Contents

[HttpPost to Create a New ResourceBook 2](#_Toc204882884)

[Step 1: Define the Models (already done) 2](#_Toc204882885)

[Step 2: Define the DbContext (already done) 2](#_Toc204882886)

[Step 3: Create DTO for Incoming POST Data 3](#_Toc204882887)

[Step 4: Write the Controller with HttpPost Method 3](#_Toc204882888)

[Optional: Add GET by id for CreatedAtAction reference (already done) 4](#_Toc204882889)

[Example of the Expected JSON POST Payload 5](#_Toc204882890)

[Notes 5](#_Toc204882891)

[. 5](#_Toc204882892)

[APPENDIX 1 6](#_Toc204882893)

[Purpose of the CreateResourceBookDTO 6](#_Toc204882894)

# HttpPost to Create a New ResourceBook

Post method in an **ASP.NET Core Web API** controller that creates a new ResourceBook, inserts a related Resource entry, and saves them to a SQL Server database (StudyResource). This assumes you're using **Entity Framework Core** for your data access.

## Step 1: Define the Models (already done)

// Models/Resource.cs

public class Resource

{

public int ResourceId { get; set; } // Auto-incrementing PK

public string Title { get; set; }

public int ResourceTypeId { get; set; }

public int ResourceTopicId { get; set; }

public ResourceBook ResourceBook { get; set; }

}

// Models/ResourceBook.cs

public class ResourceBook

{

public int Id { get; set; } // Auto-incrementing PK

public string Author { get; set; }

public int Pages { get; set; }

public decimal Cost { get; set; }

public DateTime PublishDate { get; set; }

public int ResourceId { get; set; } // FK to Resource

public Resource Resource { get; set; }

}

## Step 2: Define the DbContext (already done)

// Data/StudyResourceContext.cs

public class StudyResourceContext : DbContext

{

public StudyResourceContext(DbContextOptions<StudyResourceContext> options)

: base(options)

{

}

public DbSet<Resource> Resources { get; set; }

public DbSet<ResourceBook> ResourceBooks { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Resource>()

.HasKey(r => r.ResourceId);

modelBuilder.Entity<ResourceBook>()

.HasKey(rb => rb.Id);

modelBuilder.Entity<Resource>()

.HasOne(r => r.ResourceBook)

.WithOne(rb => rb.Resource)

.HasForeignKey<ResourceBook>(rb => rb.ResourceId)

.OnDelete(DeleteBehavior.Cascade);

}

}

## Step 3: Create DTO for Incoming POST Data

// DTOs/CreateResourceBookDto.cs

public class CreateResourceBookDto

{

public string Title { get; set; }

public int ResourceTypeId { get; set; }

public int ResourceTopicId { get; set; }

public string Author { get; set; }

public int Pages { get; set; }

public decimal Cost { get; set; }

public DateTime PublishDate { get; set; }

}

## Step 4: Write the Controller with HttpPost Method

// Controllers/ResourceBooksController.cs

[ApiController]

[Route("api/[controller]")]

public class ResourceBooksController : ControllerBase

{

private readonly StudyResourceContext \_context;

public ResourceBooksController(StudyResourceContext context)

{

\_context = context;

}

[HttpPost]

public async Task<IActionResult> CreateResourceBook([FromBody] CreateResourceBookDto dto)

{

if (!ModelState.IsValid)

return BadRequest(ModelState);

// Create Resource entity

var resource = new Resource

{

Title = dto.Title,

ResourceTypeId = dto.ResourceTypeId,

ResourceTopicId = dto.ResourceTopicId

};

// Add Resource to context to generate ResourceId

\_context.Resources.Add(resource);

await \_context.SaveChangesAsync();

// Create ResourceBook entity with FK

var resourceBook = new ResourceBook

{

Author = dto.Author,

Pages = dto.Pages,

Cost = dto.Cost,

PublishDate = dto.PublishDate,

ResourceId = resource.ResourceId

};

\_context.ResourceBooks.Add(resourceBook);

await \_context.SaveChangesAsync();

return CreatedAtAction(nameof(GetResourceBook), new { id = resourceBook.Id }, resourceBook);

}

## Optional: Add GET by id for CreatedAtAction reference (already done)

[HttpGet("{id}")]

public async Task<ActionResult<ResourceBook>> GetResourceBook(int id)

{

var resourceBook = await \_context.ResourceBooks

.Include(rb => rb.Resource)

.FirstOrDefaultAsync(rb => rb.Id == id);

if (resourceBook == null)

{

return NotFound();

}

return resourceBook;

}

}

## Example of the Expected JSON POST Payload

{

"title": "Python",

"resourceTypeId": 1,

"resourceTopicId": 2,

"author": "Shovic",

"pages": 172,

"cost": 19.99,

"publishDate": "2024-01-01"

}

## Notes

This assumes a one-to-one relationship between Resource and ResourceBook.

# .

## APPENDIX 1

## Purpose of the CreateResourceBookDTO

* Accept the **combined data** from the client (in this case, both Resource and ResourceBook fields) in one request.
* **Decouple** the API's request/response model from your database model (entities).
* Make the API **easier and safer to use**, by only exposing the fields needed to create a record.

To create a new **ResourceBook** and its related **Resource** in a single HTTP POST request for the two related tables,

*(****ResourceBook*** *which contains a* ***FK ResourceId*** *which is the* ***ResourceId/PK*** *in* ***Resource****, and* ***Resource*** *,*

*it is necessary to:*

1. Create a new Resource record (so SQL Server will generate a ResourceId),
2. Then create the ResourceBook record using that ResourceId as a **foreign key**.

However the client doesn’t know the ResourceId yet — because it hasn’t been generated. That's why your API needs to **accept both sets of fields** and handle the linking logic internally.

**What is inside CreateResourceBookDto?**

This DTO contains:

**Resource fields:**

* Title
* ResourceTypeId
* ResourceTopicId

**ResourceBook fields:**

* Author
* Pages
* Cost
* PublishDate

That way, the client just sends one JSON object, and your controller knows how to split it into two entities and link them correctly.

**✅ Benefits of using a DTO here**

* **Simplifies the client API**: The client doesn't need to send two requests (one for Resource, one for ResourceBook).
* **Keeps database concerns hidden**: The client doesn't need to worry about foreign keys or ResourceId generation.
* **Improves validation**: You can annotate DTO properties with [Required], [Range], etc., to catch errors before hitting the database.
* **Decouples internal model from API contract**: You can change your entity structure without breaking the API contract.

**🧠 Summary**

So to answer your question directly:

Does it have to do with the fact that Resource and ResourceBooks are related by foreign key?

✅ **Yes, exactly.**  
The DTO lets the client send **both parts of the data needed** for two linked tables, and your controller handles the creation and FK assignment behind the scenes.