MVC Step 10 (<https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/validation>[)](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model)

Examining the Details and Delete methods

Open the Movie controller and examine the Details method:+

Copy

C#

// GET: Movies/Details/5

public async Task<IActionResult> Details(int? id)

{

if (id == null)

{

return NotFound();

}

var movie = await \_context.Movie.SingleOrDefaultAsync(m => m.ID == id);

if (movie == null)

{

return NotFound();

}

return View(movie);

}

The MVC scaffolding engine that created this action method adds a comment showing a HTTP request that invokes the method. In this case it's a GET request with three URL segments, the Movies controller, the Details method and a id value. Recall these segments are defined in Startup.+

Copy

C#

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller=Home}/{action=Index}/{id?}");

});

Code First makes it easy to search for data using the SingleOrDefaultAsync method. An important security feature built into the method is that the code verifies that the search method has found a movie before the code tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from *http://localhost:xxxx/Movies/Details/1* to something like *http://localhost:xxxx/Movies/Details/12345* (or some other value that doesn't represent an actual movie). If you did not check for a null movie, the app would throw an exception.+

Examine the Delete and DeleteConfirmed methods.+

Copy

C#

// GET: Movies/Delete/5

public async Task<IActionResult> Delete(int? id)

{

if (id == null)

{

return NotFound();

}

var movie = await \_context.Movie.SingleOrDefaultAsync(m => m.ID == id);

if (movie == null)

{

return NotFound();

}

return View(movie);

}

// POST: Movies/Delete/5

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeleteConfirmed(int id)

{

var movie = await \_context.Movie.SingleOrDefaultAsync(m => m.ID == id);

\_context.Movie.Remove(movie);

await \_context.SaveChangesAsync();

return RedirectToAction("Index");

}

Note that the HTTP GET Delete method doesn't delete the specified movie, it returns a view of the movie where you can submit (HttpPost) the deletion. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole.+

The [HttpPost] method that deletes the data is named DeleteConfirmed to give the HTTP POST method a unique signature or name. The two method signatures are shown below:+

Copy

C#

// GET: Movies/Delete/5

public async Task<IActionResult> Delete(int? id)

// POST: Movies/Delete/

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeleteConfirmed(int id)

The common language runtime (CLR) requires overloaded methods to have a unique parameter signature (same method name but different list of parameters). However, here you need two Delete methods -- one for GET and one for POST -- that both have the same parameter signature. (They both need to accept a single integer as a parameter.)+

There are two approaches to this problem, one is to give the methods different names. That's what the scaffolding mechanism did in the preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to find that method. The solution is what you see in the example, which is to add the ActionName("Delete") attribute to the DeleteConfirmed method. That attribute performs mapping for the routing system so that a URL that includes /Delete/ for a POST request will find the DeleteConfirmed method.+

Another common work around for methods that have identical names and signatures is to artificially change the signature of the POST method to include an extra (unused) parameter. That's what we did in a previous post when we added the notUsed parameter. You could do the same thing here for the [HttpPost] Delete method:3

Copy

C#

// POST: Movies/Delete/6

[ValidateAntiForgeryToken]

public async Task<IActionResult> Delete(int id, bool notUsed)

{

var movie = await \_context.Movie.SingleOrDefaultAsync(m => m.ID == id);

\_context.Movie.Remove(movie);

await \_context.SaveChangesAsync();

return RedirectToAction("Index");

}