



# Creating Form Applications

The previous chapters have focused on individual features that deal with one aspect of HTML forms, and it can sometimes be difficult to see how they fit together to perform common tasks. In this chapter, I go through the process of creating controllers, views, and Razor Pages that support an application with create, read, update, and delete (CRUD) functionality. There are no new features described in this chapter, and the objective is to demonstrate how features such as tag helpers, model binding, and model validation can be used in conjunction with Entity Framework Core.

## Preparing for This Chapter

This chapter uses the WebApp project from Chapter 30. To prepare for this chapter, replace the contents of the `HomeController.cs` file in the `Controllers` folder with those shown in Listing 31-1.

---

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

---

**Listing 31-1.** The Contents of the `HomeController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    [ValidateAntiForgeryToken]
    public class HomeController : Controller {
        private DataContext context;

        private IEnumerable<Category> Categories => context.Categories;
        private IEnumerable<Supplier> Suppliers => context.Suppliers;

        public HomeController(DataContext data) {
            context = data;
        }

        public IActionResult Index() {
            return View(context.Products.
                Include(p => p.Category).Include(p => p.Supplier));
        }
    }
}
```

Create the Views/HomeIndex.cshtml,

**Listing 31-2.** The Contents of the Index.cshtml File in the Views/Home Folder

```
@model IEnumerable<Product>
@{ Layout = "_SimpleLayout"; }

<h4 class="bg-primary text-white text-center p-2">Products</h4>
<table class="table table-sm table-bordered table-striped">
    <thead>
        <tr>
            <th>ID</th><th>Name</th><th>Price</th><th>Category</th><th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (Product p in Model) {
            <tr>
                <td>@p.ProductId</td>
                <td>@p.Name</td>
                <td>@p.Price</td>
                <td>@p.Category.Name</td>
                <td class="text-center">
                    <a asp-action="Details" asp-route-id="@p.ProductId"
                      class="btn btn-sm btn-info">Details</a>
                    <a asp-action="Edit" asp-route-id="@p.ProductId"
                      class="btn btn-sm btn-warning">Edit</a>
                    <a asp-action="Delete" asp-route-id="@p.ProductId"
                      class="btn btn-sm btn-danger">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
<a asp-action="Create" class="btn btn-primary">Create</a>
```

Next, update the Product class as shown in Listing 31-3 to change the validation constraints to remove the model-level checking and disable remote validation.

**Listing 31-3.** Changing Validation in the Product.cs File in the Models Folder

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using WebApp.Validation;
using Microsoft.AspNetCore.Mvc;

namespace WebApp.Models {

    //[PhraseAndPrice(Phrase = "Small", Price = "100")]
    public class Product {

        public long ProductId { get; set; }

        [Required]
        [Display(Name = "Name")]
        public string Name { get; set; }
    }
}
```

```

[Column(TypeName = "decimal(8, 2)")]
[Required(ErrorMessage = "Please enter a price")]
[Range(1, 999999, ErrorMessage = "Please enter a positive price")]
public decimal Price { get; set; }

[PrimaryKey(ContextType = typeof(DataContext),
    DataType = typeof(Category))]
//[Remote("CategoryKey", "Validation",
//    ErrorMessage = "Enter an existing key")]
public long CategoryId { get; set; }
public Category Category { get; set; }

[PrimaryKey(ContextType = typeof(DataContext),
    DataType = typeof(Category))]
//[Remote("SupplierKey", "Validation",
//    ErrorMessage = "Enter an existing key")]
public long SupplierId { get; set; }
public Supplier Supplier { get; set; }
}
}

```

Finally, disable the global filters in the Startup class, as shown in Listing 31-4.

**Listing 31-4.** Disabling Filters in the Startup.cs File in the WebApp Folder

```

...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<DataContext>(opts => {
        opts.UseSqlServer(Configuration[
            "ConnectionStrings:ProductConnection"]);
        opts.EnableSensitiveDataLogging(true);
    });
    services.AddControllersWithViews().AddRazorRuntimeCompilation();
    services.AddRazorPages().AddRazorRuntimeCompilation();
    services.AddSingleton<CitiesData>();

    services.Configure<AntiforgeryOptions>(opts => {
        opts.HeaderName = "X-XSRF-TOKEN";
    });

    services.Configure<MvcOptions>(opts => opts.ModelBindingMessageProvider
        .SetValueMustBeNullAccessor(value => "Please enter a value"));

    services.AddScoped<GuidResponseAttribute>();
    //services.Configure<MvcOptions>(opts => {
    //    opts.Filters.Add<HttpsOnlyAttribute>();
    //    opts.Filters.Add(new MessageAttribute(
    //        "This is the globally-scoped filter"));
    //});
}
...

```

## Dropping the Database

Open a new PowerShell command prompt, navigate to the folder that contains the WebApp.csproj file, and run the command shown in Listing 31-5 to drop the database.

**Listing 31-5.** Dropping the Database

```
dotnet ef database drop --force
```

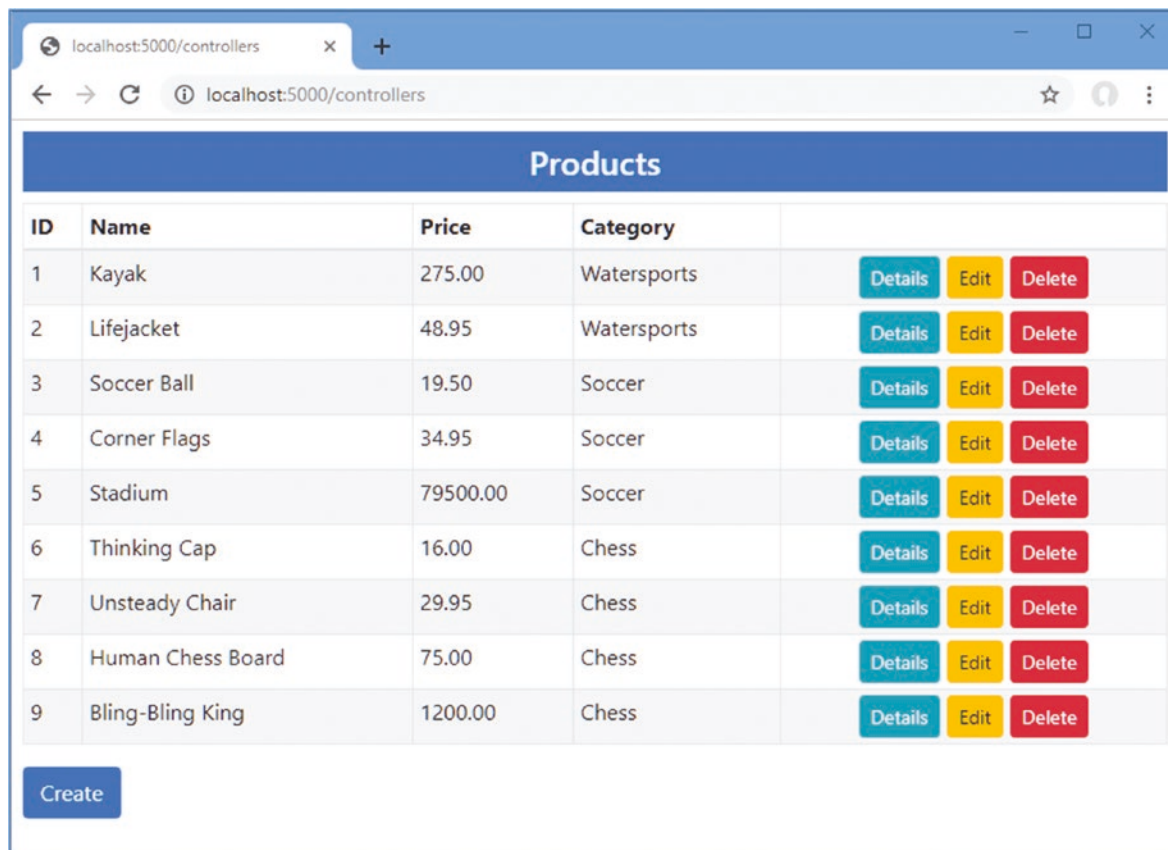
## Running the Example Application

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 31-6.

**Listing 31-6.** Running the Example Application

```
dotnet run
```

Use a browser to request <http://localhost:5000/controllers>, which will display a list of products, as shown in Figure 31-1. There are anchor elements styled to appear as buttons, but these will not work until later when I add the features to create, edit, and delete objects.



ID	Name	Price	Category	
1	Kayak	275.00	Watersports	Details Edit Delete
2	Lifejacket	48.95	Watersports	Details Edit Delete
3	Soccer Ball	19.50	Soccer	Details Edit Delete
4	Corner Flags	34.95	Soccer	Details Edit Delete
5	Stadium	79500.00	Soccer	Details Edit Delete
6	Thinking Cap	16.00	Chess	Details Edit Delete
7	Unsteady Chair	29.95	Chess	Details Edit Delete
8	Human Chess Board	75.00	Chess	Details Edit Delete
9	Bling-Bling King	1200.00	Chess	Details Edit Delete

Create

**Figure 31-1.** Running the example application

## Creating an MVC Forms Application

In the sections that follow, I show you how to perform the core data operations using MVC controllers and views. Later in the chapter, I create the same functionality using Razor Pages.

### Preparing the View Model and the View

I am going to define a single form that will be used for multiple operations, configured through its view model class. To create the view model class, add a Class File named `ProductViewModel.cs` to the `Models` folder and add the code shown in Listing 31-7.

**Listing 31-7.** The Contents of the `ProductViewModel.cs` File in the `Models` Folder

```
using System.Collections.Generic;
using System.Linq;

namespace WebApp.Models {

    public class ProductViewModel {
        public Product Product { get; set; }
        public string Action { get; set; } = "Create";
        public bool ReadOnly { get; set; } = false;
        public string Theme { get; set; } = "primary";
        public bool ShowAction { get; set; } = true;
        public IEnumerable<Category> Categories { get; set; }
            = Enumerable.Empty<Category>();
        public IEnumerable<Supplier> Suppliers { get; set; }
            = Enumerable.Empty<Supplier>();
    }
}
```

This class will allow the controller to pass data and display settings to its view. The `Product` property provides the data to display, and the `Categories` and `Suppliers` properties provide access to the `Category` and `Suppliers` objects when they are required. The other properties configure aspects of how the content is presented to the user: the `Action` property specifies the name of the action method for the current task, the `ReadOnly` property specifies whether the user can edit the data, the `Theme` property specifies the Bootstrap theme for the content, and the `ShowAction` property is used to control the visibility of the button that submits the form.

To create the view that will allow the user to interact with the application's data, add a Razor View named `ProductEditor.cshtml` to the `Views/Home` folder with the content shown in Listing 31-8.

**Listing 31-8.** The Contents of the `ProductEditor.cshtml` File in the `Views/Home` Folder

```
@model ProductViewModel
@{ Layout = "_SimpleLayout"; }

<partial name="_Validation" />

<h5 class="bg-@Model.Theme text-white text-center p-2">@Model.Action</h5>

<form asp-action="@Model.Action" method="post">
    <div class="form-group">
        <label asp-for="Product.ProductId"></label>
        <input class="form-control" asp-for="Product.ProductId" readonly />
    </div>
    <div class="form-group">
        <label asp-for="Product.Name"></label>
```

```

        <div>
            <span asp-validation-for="Product.Name" class="text-danger"></span>
        </div>
        <input class="form-control" asp-for="Product.Name"
            readonly="@Model.ReadOnly" />
    </div>
    <div class="form-group">
        <label asp-for="Product.Price"></label>
        <div>
            <span asp-validation-for="Product.Price" class="text-danger"></span>
        </div>
        <input class="form-control" asp-for="Product.Price"
            readonly="@Model.ReadOnly" />
    </div>
    <div class="form-group">
        <label asp-for="Product.CategoryId">Category</label>
        <div>
            <span asp-validation-for="Product.CategoryId" class="text-danger"></span>
        </div>
        <select asp-for="Product.CategoryId" class="form-control"
            disabled="@Model.ReadOnly"
            asp-items="@((new SelectList(Model.Categories,
                "CategoryId", "Name")))">
            <option value="" disabled selected>Choose a Category</option>
        </select>
    </div>
    <div class="form-group">
        <label asp-for="Product.SupplierId">Supplier</label>
        <div>
            <span asp-validation-for="Product.SupplierId" class="text-danger"></span>
        </div>
        <select asp-for="Product.SupplierId" class="form-control"
            disabled="@Model.ReadOnly"
            asp-items="@((new SelectList(Model.Suppliers,
                "SupplierId", "Name")))">
            <option value="" disabled selected>Choose a Supplier</option>
        </select>
    </div>
    @if (Model.ShowAction) {
        <button class="btn btn-@Model.Theme" type="submit">@Model.Action</button>
    }
    <a class="btn btn-secondary" asp-action="Index">Back</a>
</form>

```

This view can look complicated, but it combines only the features you have seen in earlier chapters and will become clearer once you see it in action. The model for this view is a `ProductViewModel` object, which provides both the data that is displayed to the user and some direction about how that data should be presented.

For each of the properties defined by the `Product` class, the view contains a set of elements: a `label` element that describes the property, an `input` or `select` element that allows the value to be edited, and a `span` element that will display validation messages. Each of the elements is configured with the `asp-for` attribute, which ensures tag helpers will transform the elements for each property. There are `div` elements to define the view structure, and all the elements are members of Bootstrap CSS classes to style the form.

## Reading Data

The simplest operation is reading data from the database and presenting it to the user. In most applications, this will allow the user to see additional details that are not present in the list view. Each task performed by the application will require a different set of `ProductViewModel` properties. To manage these combinations, add a class file named `ViewModelFactory.cs` to the `Models` folder with the code shown in Listing 31-9.

**Listing 31-9.** The Contents of the ViewModelFactory.cs File in the Models Folder

```

using System.Collections.Generic;
using System.Linq;

namespace WebApp.Models {

    public static class ViewModelFactory {

        public static ProductViewModel Details(Product p) {
            return new ProductViewModel {
                Product = p, Action = "Details",
                ReadOnly = true, Theme = "info", ShowAction = false,
                Categories = p == null ? Enumerable.Empty<Category>()
                    : new List<Category> { p.Category },
                Suppliers = p == null ? Enumerable.Empty<Supplier>()
                    : new List<Supplier> { p.Supplier },
            };
        }
    }
}

```

The Details method produces a ProductViewModel object configured for viewing an object. When the user views the details, the category and supplier details will be read-only, which means that I need to provide only the current category and supplier information.

Next, add an action method to the Home controller that uses the ViewModelFactory.Details method to create a ProductViewModel object and display it to the user with the ProductEditor view, as shown in Listing 31-10.

**Listing 31-10.** Adding an Action Method in the HomeController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    [ValidateAntiForgeryToken]
    public class HomeController : Controller {
        private DataContext context;

        private IEnumerable<Category> Categories => context.Categories;
        private IEnumerable<Supplier> Suppliers => context.Suppliers;

        public HomeController(DataContext data) {
            context = data;
        }

        public IActionResult Index() {
            return View(context.Products.
                Include(p => p.Category).Include(p => p.Supplier));
        }

        public async Task<IActionResult> Details(long id) {
            Product p = await context.Products.
                Include(p => p.Category).Include(p => p.Supplier)
                .FirstOrDefaultAsync(p => p.ProductId == id);

```

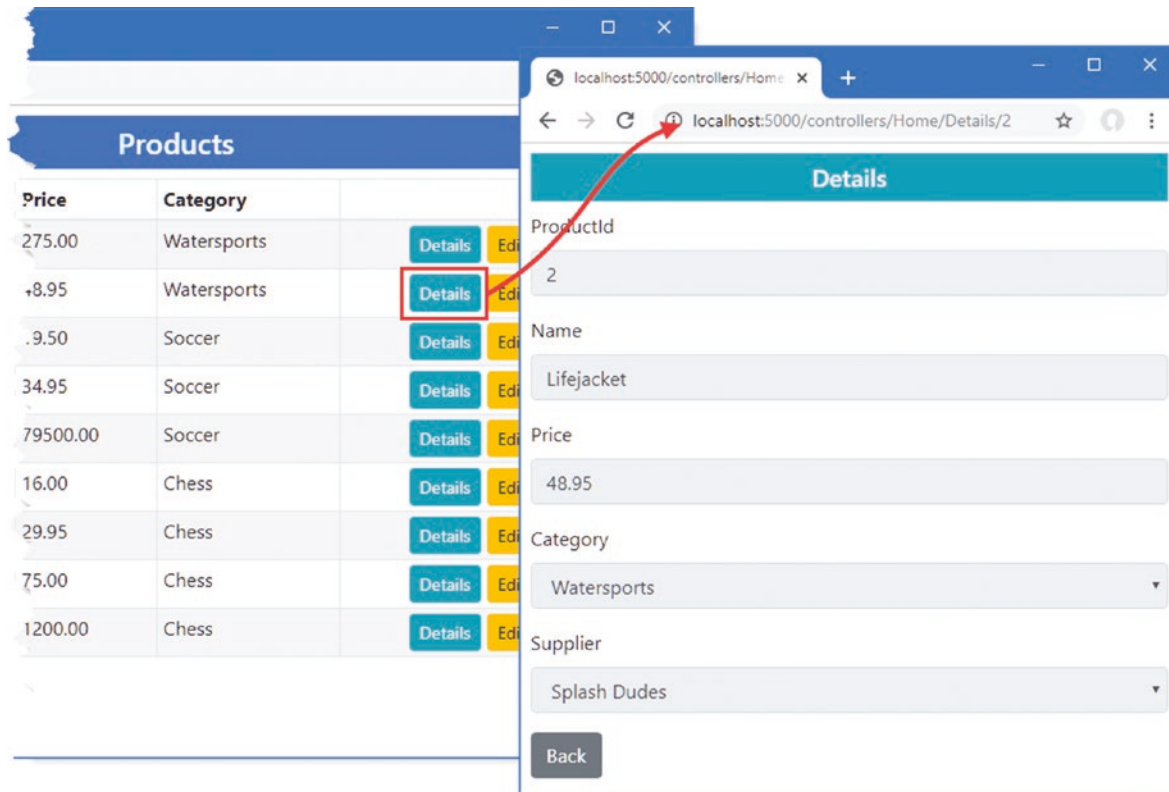
```

        ProductViewModel model = ViewModelFactory.Details(p);
        return View("ProductEditor", model);
    }
}

```

The action method uses the `id` parameter, which will be model bound from the routing data, to query the database and passes the `Product` object to the `ViewModelFactory.Details` method. Most of the operations are going to require the `Category` and `Supplier` data, so I have added properties that provide direct access to the data.

To test the details feature, restart ASP.NET Core and request `http://localhost:5000/controllers`. Click one of the `Details` buttons, and you will see the selected object presented in read-only form using the `ProductEditor` view, as shown in Figure 31-2.



**Figure 31-2.** Viewing data

If the user navigates to a URL that doesn't correspond to an object in the database, such as `http://localhost:5000/controllers/Home/Details/100`, for example, then an empty form will be displayed.

## Creating Data

Creating data relies on model binding to get the form data from the request and relies on validation to ensure the data can be stored in the database. The first step is to add a factory method that will create the view model object for creating data, as shown in Listing 31-11.

**Listing 31-11.** Adding a Method in the `ViewModelFactory.cs` File in the Models Folder

```

using System.Collections.Generic;
using System.Linq;

namespace WebApp.Models {

```



```

public static class ViewModelFactory {

    public static ProductViewModel Details(Product p) {
        return new ProductViewModel {
            Product = p, Action = "Details",
            ReadOnly = true, Theme = "info", ShowAction = false,
            Categories = p == null ? Enumerable.Empty<Category>()
                : new List<Category> { p.Category },
            Suppliers = p == null ? Enumerable.Empty<Supplier>()
                : new List<Supplier> { p.Supplier },
        };
    }

    public static ProductViewModel Create(Product product,
        IEnumerable<Category> categories, IEnumerable<Supplier> suppliers) {
        return new ProductViewModel {
            Product = product, Categories = categories, Suppliers = suppliers
        };
    }
}

```

The defaults I used for the ProductViewModel properties were set for creating data, so the Create method in Listing 31-11 sets only the Product, Categories, and Suppliers properties. Listing 31-12 adds the action methods that will create data to the Home controller.

**Listing 31-12.** Adding Actions in the HomeController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class HomeController : Controller {
        private DataContext context;

        private IEnumerable<Category> Categories => context.Categories;
        private IEnumerable<Supplier> Suppliers => context.Suppliers;

        public HomeController(DataContext data) {
            context = data;
        }

        public IActionResult Index() {
            return View(context.Products.
                Include(p => p.Category).Include(p => p.Supplier));
        }

        public async Task<IActionResult> Details(long id) {
            Product p = await context.Products.
                Include(p => p.Category).Include(p => p.Supplier)
                .FirstOrDefaultAsync(p => p.ProductId == id);
            ProductViewModel model = ViewModelFactory.Details(p);
            return View("ProductEditor", model);
        }
    }
}

```

```

    public IActionResult Create() {
        return View("ProductEditor",
            ViewModelFactory.Create(new Product(), Categories, Suppliers));
    }

    [HttpPost]
    public async Task<IActionResult> Create([FromForm] Product product) {
        if (ModelState.IsValid) {
            product.ProductId = default;
            product.Category = default;
            product.Supplier = default;
            context.Products.Add(product);
            await context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        return View("ProductEditor",
            ViewModelFactory.Create(product, Categories, Suppliers));
    }
}

```

There are two `Create` methods, which are differentiated by the `HttpPost` attribute and method parameters. HTTP GET requests will be handled by the first method, which selects the `ProductEditor` view and provides it with a `ProductViewModel` object. When the user submits the form, it will be received by the second method, which relies on model binding to receive the data and model validation to ensure the data is valid.

If the data passes validation, then I prepare the object for storage in the database by resetting three properties, like this:

```

...
product.ProductId = default;
product.Category = default;
product.Supplier = default;
...

```

Entity Framework Core configures the database so that primary keys are allocated by the database server when new data is stored. If you attempt to store an object and provide a `ProductId` value other than zero, then an exception will be thrown.

I reset the `Category` and `Supplier` properties to prevent Entity Framework Core from trying to deal with related data when storing an object. Entity Framework Core is capable of processing related data, but it can produce unexpected outcomes. (I show you how to create related data in the “Creating New Related Data Objects” section, later in this chapter.)

Notice I call the `View` method with arguments when validation fails, like this:

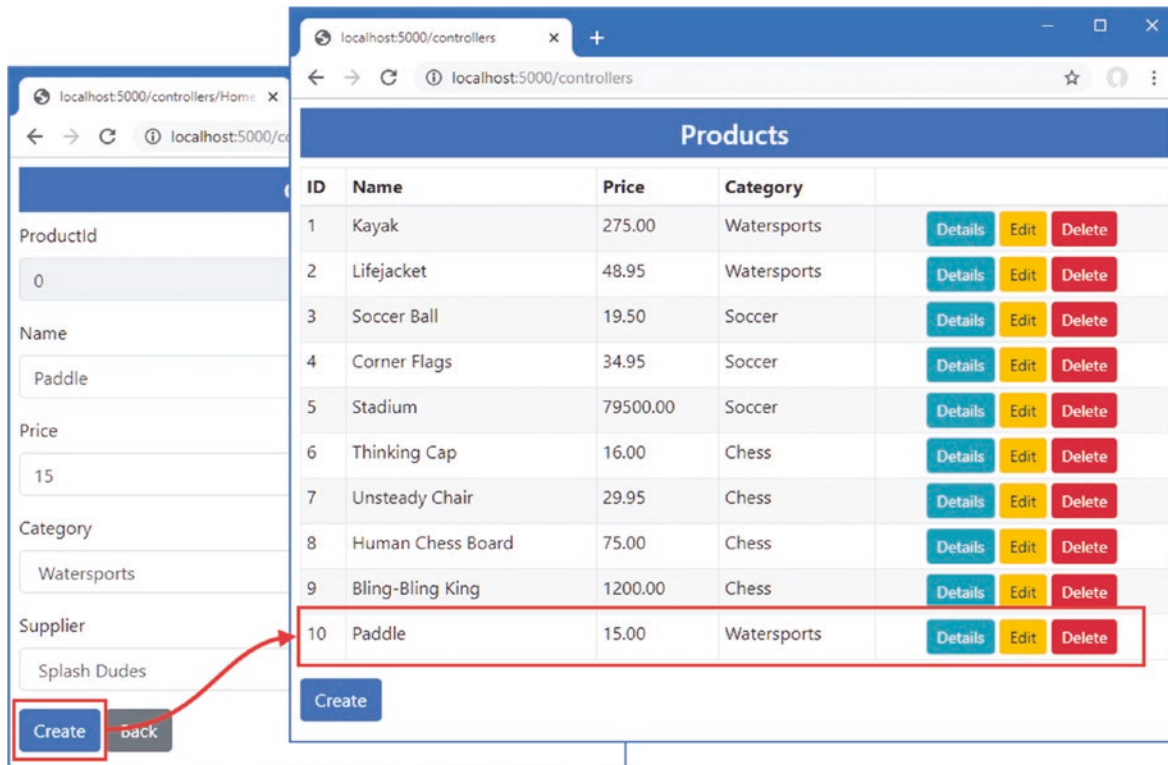
```

...
return View("ProductEditor",
    ViewModelFactory.Create(product, Categories, Suppliers));
...

```

I do this because the view model object expected by the view isn’t the same data type that I have extracted from the request using model binding. Instead, I create a new view model object that incorporates the model bound data and passes this to the `View` method.

Restart ASP.NET Core, request `http://localhost:5000/controllers`, and click `Create`. Fill out the form and click the `Create` button to submit the data. The new object will be stored in the database and displayed when the browser is redirected to the `Index` action, as shown in Figure 31-3.



**Figure 31-3.** Creating a new object

Notice that select elements allow the user to select the values for the `CategoryId` and `SupplierId` properties, using the category and supplier names, like this:

```
...
<select asp-for="Product.SupplierId" class="form-control" disabled="@Model.ReadOnly"
    asp-items="@((new SelectList(Model.Suppliers, "SupplierId", "Name")))">
    <option value="" disabled selected>Choose a Supplier</option>
</select>
...
```

In Chapter 30, I used input elements to allow the value of these properties to be set directly, but that was because I wanted to demonstrate different types of validation. In real applications, it is a good idea to provide the user with restricted choices when the application already has the data it expects the user to choose from. Making the user enter a valid primary key, for example, makes no sense in a real project because the application can easily provide the user with a list of those keys to choose from, as shown in Figure 31-4.

---

■ **Tip** I show you different techniques for creating related data in the “Creating New Related Data Objects” section.

---

**Figure 31-4.** Presenting the user with a choice

## Editing Data

The process for editing data is similar to creating data. The first step is to add a new method to the view model factory that will configure the way the data is presented to the user, as shown in Listing 31-13.

**Listing 31-13.** Adding a Method in the ViewModelFactory.cs File in the Models Folder

```
using System.Collections.Generic;
using System.Linq;

namespace WebApp.Models {

    public static class ViewModelFactory {

        public static ProductViewModel Details(Product p) {
            return new ProductViewModel {
                Product = p, Action = "Details",
                ReadOnly = true, Theme = "info", ShowAction = false,
                Categories = p == null ? Enumerable.Empty<Category>()
                    : new List<Category> { p.Category },
                Suppliers = p == null ? Enumerable.Empty<Supplier>()
                    : new List<Supplier> { p.Supplier },
            };
        }

        public static ProductViewModel Create(Product product,
            IEnumerable<Category> categories, IEnumerable<Supplier> suppliers) {
            return new ProductViewModel {
                Product = product, Categories = categories, Suppliers = suppliers
            };
        }

        public static ProductViewModel Edit(Product product,
            IEnumerable<Category> categories, IEnumerable<Supplier> suppliers) {
            return new ProductViewModel {
                Product = product, Categories = categories, Suppliers = suppliers,
                Theme = "warning", Action = "Edit"
            };
        }
    }
}
```

The next step is to add the action methods to the Home controller that will display the current properties of a Product object to the user and receive the changes the user makes, as shown in Listing 31-14.

**Listing 31-14.** Adding Action Methods in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    [AutoValidateAntiforgeryToken]
    public class HomeController : Controller {
        private DataContext context;

        private IEnumerable<Category> Categories => context.Categories;
        private IEnumerable<Supplier> Suppliers => context.Suppliers;

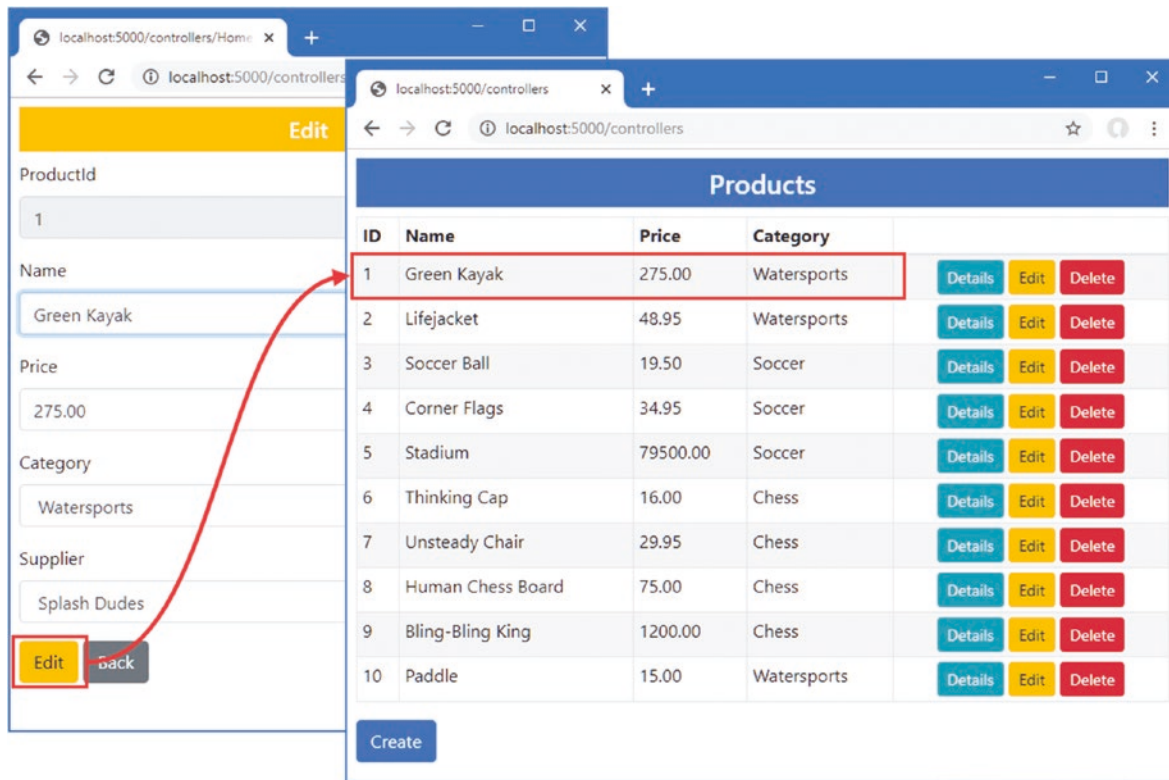
        public HomeController(DataContext data) {
            context = data;
        }

        // ...other action methods omitted for brevity...

        public async Task<IActionResult> Edit(long id) {
            Product p = await context.Products.FindAsync(id);
            ProductViewModel model = ViewModelFactory.Edit(p, Categories, Suppliers);
            return View("ProductEditor", model);
        }

        [HttpPost]
        public async Task<IActionResult> Edit([FromForm]Product product) {
            if (ModelState.IsValid) {
                product.Category = default;
                product.Supplier = default;
                context.Products.Update(product);
                await context.SaveChangesAsync();
                return RedirectToAction(nameof(Index));
            }
            return View("ProductEditor",
                ViewModelFactory.Edit(product, Categories, Suppliers));
        }
    }
}
```

To see the editing feature at work, restart ASP.NET Core, navigate to <http://localhost:5000/controllers>, and click one of the Edit buttons. Change one or more property values and submit the form. The changes will be stored in the database and reflected in the list displayed when the browser is redirected to the Index action, as shown in Figure 31-5.



**Figure 31-5.** Editing a product

Notice that the `ProductId` property cannot be changed. Attempting to change the primary key of an object should be avoided because it interferes with the Entity Framework Core understanding of the identity of its objects. If you can't avoid changing the primary key, then the safest approach is to delete the existing object and store a new one.

## Deleting Data

The final basic operation is removing objects from the database. By now the pattern will be clear, and the first step is to add a method to create a view model object to determine how the data is presented to the user, as shown in Listing 31-15.

**Listing 31-15.** Adding a Method in the `ViewModelFactory.cs` File in the Models Folder

```
using System.Collections.Generic;
using System.Linq;

namespace WebApp.Models {

    public static class ViewModelFactory {

        // ...other methods omitted for brevity...

        public static ProductViewModel Delete(Product p,
            IEnumerable<Category> categories, IEnumerable<Supplier> suppliers) {
            return new ProductViewModel {
                Product = p, Action = "Delete",
                ReadOnly = true, Theme = "danger",
            };
        }
    }
}
```

```

        Categories = categories, Suppliers = suppliers
    };
}
}
}

```

Listing 31-16 adds the action methods to the Home controller that will respond to the GET request by displaying the selected object and the POST request to remove that object from the database.

**Listing 31-16.** Adding Action Methods in the HomeController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    [ValidateAntiForgeryToken]
    public class HomeController : Controller {
        private DataContext context;

        private IEnumerable<Category> Categories => context.Categories;
        private IEnumerable<Supplier> Suppliers => context.Suppliers;

        public HomeController(DataContext data) {
            context = data;
        }

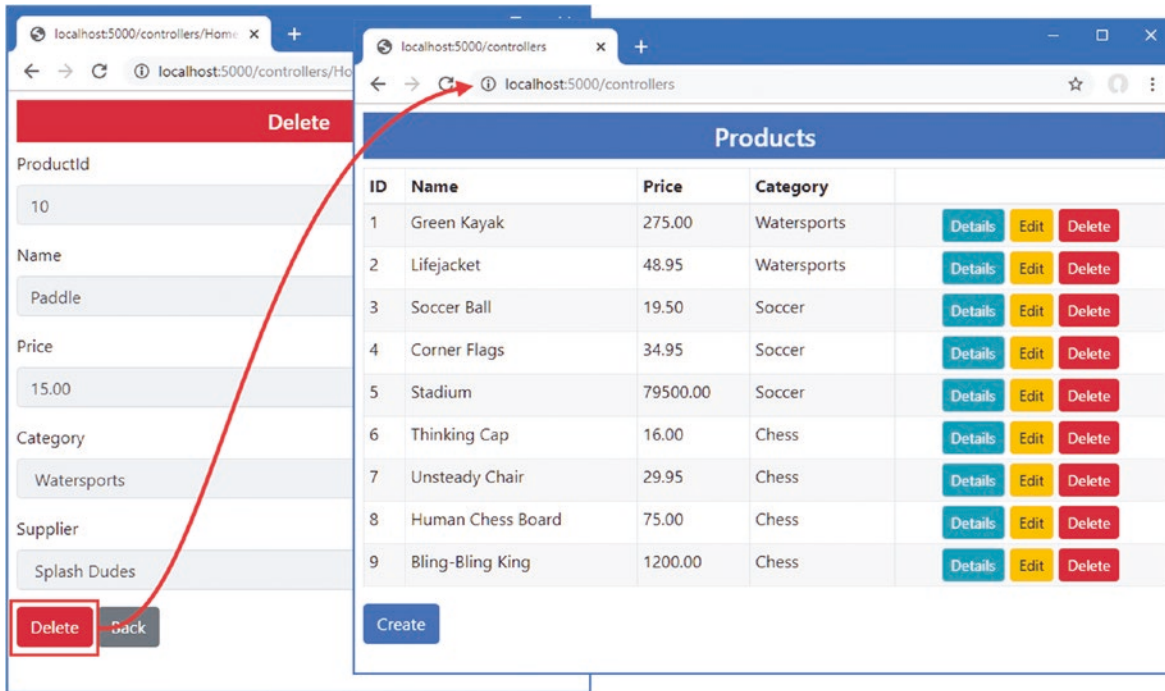
        // ...other action methods removed for brevity...

        public async Task<IActionResult> Delete(long id) {
            ProductViewModel model = ViewModelFactory.Delete(
                await context.Products.FindAsync(id), Categories, Suppliers);
            return View("ProductEditor", model);
        }

        [HttpPost]
        public async Task<IActionResult> Delete(Product product) {
            context.Products.Remove(product);
            await context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
    }
}

```

The model binding process creates a Product object from the form data, which is passed to Entity Framework Core to remove from the database. Once the data has been removed from the database, the browser is redirected to the Index action, as shown in Figure 31-6.



**Figure 31-6.** Deleting data

## Creating a Razor Pages Forms Application

Working with Razor Forms relies on similar techniques as the controller examples, albeit broken up into smaller chunks of functionality. As you will see, the main difficulty is preserving the modular nature of Razor Pages without duplicating code and markup. The first step is to create the Razor Page that will display the list of Product objects and provide the links to the other operations. Add a Razor Page named `Index.cshtml` to the Pages folder with the content shown in Listing 31-17.

**Listing 31-17.** The Contents of the `Index.cshtml` File in the Pages Folder

```
@page "/pages/{id:long?}"
@model IndexModel
@using Microsoft.AspNetCore.Mvc.RazorPages
@using Microsoft.EntityFrameworkCore

<div class="m-2">
    <h4 class="bg-primary text-white text-center p-2">Products</h4>
    <table class="table table-sm table-bordered table-striped">
        <thead>
            <tr>
                <th>ID</th><th>Name</th><th>Price</th><th>Category</th><th></th>
            </tr>
        </thead>
        <tbody>
            @foreach (Product p in Model.Products) {
```



```

        <tr>
            <td>@p.ProductId</td>
            <td>@p.Name</td>
            <td>@p.Price</td>
            <td>@p.Category.Name</td>
            <td class="text-center">
                <a asp-page="Details" asp-route-id="@p.ProductId"
                    class="btn btn-sm btn-info">Details</a>
                <a asp-page="Edit" asp-route-id="@p.ProductId"
                    class="btn btn-sm btn-warning">Edit</a>
                <a asp-page="Delete" asp-route-id="@p.ProductId"
                    class="btn btn-sm btn-danger">Delete</a>
            </td>
        </tr>
    }
</tbody>
</table>
<a asp-page="Create" class="btn btn-primary">Create</a>
</div>

@functions {
    public class IndexModel: PageModel {
        private DataContext context;

        public IndexModel(DataContext dbContext) {
            context = dbContext;
        }

        public IEnumerable<Product> Products { get; set; }

        public void OnGetAsync(long id = 1) {
            Products = context.Products
                .Include(p => p.Category).Include(p => p.Supplier);
        }
    }
}

```

This view part of the page displays a table populated with the details of the `Product` objects obtained from the database by the page model. Use a browser to request `http://localhost:5000/pages`, and you will see the response shown in Figure 31-7. Alongside the details of the `Product` objects, the page displays anchor elements that navigate to other Razor Pages, which I define in the sections that follow.

The screenshot shows a web browser window with the address bar at `localhost:5000/pages/`. The page has a dark blue header with the text "Razor Page". Below this is a blue section header "Products". The main content is a table with 5 columns: ID, Name, Price, Category, and a set of action buttons. The table contains 9 rows of product data. Each row has three buttons: "Details" (blue), "Edit" (yellow), and "Delete" (red). At the bottom left of the table area is a blue "Create" button.

ID	Name	Price	Category	
1	Kayak	275.00	Watersports	Details Edit Delete
2	Lifejacket	48.95	Watersports	Details Edit Delete
3	Soccer Ball	19.50	Soccer	Details Edit Delete
4	Corner Flags	34.95	Soccer	Details Edit Delete
5	Stadium	79500.00	Soccer	Details Edit Delete
6	Thinking Cap	16.00	Chess	Details Edit Delete
7	Unsteady Chair	29.95	Chess	Details Edit Delete
8	Human Chess Board	75.00	Chess	Details Edit Delete
9	Bling-Bling King	1200.00	Chess	Details Edit Delete

Create

**Figure 31-7.** Listing data using a Razor Page

## Creating Common Functionality

I don't want to duplicate the same HTML form and supporting code in each of the pages required by the example application. Instead, I am going to define a partial view that defines the HTML form and a base class that defines the common code required by the page model classes. For the partial view, a Razor View named `_ProductEditor.cshtml` to the Pages folder with the content shown in Listing 31-18.

### USING MULTIPLE PAGE

The `asp-page-handler` attribute can be used to specify the name of a handler method, which allows a Razor Page to be used for more than one operation. I don't like this feature because the result is too close to a standard MVC controller and undermines the self-contained and modular aspects of Razor Page development that I like.

The approach I prefer is, of course, the one that I have taken in this chapter, which is to consolidate common content in partial views and a shared base class. Either approach works, and I recommend you try both to see which suits you and your project.

**Listing 31-18.** The Contents of the \_ProductEditor.cshtml File in the Pages Folder

```

@model ProductViewModel

<partial name="_Validation" />

<h5 class="bg-@Model.Theme text-white text-center p-2">@Model.Action</h5>

<form asp-page="@Model.Action" method="post">
    <div class="form-group">
        <label asp-for="Product.ProductId"></label>
        <input class="form-control" asp-for="Product.ProductId" readonly />
    </div>
    <div class="form-group">
        <label asp-for="Product.Name"></label>
        <div>
            <span asp-validation-for="Product.Name" class="text-danger"></span>
        </div>
        <input class="form-control" asp-for="Product.Name"
            readonly="@Model.ReadOnly" />
    </div>
    <div class="form-group">
        <label asp-for="Product.Price"></label>
        <div>
            <span asp-validation-for="Product.Price" class="text-danger"></span>
        </div>
        <input class="form-control" asp-for="Product.Price"
            readonly="@Model.ReadOnly" />
    </div>
    <div class="form-group">
        <label asp-for="Product.CategoryId">Category</label>
        <div>
            <span asp-validation-for="Product.CategoryId" class="text-danger"></span>
        </div>
        <select asp-for="Product.CategoryId" class="form-control"
            disabled="@Model.ReadOnly"

            asp-items="@((new SelectList(Model.Categories,
                "CategoryId", "Name")))">
            <option value="" disabled selected>Choose a Category</option>
        </select>
    </div>
    <div class="form-group">
        <label asp-for="Product.SupplierId">Supplier</label>
        <div>
            <span asp-validation-for="Product.SupplierId" class="text-danger"></span>
        </div>
        <select asp-for="Product.SupplierId" class="form-control"
            disabled="@Model.ReadOnly"
            asp-items="@((new SelectList(Model.Suppliers,
                "SupplierId", "Name")))">
            <option value="" disabled selected>Choose a Supplier</option>
        </select>
    </div>
    @if (Model.ShowAction) {
        <button class="btn btn-@Model.Theme" type="submit">@Model.Action</button>
    }
    <a class="btn btn-secondary" asp-page="Index">Back</a>
</form>

```

The partial view uses the `ProductViewModel` class as its model type and relies on the built-in tag helpers to present input and select elements for the properties defined by the `Product` class. This is the same content used earlier in the chapter, except with the `asp-action` attribute replaced with `asp-page` to specify the target for the form and anchor elements.

To define the page model base class, add a class file named `EditorPageModel.cs` to the Pages folder and use it to define the class shown in Listing 31-19.

**Listing 31-19.** The Contents of the `EditorPageModel.cs` File in the Pages Folder

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
using WebApp.Models;

namespace WebApp.Pages {

    public class EditorPageModel : PageModel {

        public EditorPageModel(DataContext dbContext) {
            DataContext = dbContext;
        }

        public DataContext DataContext { get; set; }

        public IEnumerable<Category> Categories => DataContext.Categories;
        public IEnumerable<Supplier> Suppliers => DataContext.Suppliers;

        public ProductViewModel ViewModel { get; set; }
    }
}
```

The properties defined by this class are simple, but they will help simplify the page model classes of the Razor Pages that handle each operation.

All the Razor Pages required for this example depend on the same namespaces. Add the expressions shown in Listing 31-20 to the `_ViewImports.cshtml` file in the Pages folder to avoid duplicate expressions in the individual pages.

---

■ **Tip** Make sure you alter the `_ViewImports.cshtml` file in the Pages folder and not the file with the same name in the Views folder.

---

**Listing 31-20.** Adding Namespaces in the `_ViewImports.cshtml` File in the Pages Folder

```
@namespace WebApp.Pages
@using WebApp.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, WebApp
@using Microsoft.AspNetCore.Mvc.RazorPages
@using Microsoft.EntityFrameworkCore
@using WebApp.Pages
@using System.Text.Json
@using Microsoft.AspNetCore.Http
```

## Defining Pages for the CRUD Operations

With the partial view and shared base class in place, the pages that handle individual operations are simple. Add a Razor Page named `Details.cshtml` to the Pages folder with the code and content shown in Listing 31-21.

**Listing 31-21.** The Contents of the Details.cshtml File in the Pages Folder

```

@page "/pages/details/{id}"
@model DetailsModel

<div class="m-2">
    <partial name="_ProductEditor" model="@Model.ViewModel" />
</div>

@functions {

    public class DetailsModel: EditorPageModel {

        public DetailsModel(DataContext dbContext): base(dbContext) {}

        public async Task OnGetAsync(long id) {
            Product p = await DataContext.Products.
                Include(p => p.Category).Include(p => p.Supplier)
                .FirstOrDefaultAsync(p => p.ProductId == id);
            ViewModel = ViewModelFactory.Details(p);
        }
    }
}

```

The constructor receives an Entity Framework Core context object, which it passes to the base class. The handler method responds to requests by querying the database and using the response to create a `ProductViewModel` object using the `ViewModelFactory` class.

Add a Razor Page named `Create.cshtml` to the Pages folder with the code and content shown in Listing 31-22.

---

■ **Tip** Using a partial view means that the `asp-for` attributes set element names without an additional prefix. This allows me to use the `FromForm` attribute for model binding without using the `Name` argument.

---

**Listing 31-22.** The Contents of the Create.cshtml File in the Pages Folder

```

@page "/pages/create"
@model CreateModel

<div class="m-2">
    <partial name="_ProductEditor" model="@Model.ViewModel" />
</div>

@functions {

    public class CreateModel: EditorPageModel {

        public CreateModel(DataContext dbContext): base(dbContext) {}

        public void OnGet() {
            ViewModel = ViewModelFactory.Create(new Product(),
                Categories, Suppliers);
        }
    }
}

```

```

        public async Task<IActionResult> OnPostAsync([FromForm]Product product) {
            if (ModelState.IsValid) {
                product.ProductId = default;
                product.Category = default;
                product.Supplier = default;
                DataContext.Products.Add(product);
                await DataContext.SaveChangesAsync();
                return RedirectToPage(nameof(Index));
            }
            ViewModel = ViewModelFactory.Create(product, Categories, Suppliers);
            return Page();
        }
    }
}

```

Add a Razor Page named `Edit.cshtml` to the Pages folder with the code and content shown in Listing 31-23.

**Listing 31-23.** The Contents of the `Edit.cshtml` File in the Pages Folder

```

@page "/pages/edit/{id}"
@model EditModel

<div class="m-2">
    <partial name="_ProductEditor" model="@Model.ViewModel" />
</div>

@functions {

    public class EditModel: EditorPageModel {

        public EditModel(DataContext dbContext): base(dbContext) {}

        public async Task OnGetAsync(long id) {
            Product p = await this.DataContext.Products.FindAsync(id);
            ViewModel = ViewModelFactory.Edit(p, Categories, Suppliers);
        }

        public async Task<IActionResult> OnPostAsync([FromForm]Product product) {
            if (ModelState.IsValid) {
                product.Category = default;
                product.Supplier = default;
                DataContext.Products.Update(product);
                await DataContext.SaveChangesAsync();
                return RedirectToPage(nameof(Index));
            }
            ViewModel = ViewModelFactory.Edit(product, Categories, Suppliers);
            return Page();
        }
    }
}

```

Add a Razor Page named `Delete.cshtml` to the Pages folder with the code and content shown in Listing 31-24.

**Listing 31-24.** The Contents of the Delete.cshtml File in the Pages Folder

```

@page "/pages/delete/{id}"
@model DeleteModel

<div class="m-2">
    <partial name="_ProductEditor" model="@Model.ViewModel" />
</div>

@functions {

    public class DeleteModel: EditorPageModel {

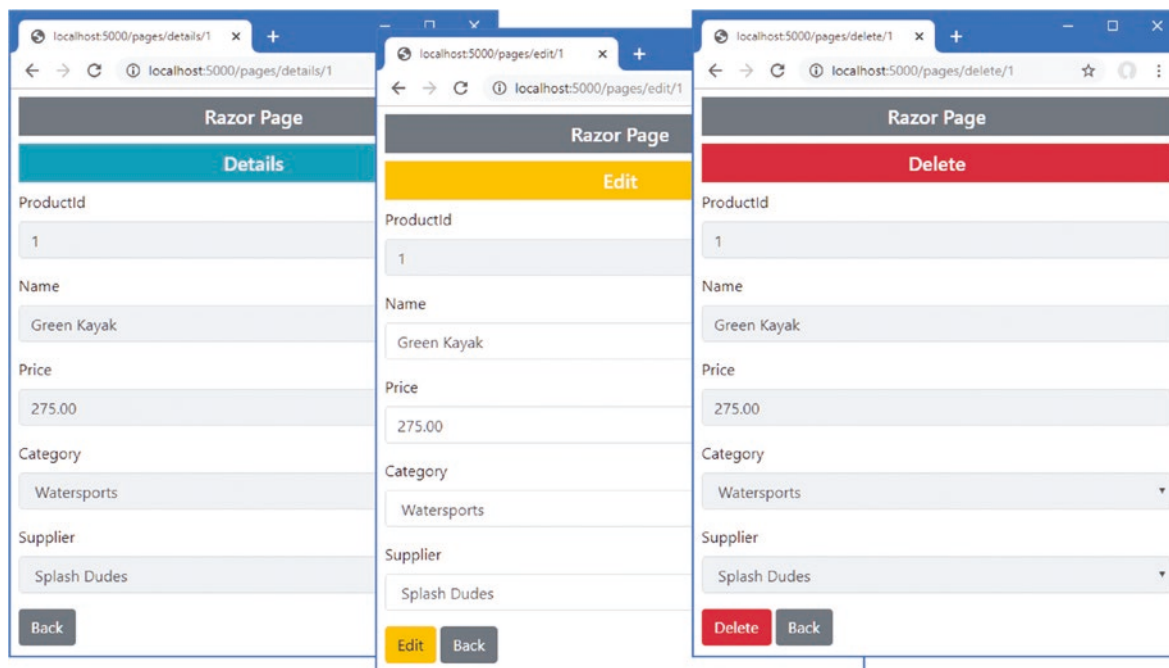
        public DeleteModel(DataContext dbContext): base(dbContext) {}

        public async Task OnGetAsync(long id) {
            ViewModel = ViewModelFactory.Delete(
                await DataContext.Products.FindAsync(id), Categories, Suppliers);
        }

        public async Task<IActionResult> OnPostAsync([FromForm]Product product) {
            DataContext.Products.Remove(product);
            await DataContext.SaveChangesAsync();
            return RedirectToPage(nameof(Index));
        }
    }
}

```

Restart ASP.NET Core and navigate to <http://localhost:5000/pages>, and you will be able to click the links to view, create, edit, and remove data, as shown in Figure 31-8.

**Figure 31-8.** Using Razor Pages

## Creating New Related Data Objects

Some applications will need to allow the user to create new related data so that, for example, a new `Category` can be created along with a `Product` in that `Category`. There are two ways to approach this problem, as described in the sections that follow.

### Providing the Related Data in the Same Request

The first approach is to ask the user to provide the data required to create the related data in the same form. For the example application, this means collecting details for a `Category` object in the same form that the user enters the values for the `Product` object.

This can be a useful approach for simple data types, where only a small amount of data is required to create the related object but is not well suited for types with many properties.

I prefer to define the HTML elements for the related data type in their own partial view. Add a Razor View named `_CategoryEditor.cshtml` to the `Pages` folder with the content shown in Listing 31-25.

**Listing 31-25.** The Contents of the `_CategoryEditor.cshtml` File in the `Pages` Folder

```
@model Product
<script type="text/javascript">
    $(document).ready(() => {
        const catGroup = $("#categoryGroup").hide();
        $("select[name='Product.CategoryId']").on("change", (event) =>
            event.target.value === "-1" ? catGroup.show() : catGroup.hide());
    });
</script>

<div class="form-group bg-info p-1" id="categoryGroup">
    <label class="text-white" asp-for="Category.Name">
        New Category Name
    </label>
    <input class="form-control" asp-for="Category.Name" value="" />
</div>
```

The `Category` type requires only one property, which the user will provide using a standard input element. The script element in the partial view contains jQuery code that hides the new elements until the user selects an option element that sets a value of `-1` for the `Product.CategoryId` property. (Using JavaScript is entirely optional, but it helps to emphasize the purpose of the new elements.)

Listing 31-26 adds the partial view to the editor, along with the option element that will display the elements for creating a new `Category` object.

**Listing 31-26.** Adding Elements in the `_ProductEditor.cshtml` File in the `Pages` Folder

```
...
<div class="form-group">
    <label asp-for="Product.CategoryId">Category</label>
    <div>
        <span asp-validation-for="Product.CategoryId" class="text-danger"></span>
    </div>
    <select asp-for="Product.CategoryId" class="form-control"
        disabled="@Model.ReadOnly" asp-items="@((new SelectList(Model.Categories,
            "CategoryId", "Name")))">
        <option value="-1">Create New Category...</option>
        <option value="" disabled selected>Choose a Category</option>
    </select>
</div>
```



```

<partial name="_CategoryEditor" for="Product" />
<div class="form-group">
    <label asp-for="Product.SupplierId">Supplier</label>
    <div><span asp-validation-for="Product.SupplierId" class="text-danger"></span></div>
    <select asp-for="Product.SupplierId" class="form-control" disabled="@Model.ReadOnly"
        asp-items="@((new SelectList(Model.Suppliers,
            "SupplierId", "Name")))">
        <option value="" disabled selected>Choose a Supplier</option>
    </select>
</div>
...

```

I need the new functionality in multiple pages, so to avoid code duplication, I have added a method that handles the related data to the page model base class, as shown in Listing 31-27.

**Listing 31-27.** Adding a Method in the EditorPageModel.cs File in the Pages Folder

```

using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
using WebApp.Models;
using System.Threading.Tasks;

namespace WebApp.Pages {

    public class EditorPageModel : PageModel {

        public EditorPageModel(DataContext dbContext) {
            DataContext = dbContext;
        }

        public DataContext DataContext { get; set; }

        public IEnumerable<Category> Categories => DataContext.Categories;
        public IEnumerable<Supplier> Suppliers => DataContext.Suppliers;

        public ProductViewModel ViewModel { get; set; }

        protected async Task CheckNewCategory(Product product) {
            if (product.CategoryId == -1
                && !string.IsNullOrEmpty(product.Category?.Name)) {
                DataContext.Categories.Add(product.Category);
                await DataContext.SaveChangesAsync();
                product.CategoryId = product.Category.CategoryId;
                ModelState.Clear();
                TryValidateModel(product);
            }
        }
    }
}

```

The new code creates a Category object using the data received from the user and stores it in the database. The database server assigns a primary key to the new object, which Entity Framework Core uses to update the Category object. This allows me to update the CategoryId property of the Product object and then re-validate the model data, knowing that the value assigned to the CategoryId property will pass validation because it corresponds to the newly allocated key. To integrate the new functionality into the Create page, add the statement shown in Listing 31-28.

**Listing 31-28.** Adding a Statement in the Create.cshtml File in the Pages Folder

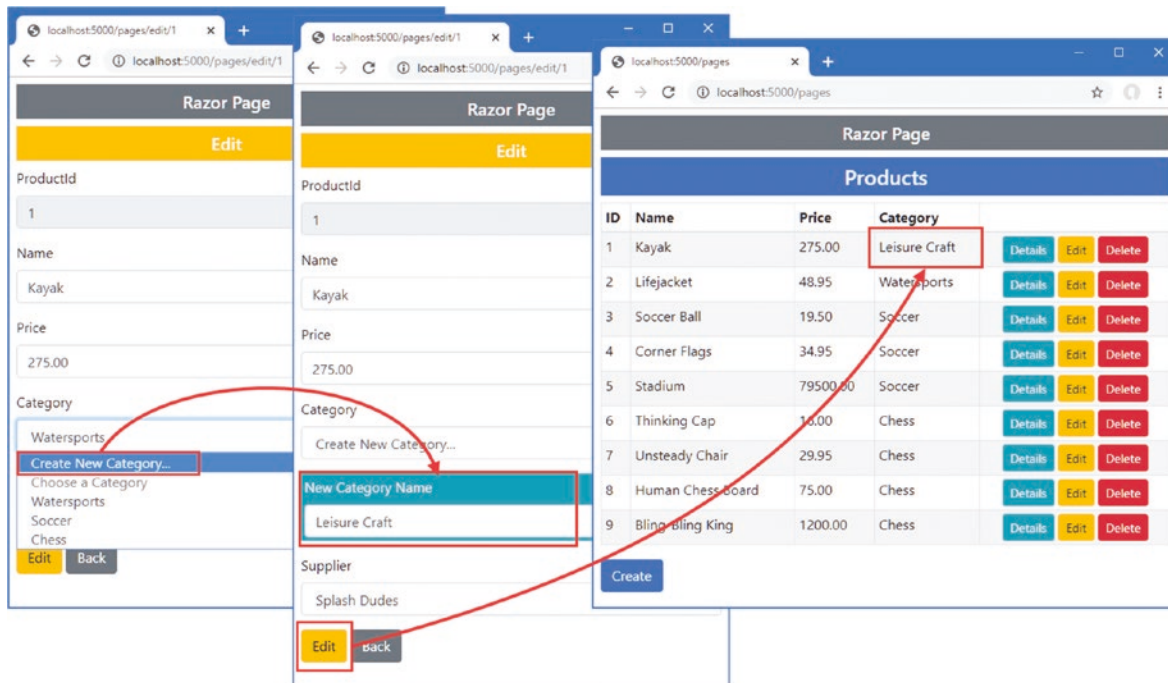
```
...
public async Task<IActionResult> OnPostAsync([FromForm]Product product) {
    await CheckNewCategory(product);
    if (ModelState.IsValid) {
        product.ProductId = default;
        product.Category = default;
        product.Supplier = default;
        DataContext.Products.Add(product);
        await DataContext.SaveChangesAsync();
        return RedirectToPage(nameof(Index));
    }
    ViewModel = ViewModelFactory.Create(product, Categories, Suppliers);
    return Page();
}
...
```

Add the same statement to the handler method in the Edit page, as shown in Listing 31-29.

**Listing 31-29.** Adding a Statement in the Edit.cshtml File in the Pages Folder

```
...
public async Task<IActionResult> OnPostAsync([FromForm]Product product) {
    await CheckNewCategory(product);
    if (ModelState.IsValid) {
        product.Category = default;
        product.Supplier = default;
        DataContext.Products.Update(product);
        await DataContext.SaveChangesAsync();
        return RedirectToPage(nameof(Index));
    }
    ViewModel = ViewModelFactory.Edit(product, Categories, Suppliers);
    return Page();
}
...
```

Restart ASP.NET Core so the page model base class is recompiled and use a browser to request <http://localhost:5000/pages/edit/1>. Click the Category select element and choose Create New Category from the list of options. Enter a new category name into the input element and click the Edit button. When the request is processed, a new Category object will be stored in the database and associated with the Product object, as shown in Figure 31-9.



**Figure 31-9.** Creating related data

## Breaking Out to Create New Data

For related data types that have their own complex creation process, adding elements to the main form can be overwhelming to the user; a better approach is to navigate away from the main form to another controller or page, let the user create the new object, and then return to complete the original task. I will demonstrate this technique for the creation of *Supplier* objects, even though the *Supplier* type is simple and requires only two values from the user.

To create a form that will let the user create *Supplier* objects, add a Razor Page named *SupplierBreakOut.cshtml* to the *Pages* folder with the content shown in Listing 31-30.

**Listing 31-30.** The Contents of the *SupplierBreakOut.cshtml* File in the *Pages* Folder

```
@page "/pages/supplier"
@model SupplierPageModel

<div class="m-2">
  <h5 class="bg-secondary text-white text-center p-2">New Supplier</h5>
  <form asp-page="SupplierBreakOut" method="post">
    <div class="form-group">
      <label asp-for="Supplier.Name"></label>
      <input class="form-control" asp-for="Supplier.Name" />
    </div>
    <div class="form-group">
      <label asp-for="Supplier.City"></label>
      <input class="form-control" asp-for="Supplier.City" />
    </div>
    <button class="btn btn-secondary" type="submit">Create</button>
    <a class="btn btn-outline-secondary"
      asp-page="@Model.ReturnPage" asp-route-id="@Model.ProductId">
      Cancel
    </a>
  </form>
</div>
```

```

@functions {

    public class SupplierPageModel: PageModel {
        private DataContext context;

        public SupplierPageModel(DataContext dbContext) {
            context = dbContext;
        }

        [BindProperty]
        public Supplier Supplier { get; set; }

        public string ReturnPage { get; set; }
        public string ProductId { get; set; }

        public void OnGet([FromQuery(Name="Product")] Product product,
            string returnPage) {
            TempData["product"] = Serialize(product);
            TempData["returnAction"] = ReturnPage = returnPage;
            TempData["productId"] = ProductId = product.ProductId.ToString();
        }

        public async Task<IActionResult> OnPostAsync() {
            context.Suppliers.Add(Supplier);
            await context.SaveChangesAsync();
            Product product = Deserialize(TempData["product"] as string);
            product.SupplierId = Supplier.SupplierId;
            TempData["product"] = Serialize(product);
            string id = TempData["productId"] as string;
            return RedirectToPage(TempData["returnAction"] as string,
                new { id = id });
        }

        private string Serialize(Product p) => JsonSerializer.Serialize(p);
        private Product Deserialize(string json) =>
            JsonSerializer.Deserialize<Product>(json);
    }
}

```

The user will navigate to this page using a GET request that will contain the details of the Product the user has provided and the name of the page that the user should be returned to. This data is stored using the temp data feature.

This page presents the user with a form containing fields for the Name and City properties required to create a new Supplier object. When the form is submitted, the POST handler method stores a new Supplier object and uses the key assigned by the database server to update the Product object, which is then stored as temp data again. The user is redirected back to the page from which they arrived.

Listing 31-31 adds elements to the `_ProductEditor` partial view that will allow the user to navigate to the new page.

**Listing 31-31.** Adding Elements in the `_ProductEditor.cshtml` File in the Pages Folder

```

...
<partial name="_CategoryEditor" for="Product" />

<div class="form-group">
    <label asp-for="Product.SupplierId">
        Supplier
        @if (!Model.ReadOnly) {
            <input type="hidden" name="returnPage" value="@Model.Action" />
            <button class="btn btn-sm btn-outline-primary m1-3"

```

```

        asp-page="SupplierBreakOut" formmethod="get" formnovalidate>
        Create New Supplier
    </button>
}
</label>
<div>
    <span asp-validation-for="Product.SupplierId" class="text-danger"></span>
</div>
<select asp-for="Product.SupplierId" class="form-control"
    disabled="@Model.ReadOnly" asp-items="@((new SelectList(Model.Suppliers,
        "SupplierId", "Name")))">
    <option value="" disabled selected>Choose a Supplier</option>
</select>
</div>
...

```

The new elements add a hidden input element that captures the page to return to and a button element that submits the form data to the SupplierBreakOut page using a GET request, which means the form values will be encoded in the query string (and is the reason I used the FromQuery attribute in Listing 31-30). Listing 31-32 shows the change required to the Create page to add support for retrieving the temp data and using it to populate the Product form.

**Listing 31-32.** Retrieving Data in the Create.cshtml File in the Pages Folder

```

@page "/pages/create"
@model CreateModel

<div class="m-2">
    <partial name="_ProductEditor" model="@Model.ViewModel" />
</div>

@functions {

    public class CreateModel: EditorPageModel {

        public CreateModel(DataContext dbContext): base(dbContext) {}

        public void OnGet() {
            Product p = TempData.ContainsKey("product")
            ? JsonSerializer.Deserialize<Product>(TempData["product"] as string)
            : new Product();
            ViewModel = ViewModelFactory.Create(p, Categories, Suppliers);
        }

        public async Task<IActionResult> OnPostAsync([FromForm]Product product) {
            await CheckNewCategory(product);
            if (ModelState.IsValid) {
                product.ProductId = default;
                product.Category = default;
                product.Supplier = default;
                DataContext.Products.Add(product);
                await DataContext.SaveChangesAsync();
                return RedirectToPage(nameof(Index));
            }
            ViewModel = ViewModelFactory.Create(product, Categories, Suppliers);
            return Page();
        }
    }
}

```

A similar change is required in the Edit page, as shown in Listing 31-33. (The other pages do not require a change since the breakout is required only when the user is able to create or edit Product data.)

**Listing 31-33.** Retrieving Data in the Edit.cshtml File in the Pages Folder

```
@page "/pages/edit/{id}"
@model EditModel

<div class="m-2">
    <partial name="_ProductEditor" model="@Model.ViewModel" />
</div>

@functions {

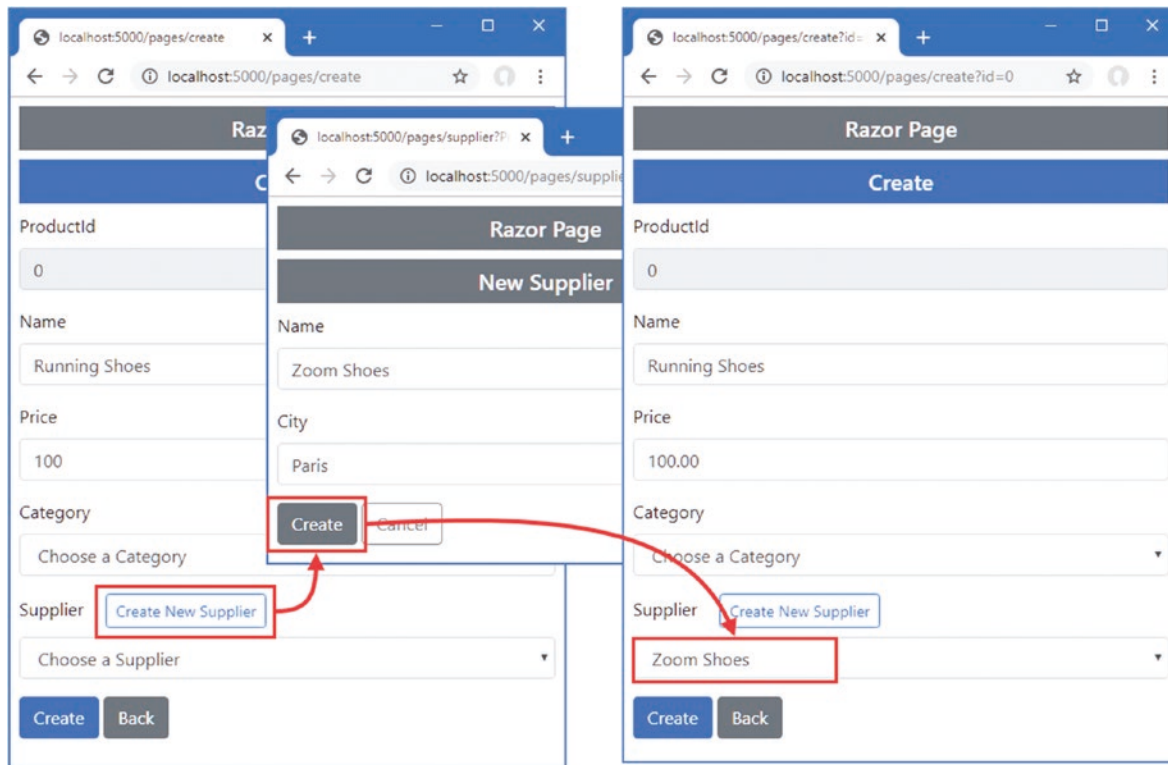
    public class EditModel: EditorPageModel {

        public EditModel(DataContext dbContext): base(dbContext) {}

        public async Task OnGetAsync(long id) {
            Product p = TempData.ContainsKey("product")
            ? JsonSerializer.Deserialize<Product>(TempData["product"] as string)
            : await this.DataContext.Products.FindAsync(id);
            ViewModel = ViewModelFactory.Edit(p, Categories, Suppliers);
        }

        public async Task<IActionResult> OnPostAsync([FromForm]Product product) {
            await CheckNewCategory(product);
            if (ModelState.IsValid) {
                product.Category = default;
                product.Supplier = default;
                DataContext.Products.Update(product);
                await DataContext.SaveChangesAsync();
                return RedirectToPage(nameof(Index));
            }
            ViewModel = ViewModelFactory.Edit(product, Categories, Suppliers);
            return Page();
        }
    }
}
```

The effect is that the user is presented with a Create New Supplier button, which sends the browser to a form that can be used to create a Supplier object. Once the Supplier has been stored in the database, the browser is sent back to the originating page, and the form is populated with the data the user had entered, and the Supplier select element is set to the newly created object, as shown in Figure 31-10.



**Figure 31-10.** Breaking out to create related data

## Summary

In this chapter, I demonstrated how the features described in earlier chapters can be combined with Entity Framework Core to create, read, update, and delete data. In Part 4, I describe some of the advanced features that ASP.NET Core provides.