========== GUIDE 11 CART AND CART ITEMS

Create the models, create the cart, add products to cart, display cart

End goal

A logged in user gets one cart.

A cart holds many cart items.

Each cart item points to one product and stores quantity.

Users add products to cart from the site.

Users view their cart on the Cart page.

STEP 1 Create the Cart model

Code

```
namespace RTWebApplication.Models
{
    public class Cart
    {
        public int Id { get; set; }

        public string UserId { get; set; } // Foreign key to ApplicationUser

        public ApplicationUser User { get; set; } // Navigation property to ApplicationUser

        public List<CartItem> CartItems { get; set; } = new ();
    }
}
```

step by step

1. This class represents the Cart table in the database.
2. Each row is one cart.
3. The cart is linked to one user.
4. The cart holds a list of items, CartItems.

Purpose of each key part

Id

Primary key: EF Core uses it to identify each cart row.

UserId

Foreign key. Stores the logged in user's Id from Identity. This is the column you filter on when you want "the current user's cart".

User

Navigation property. Lets EF Core load the full ApplicationUser object when needed. You do not always need this for the cart page, but it helps for admin views.

CartItems

Navigation property. One cart to many cart items. Initialised to an empty list so views do not crash with null.

STEP 2 Create the CartItem model

Code

```
namespace RTWebApplication.Models
{
    public class CartItem
    {
        public int Id { get; set; }
        public int CartId { get; set; } // Foreign key to Cart
        public Cart Cart { get; set; } // Navigation property to Cart
        public int ProductId { get; set; } // Foreign key to Product
        public Product Product { get; set; } // Navigation property to Product
        public int Quantity { get; set; }
    }
}
```

step by step

1. This class represents a single line in the cart.
2. It links a cart to a product.
3. It stores how many of that product the user wants.

Purpose of each key part

CartId

Foreign key linking this item to a Cart row.

Cart

Navigation property back to the Cart object.

ProductId

Foreign key linking this item to a Product row.

Product

Navigation property so you can access Product.Name and Product.Price in the view.

Quantity

How many units of the product are in the cart.

STEP 3

Add DbSet to ApplicationDbContext and run migrations

Why this step exists

EF Core only creates tables for models that are included in the DbContext.

What you need in ApplicationDbContext

Add these DbSets.

```
public DbSet<Cart> Carts { get; set; }
public DbSet<CartItem> CartItems { get; set; }
```

Then run migrations.

```
Add-Migration AddCartAndCartItems
Update-Database
```

What tables you should see

Carts table

CartItems table

Relationship expectation

CartItems has CartId, linking to Carts.Id

CartItems has ProductId, linking to Products.Id

STEP 4

Create CartController and define the key services

Why this step exists

The cart is user specific, so you need the logged in user id.

You also need database access.

Your controller header and constructor

```
[Authorize]
public class CartController : Controller
{
    private readonly ApplicationDbContext context;
    private readonly UserManager<ApplicationUser> userManager;

    public CartController(ApplicationDbContext context, UserManager<ApplicationUser> userManager)
    {
        this.context = context;
        this.userManager = userManager;
    }
}
```

step by step

1. Authorize forces login before any cart action runs.
2. ApplicationDbContext gives database access.
3. UserManager gives the current user id.

Purpose of each key part

Authorize

Stops carts being created for anonymous visitors. This is critical.

context

Used to query and save Carts and CartItems.

userManager

Used to get the current user's Identity id.

STEP 5

Create the helper method that finds or creates the user's cart

This method is the core of your cart system.

Code

```
public async Task<Cart> GetUserCartAsync()
{
    //Get the ID of the currently logged -in user
    var currentUserId = userManager.GetUserId(User);

    //Try to find the cart that belongs to this user
    // Include all items in the cart and the details of each product
    var currentUserCart = await context.Carts
        .Include(currentUserCart => currentUserCart.CartItems)
        .ThenInclude(CartItem => CartItem.Product)
        .FirstOrDefaultAsync(cartOwner => cartOwner.UserId == currentUserId);

    if (currentUserCart == null)
    {
        currentUserCart = new Cart
        {
            UserId = currentUserId
        };
        context.Carts.Add(currentUserCart);
        await context.SaveChangesAsync();
    }

    return currentUserCart;
}
```

step by step

1. Get the logged in user's Id from Identity.
2. Query the database for a Cart where Cart.UserId matches that Id.
3. While loading the cart, also load CartItems and each item's Product.
4. If no cart exists, create a new cart row for the user and save it.

5. Return the cart object.

Purpose of each key part

currentUserId

The filter key. This ensures each user only sees their own cart.

Include and ThenInclude

Ensures cart.CartItems is filled.

Ensures cartItem.Product is filled so the view can show name and price.

FirstOrDefaultAsync

Returns null when no cart exists for this user.

Create cart when null

Guarantees later code always has a cart to work with.

STEP 6

Create the DisplayCart action and view

Action code

```
public async Task<IActionResult> DisplayCart()
{
    var displayCart = await GetUserCartAsync();
    return View(displayCart);
}
```

step by step

1. Call the helper method to get the cart for the current user.
2. Send that cart object to the view.

Purpose

This keeps the action short.

It ensures the view always receives a Cart model, not a ViewResult.

Pitfall to avoid Do not do this:

return View(DisplayCart());

That passes a ViewResult into a view expecting Cart and triggers the InvalidOperationException error

---

STEP 7

AddProductToCart action

Add a product or increase quantity

Code

```
 [HttpPost]
public async Task<IActionResult> AddProductToCart(int productId)
{
    // STEP 1: Get the current user's cart (create one if it doesn't exist)
    var userCart = await GetUserCartAsync();

    // STEP 2: Check if the product is already in the cart
    // We look through CartItems and try to find a matching ProductId
    var existingCartItem = userCart.CartItems
        .FirstOrDefault(item => item.ProductId == productId);

    // STEP 3: Decide what to do
    if (existingCartItem != null)
    {
        // The product is already in the cart
        // Increase the quantity by 1
        existingCartItem.Quantity += 1;
    }
    else
    {
        // The product is NOT in the cart
        // Create a new CartItem and add it to the cart
        var newCartItem = new CartItem
        {
            ProductId = productId,   // Which product is being added
            Quantity = 1,            // Start with quantity of 1
            CartId = userCart.Id     // Link this item to the user's cart
        };

        userCart.CartItems.Add(newCartItem);
    }

    // STEP 4: Save all changes to the database
    await context.SaveChangesAsync();

    // STEP 5: Store a temporary success message
    TempData["CartMessage"] = "Product added to cart!";

    // STEP 6: Redirect the user back to the product list
    return RedirectToAction("Index", "Home");
}
```

step by step

1. Load the current user's cart. If it does not exist, create it.
2. Check if the product is already in the cart by searching CartItems for the same ProductId.
3. If it exists, increase Quantity by 1.
4. If it does not exist, create a new CartItem with Quantity 1 and link it to the cart.
5. Save changes to the database.
6. Put a success message in TempData.
7. Redirect back to Home Index.

Purpose of key parts

productId parameter

This is the product being added. Usually passed from a button on the product list.

existingCartItem lookup

Stops duplicate rows for the same product. Instead, it increases quantity, which is the correct cart behaviour.

SaveChangesAsync Without it, nothing is stored.

TempData Stores a message for the next request after redirect.

STEP 8

DisplayCart view

Loop items, show totals, show action buttons

Your view expects Cart:

```
@model Cart
```

What the view does, step by step

1. If CartItems is empty, it prints “Your cart is empty.”
2. Else it prints a table.
3. For each cart item, it prints product name, quantity, price, line total.
4. It shows action buttons to increase, decrease, remove.
5. It prints the cart total at the bottom.

STEP 9

Common errors and how to fix

Error

The model item passed into the ViewDataDictionary is of type ViewResult, but the view requires Cart

Cause

You passed an action result into a view as the model.

Fix

Call the helper method returning Cart, then pass that Cart to the view.

Error

NullReferenceException on product.Product.Name

Cause

Product navigation not loaded, or ProductId points to a deleted product.

Fix

Use Include ThenInclude when loading cart. Stop deletion of products that appear in carts, or handle null in the view.

Error

Cart duplicates for one user

Cause

No unique constraint on UserId, plus race conditions.

Fix

Add unique index on Carts.UserId. Reload cart after a failed insert.

Error

AddProductToCart creates rows but DisplayCart shows empty

Cause

You added CartItem but did not save changes, or CartId not set correctly.

Fix

Ensure SaveChangesAsync runs.

Ensure new CartItem has CartId set or Cart navigation set.

Error

Increase, Decrease, Remove buttons do nothing

Cause

Controller actions not implemented.

Fix

Add IncreaseQuantity, DecreaseQuantity, RemoveItem POST actions.

STEP 10

Layout dropdown menu

Add a Cart link in the dropdown

Use this exact pattern in _Layout.cshtml, inside the dropdown menu list.

```
<li>
    <a class="dropdown-item" asp-controller="Cart" asp-action="DisplayCart">Cart</a>
</li>
```

When the user clicks Cart, they go to CartController.DisplayCart.

That action loads their cart and returns the DisplayCart view.

Pitfall If your controller class name is CartController, the controller route is Cart. If you rename the controller, update asp-controller to match.
