw in the Details method. You don't need to change this method.

These changes implement a security best practice to prevent [overposting](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/implementing-basic-crud-functionality-with-the-entity-framework-in-asp-net-mvc-application" \l "overpost), The scaffolder generated a Bind attribute and added the entity created by the model binder to the entity set with a Modified flag. That code is no longer recommended because the Bind attribute clears out any pre-existing data in fields not listed in the Include parameter. In the future, the MVC controller scaffolder will be updated so that it doesn't generate Bind attributes for Edit methods.

The new code reads the existing entity and calls [TryUpdateModel](https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.controller.tryupdatemodel) to update fields from user input in the posted form data. The Entity Framework's automatic change tracking sets the [EntityState.Modified](https://docs.microsoft.com/en-us/dotnet/api/system.data.entitystate" \l "System_Data_EntityState_Modified) flag on the entity. When the [SaveChanges](https://msdn.microsoft.com/library/system.data.entity.dbcontext.savechanges(v=VS.103).aspx) method is called, the [Modified](https://docs.microsoft.com/en-us/dotnet/api/system.data.entitystate#System_Data_EntityState_Modified) flag causes the Entity Framework to create SQL statements to update the database row. [Concurrency conflicts](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/handling-concurrency-with-the-entity-framework-in-an-asp-net-mvc-application) are ignored, and all columns of the database row are updated, including those that the user didn't change. (A later tutorial shows how to handle concurrency conflicts, and if you only want individual fields to be updated in the database, you can set the entity to [EntityState.Unchanged](https://docs.microsoft.com/en-us/dotnet/api/system.data.entitystate" \l "System_Data_EntityState_Unchanged) and set individual fields to [EntityState.Modified](https://docs.microsoft.com/en-us/dotnet/api/system.data.entitystate" \l "System_Data_EntityState_Modified).)

To prevent overposting, the fields that you want to be updateable by the Edit page are whitelisted in the TryUpdateModel parameters. Currently there are no extra fields that you're protecting, but listing the fields that you want the model binder to bind ensures that if you add fields to the data model in the future, they're automatically protected until you explicitly add them here.

As a result of these changes, the method signature of the HttpPost Edit method is the same as the HttpGet edit method; therefore you've renamed the method EditPost.

**Tip**

**Entity States and the Attach and SaveChanges Methods**

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database, and this information determines what happens when you call the SaveChanges method. For example, when you pass a new entity to the [**Add**](https://msdn.microsoft.com/library/system.data.entity.dbset.add(v=vs.103).aspx) method, that entity's state is set to Added. Then when you call the **[SaveChanges](https://msdn.microsoft.com/library/system.data.entity.dbcontext.savechanges(v=VS.103).aspx)** method, the database context issues a SQL INSERT command.

An entity may be in one of the following [**states**](https://docs.microsoft.com/en-us/dotnet/api/system.data.entitystate):

* + Added. The entity does not yet exist in the database. The SaveChanges method must issue an INSERT statement.
  + Unchanged. Nothing needs to be done with this entity by the SaveChanges method. When you read an entity from the database, the entity starts out with this status.
  + Modified. Some or all of the entity's property values have been modified. The SaveChanges method must issue an UPDATE statement.
  + Deleted. The entity has been marked for deletion. The SaveChanges method must issue a DELETE statement.
  + Detached. The entity isn't being tracked by the database context.

In a desktop application, state changes are typically set automatically. In a desktop type of application, you read an entity and make changes to some of its property values. This causes its entity state to automatically be changed to Modified. Then when you call SaveChanges, the Entity Framework generates a SQL UPDATE statement that updates only the actual properties that you changed.

The disconnected nature of web apps doesn't allow for this continuous sequence. The **[DbContext](https://msdn.microsoft.com/library/system.data.entity.dbcontext(v=VS.103).aspx)** that reads an entity is disposed after a page is rendered. When the HttpPost Edit action method is called, a new request is made and you have a new instance of the **[DbContext](https://msdn.microsoft.com/library/system.data.entity.dbcontext(v=VS.103).aspx)**, so you have to manually set the entity state to Modified. Then when you call SaveChanges, the Entity Framework updates all columns of the database row, because the context has no way to know which properties you changed.

If you want the SQL Update statement to update only the fields that the user actually changed, you can save the original values in some way (such as hidden fields) so that they are available when the HttpPost Edit method is called. Then you can create a Student entity using the original values, call the Attach method with that original version of the entity, update the entity's values to the new values, and then call SaveChanges. For more information, see [**Entity states and SaveChanges**](https://docs.microsoft.com/en-us/ef/ef6/saving/change-tracking/entity-state) and [**Local Data**](https://docs.microsoft.com/en-us/ef/ef6/querying/local-data).

The HTML and Razor code in *Views\Student\Edit.cshtml* is similar to what you saw in *Create.cshtml*, and no changes are required.

1. Run the page by starting the program, selecting the **Students** tab, and then clicking an **Edit** hyperlink.
2. Change some of the data and click **Save**. You see the changed data in the Index page.
3. Close the browser.

**Update the Delete page**

In *Controllers\StudentController.cs*, the template code for the [HttpGetAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.httpgetattribute) Delete method uses the Find method to retrieve the selected Student entity, as you saw in the Details and Edit methods. However, to implement a custom error message when the call to SaveChanges fails, you'll add some functionality to this method and its corresponding view.

As you saw for update and create operations, delete operations require two action methods. The method that is called in response to a GET request displays a view that gives the user a chance to approve or cancel the delete operation. If the user approves it, a POST request is created. When that happens, the HttpPost Delete method is called and then that method actually performs the delete operation.

You'll add a try-catch block to the [HttpPostAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.httppostattribute) Delete method to handle any errors that might occur when the database is updated. If an error occurs, the [HttpPostAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.httppostattribute) Delete method calls the [HttpGetAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.httpgetattribute) Delete method, passing it a parameter that indicates that an error has occurred. The [HttpGetAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.httpgetattribute) Delete method then redisplays the confirmation page along with the error message, giving the user an opportunity to cancel or try again.

1. Replace the [HttpGetAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.httpgetattribute) Delete action method with the following code, which manages error reporting:

C#Copy

public ActionResult Delete(int? id, bool? saveChangesError=false)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

if (saveChangesError.GetValueOrDefault())

{

ViewBag.ErrorMessage = "Delete failed. Try again, and if the problem persists see your system administrator.";

}

Student student = db.Students.Find(id);

if (student == null)

{

return HttpNotFound();

}

return View(student);

}

This code accepts an [optional parameter](https://msdn.microsoft.com/library/dd264739.aspx) that indicates whether the method was called after a failure to save changes. This parameter is false when the HttpGet Delete method is called without a previous failure. When it is called by the HttpPost Delete method in response to a database update error, the parameter is true and an error message is passed to the view.

1. Replace the [HttpPostAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.httppostattribute) Delete action method (named DeleteConfirmed) with the following code, which performs the actual delete operation and catches any database update errors.

C#Copy

[HttpPost]

[ValidateAntiForgeryToken]

public ActionResult Delete(int id)

{

try

{

Student student = db.Students.Find(id);

db.Students.Remove(student);

db.SaveChanges();

}

catch (DataException/\* dex \*/)

{

//Log the error (uncomment dex variable name and add a line here to write a log.

return RedirectToAction("Delete", new { id = id, saveChangesError = true });

}

return RedirectToAction("Index");

}

This code retrieves the selected entity, then calls the [Remove](https://msdn.microsoft.com/library/system.data.entity.dbset.remove(v=vs.103).aspx) method to set the entity's status to Deleted. When SaveChanges is called, a SQL DELETE command is generated. You have also changed the action method name from DeleteConfirmed to Delete. The scaffolded code named the HttpPost Delete method DeleteConfirmed to give the HttpPost method a unique signature. (The CLR requires overloaded methods to have different method parameters.) Now that the signatures are unique, you can stick with the MVC convention and use the same name for the HttpPost and HttpGet delete methods.

If improving performance in a high-volume application is a priority, you could avoid an unnecessary SQL query to retrieve the row by replacing the lines of code that call the Find and Remove methods with the following code:

C#Copy

Student studentToDelete = new Student() { ID = id };

db.Entry(studentToDelete).State = EntityState.Deleted;

This code instantiates a Student entity using only the primary key value and then sets the entity state to Deleted. That's all that the Entity Framework needs in order to delete the entity.

As noted, the HttpGet Delete method doesn't delete the data. Performing a delete operation in response to a GET request (or for that matter, performing any edit operation, create operation, or any other operation that changes data) creates a security risk. For more information, see [ASP.NET MVC Tip #46 — Don't use Delete Links because they create Security Holes](http://stephenwalther.com/blog/archive/2009/01/21/asp.net-mvc-tip-46-ndash-donrsquot-use-delete-links-because.aspx) on Stephen Walther's blog.

1. In *Views\Student\Delete.cshtml*, add an error message between the h2 heading and the h3 heading, as shown in the following example:

CSHTMLCopy

<h2>Delete</h2>

<p class="error">@ViewBag.ErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>

1. Run the page by starting the program, selecting the **Students** tab, and then clicking a **Delete** hyperlink.
2. Choose **Delete** on the page that says **Are you sure you want to delete this?**.

The Index page displays without the deleted student. (You'll see an example of the error handling code in action in the [concurrency tutorial](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/handling-concurrency-with-the-entity-framework-in-an-asp-net-mvc-application).)

**Close database connections**

To close database connections and free up the resources they hold as soon as possible, dispose the context instance when you are done with it. That is why the scaffolded code provides a [Dispose](https://msdn.microsoft.com/library/system.idisposable.dispose(v=vs.110).aspx) method at the end of the StudentController class in *StudentController.cs*, as shown in the following example:

C#Copy

protected override void Dispose(bool disposing)

{

if (disposing)

{

db.Dispose();

}

base.Dispose(disposing);

}

The base Controller class already implements the IDisposable interface, so this code simply adds an override to the Dispose(bool) method to explicitly dispose the context instance.

**Handle transactions**

By default the Entity Framework implicitly implements transactions. In scenarios where you make changes to multiple rows or tables and then call SaveChanges, the Entity Framework automatically makes sure that either all of your changes succeed or all fail. If some changes are done first and then an error happens, those changes are automatically rolled back. For scenarios where you need more control—for example, if you want to include operations done outside of Entity Framework in a transaction—see [Working with Transactions](https://docs.microsoft.com/en-us/ef/ef6/saving/transactions).

**Get the code**

[Download Completed Project](https://webpifeed.blob.core.windows.net/webpifeed/Partners/ASP.NET%20MVC%20Application%20Using%20Entity%20Framework%20Code%20First.zip)

**Additional resources**

You now have a complete set of pages that perform simple CRUD operations for Student entities. You used MVC helpers to generate UI elements for data fields. For more info about MVC helpers, see [Rendering a Form Using HTML Helpers](https://docs.microsoft.com/en-us/previous-versions/aspnet/dd410596(v=vs.98)) (the article is for MVC 3 but is still relevant for MVC 5).

Links to other EF 6 resources can be found in [ASP.NET Data Access - Recommended Resources](https://docs.microsoft.com/en-us/aspnet/whitepapers/aspnet-data-access-content-map).

**Next steps**

In this tutorial, you:

* Created a Details page
* Updated the Create page
* Updated the HttpPost Edit method
* Updated the Delete page
* Closed database connections
* Handled transactions

Advance to the next article to learn how to add sorting, filtering, and paging to the project.

[Sorting, Filtering, and Paging](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/sorting-filtering-and-paging-with-the-entity-framework-in-an-asp-net-mvc-application)

**Recommended content**

**[Tutorial: Get started with EF Database First using MVC 5](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/database-first-development/setting-up-database)**

This tutorial shows how to start with an existing database and quickly create a web application that enables users to interact with the data.

**[Tutorial: Read related data with EF in an ASP.NET MVC app](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/reading-related-data-with-the-entity-framework-in-an-asp-net-mvc-application)**

In this tutorial you'll read and display related data — that is, data that the Entity Framework loads into navigation properties.

**[Creating an Entity Framework Data Model for an ASP.NET MVC Application (1 of 10)](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application)**

A newer version of this tutorial series is available, for Visual Studio 2013, Entity Framework 6, and MVC 5. The Contoso University sample web application de...

**[Tutorial: Create a more complex data model for an ASP.NET MVC app](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-a-more-complex-data-model-for-an-asp-net-mvc-application)**

In this tutorial you'll add more entities and relationships and you'll customize the data model by specifying formatting, validation, and database mapping rules.

**[Tutorial: Create the Web Application and Data Models for EF Database First with ASP.NET MVC](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/database-first-development/creating-the-web-application)**

This tutorial focuses on creating the web application, and generating the data models based on your database tables.

**[Tutorial: Enhance data validation for EF Database First with ASP.NET MVC app](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/database-first-development/enhancing-data-validation)**

This tutorial focuses on adding data annotations to the data model to specify validation requirements and display formatting.

**[Tutorial: Change the database for EF Database First with ASP.NET MVC app](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/database-first-development/changing-the-database)**

This tutorial focuses on making an update to the database structure and propagating that change throughout the web application.

**[Tutorial: Learn about advanced EF Scenarios for an MVC 5 Web app](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/advanced-entity-framework-scenarios-for-an-mvc-web-application)**

This tutorial includes introduces several topics that are useful to be aware of when you go beyond the basics of developing ASP.NET web applications that use Entity Framework Code First.

Show less