Contents

[HttpPost to Create a New ResourceBook 2](#_Toc204932836)

[Step 1: Define the Models (already done) 2](#_Toc204932837)

[Step 2: Define the DbContext (already done) 2](#_Toc204932838)

[Step 3: Create DTO for Incoming POST Data 3](#_Toc204932839)

[Step 4: Write the Controller with HttpPost Method 4](#_Toc204932840)

[Optional: Add GET by id for CreatedAtAction reference (already done) 5](#_Toc204932841)

[Example of the Expected JSON POST Payload 5](#_Toc204932842)

[Notes 5](#_Toc204932843)

[. 5](#_Toc204932844)

[APPENDIX 1 6](#_Toc204932845)

[Purpose of the CreateResourceBookDTO 6](#_Toc204932846)

[APPENDIX 2: 8](#_Toc204932847)

[AspNetCore.Mvc.FromBodyAttribute 8](#_Toc204932848)

[What is "the body of an HTTP request"? 10](#_Toc204932849)

[What Is the Method Parameter? 12](#_Toc204932850)

[Is [FromBody] *always* needed? No 13](#_Toc204932851)

[You must use [FromBody] when: 13](#_Toc204932852)

[ You have multiple parameters, and you want to specify which one should come from the body. 13](#_Toc204932853)

[ You want to be explicit and avoid ambiguity in model binding. 13](#_Toc204932854)

[ You're not using [ApiController] 13](#_Toc204932855)

# 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

[HttpPost]

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

{

if (!ModelState.IsValid)

return BadRequest(ModelState);

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);

}

|  |
| --- |
|  |

See APPENDIX 2 for detailed explanation of FromBody attribute

## 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.

# APPENDIX 2:

## AspNetCore.Mvc.FromBodyAttribute

**What it is, What it Does, and When to Use It**

**🔹 What It Is**

[FromBody] is an attribute in **ASP.NET Core** (specifically from the Microsoft.AspNetCore.Mvc namespace).

|  |
| --- |
| It's used in **Web API controllers** to tell the framework to **bind a method parameter to the body of an HTTP request**. |

It's used in **Web API controllers** to tell the framework to **bind a method parameter to the body of an HTTP request**.

The **body** is:

{

"id": 1,

"name": "Laptop",

"price": 999.99

}

* The **FromBodyAttribute** is part of ASP.NET Core’s **model binding system**, which maps incoming HTTP request data to controller action parameters.

**🔹 What It Does**

When you decorate a parameter with [FromBody], ASP.NET Core:

1. **Reads the HTTP request body** (typically JSON, but can be XML, etc. depending on configuration).
2. **Deserializes it** using the configured input formatter (usually JSON via System.Text.Json or

Newtonsoft.Json).

1. **Binds the resulting object** to the parameter in your controller action.

**🔹 Example Usage**

[ApiController]

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

public class ProductsController : ControllerBase

{

[HttpPost]

public IActionResult CreateProduct([FromBody] Product product)

{

if (!ModelState.IsValid)

{

return BadRequest(ModelState);

}

// Save product to DB, etc.

return Ok(product);

}

}

**Request Example (JSON body):**

{

"id": 1,

"name": "Laptop",

"price": 999.99

}

**🔹 Why It’s Needed**

* ASP.NET Core uses multiple sources for binding (route, query string, form, body, etc.).
* Without an attribute, the framework tries to infer the source — but **only one parameter per action**

**can be bound from the body**.

* [FromBody] makes it **explicit and unambiguous**.

**🔹 When to Use It**

Use [FromBody] when:

* You want to **accept complex types** (like classes) from the request body.
* You are building RESTful APIs that **accept JSON data** in POST/PUT/PATCH requests.
* You want to **deserialize structured input** from the client.

**🔸 Notes**

* Only **one parameter** in an action can use [FromBody], because the body can only be read once.
* For simple types like int, string, etc., use [FromQuery], [FromRoute], or [FromForm] instead.

**🔹 Alternatives**

| **Attribute** | **Source of Binding** |
| --- | --- |
| [FromQuery] | URL query string |
| [FromRoute] | Route values |
| [FromForm] | Form data (e.g., HTML forms) |
| [FromHeader] | HTTP headers |
| [FromBody] | HTTP request body |
| [FromServices] | DI services |

## What is "the body of an HTTP request"?

The **body** of an HTTP request is the part of the request where the **actual data or content** is sent. It is **not part of the URL**, headers, or query string.

Think of it as the **payload** — especially important in POST, PUT, and PATCH requests — where the client sends structured data (like JSON) to the server.

**🔸 Example: HTTP Request with a Body**

Suppose you're calling the CreateProduct action from earlier:

POST /api/products HTTP/1.1

Host: example.com

Content-Type: application/json

{

"id": 1,

"name": "Laptop",

"price": 999.99

}

* The **URL** is: /api/products
* The **HTTP method** is: POST
* The **header** Content-Type: application/json tells the server the body is JSON.

The **body** is:

{

"id": 1,

"name": "Laptop",

"price": 999.99

}

**🔸 Where the Body Fits in the HTTP Request**

Here's a simplified breakdown:

POST /api/products HTTP/1.1

Host: example.com

Content-Type: application/json

<-- Headers end here -->

{

"id": 1,

"name": "Laptop",

"price": 999.99

}

<-- This is the body -->

**🔸 When Is the Body Used?**

* **POST** — To create a new resource (e.g., send a full object).
* **PUT/PATCH** — To update an existing resource.
* **Not used in GET** — GET requests usually do **not** have a body; they rely on query strings or route parameters.

**🔸 How [FromBody] Relates**

In this context:

public IActionResult CreateProduct([FromBody] Product product)

* ASP.NET Core takes the **raw JSON from the body**.
* It **deserializes it into a Product object**.
* That object is passed into your controller method.

## What Is the Method Parameter?

In this example:

public IActionResult CreateProduct([FromBody] Product product)

The **method parameter** that needs to be bound is:

Product product

* Product is the **type** (a class you define, usually representing a model or DTO).
* product is the **parameter name**.
* It's the parameter passed into the CreateProduct method of the controller.

In WebApplication2’s ResourceBookController the parameter is CreateResourceBookDTO dto

dto is the **parameter** that will receive the deserialized object.

**🔹 Why Does It Need Binding?**

HTTP requests only send data — typically as raw text (like JSON).

ASP.NET Core doesn't automatically know how to turn that text into a Product object.

So:

* The client sends JSON data in the **body**.
* ASP.NET Core needs to **deserialize** that JSON into a Product object.
* [FromBody] tells ASP.NET Core:  
  ➜ “Hey, take the JSON from the body of the request, and use it to create the product parameter.”

**🔸 Visual Breakdown**

If the client sends this JSON in the body:

{

"id": 1,

"name": "Laptop",

"price": 999.99

}

ASP.NET Core will:

* Read the JSON from the request body.
* Match it to the properties of your Product class.
* Instantiate a Product object with:
* product.Id = 1;
* product.Name = "Laptop";
* product.Price = 999.99;
* Pass that product object into your action method.

**🔸 Summary**

* **Method parameter**: Product product
* **Source of binding**: HTTP request body
* **Instruction to the framework**: [FromBody] tells it to deserialize from body into that parameter.

Let me know if you'd like a quick view of what the Product class looks like or how [FromBody] behaves with different content types.

## Is [FromBody] *always* needed? No

[FromBody] is **not strictly required in all cases** — **but** it's often necessary or recommended depending on the context.

When [FromBody] *Is Required*

### You must use [FromBody] when:

### You have multiple parameters, and you want to specify which one should come from the body.

### You want to be explicit and avoid ambiguity in model binding.

### You're not using [ApiController]

(more on this below).

Example:

public IActionResult PostSomething(string id, [FromBody] MyModel data)

Without [FromBody], ASP.NET Core wouldn't know where to get data from — it defaults to [FromRoute], [FromQuery], etc., and not the body.

**🔸 When [FromBody] Is *Optional***

If your controller is decorated with:

[ApiController]

Then **ASP.NET Core automatically infers** [FromBody] for **complex types** (like classes, DTOs). So this works fine:

[ApiController]

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

public class BooksController : ControllerBase

{

[HttpPost]

public IActionResult AddBook(CreateBookDTO dto) // No [FromBody], still works

{

// ...

}

}

This behavior is enabled by the **[ApiController] attribute**, which introduces "parameter inference".

**🔹 TL;DR – When Do You Need [FromBody]?**

| **Scenario** | **Is [FromBody] Required?** | **Why** |
| --- | --- | --- |
| Controller has [ApiController], single complex type param | ❌ Optional | Inferred automatically |
| No [ApiController], using complex type | ✅ Required | Otherwise won’t bind from body |
| Multiple parameters, only one should bind from body | ✅ Required | Must be explicit |
| Simple types like int, string, etc. | ❌ Use [FromQuery] or [FromRoute] instead |  |

**🔸 Best Practice?**

Even though it’s sometimes optional, many developers prefer to use [FromBody] **explicitly** for clarity and to avoid surprises — especially when onboarding new developers or writing unit tests.

Let me know if you’d like a chart showing where other attributes like [FromQuery] or [FromRoute] apply!

Top of Form

Bottom of Form

ChatGPT can make mistakes. Check important info. See Cookie Preferences.