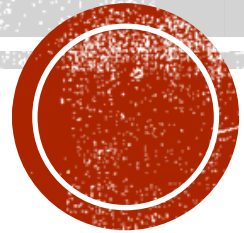
MODEL BINDING AND VALIDATION IN MINIMAL APIS



Kamal Beydoun
Lebanese University – Faculty of Sciences I
Kamal.beydoun@ul.edu.lb

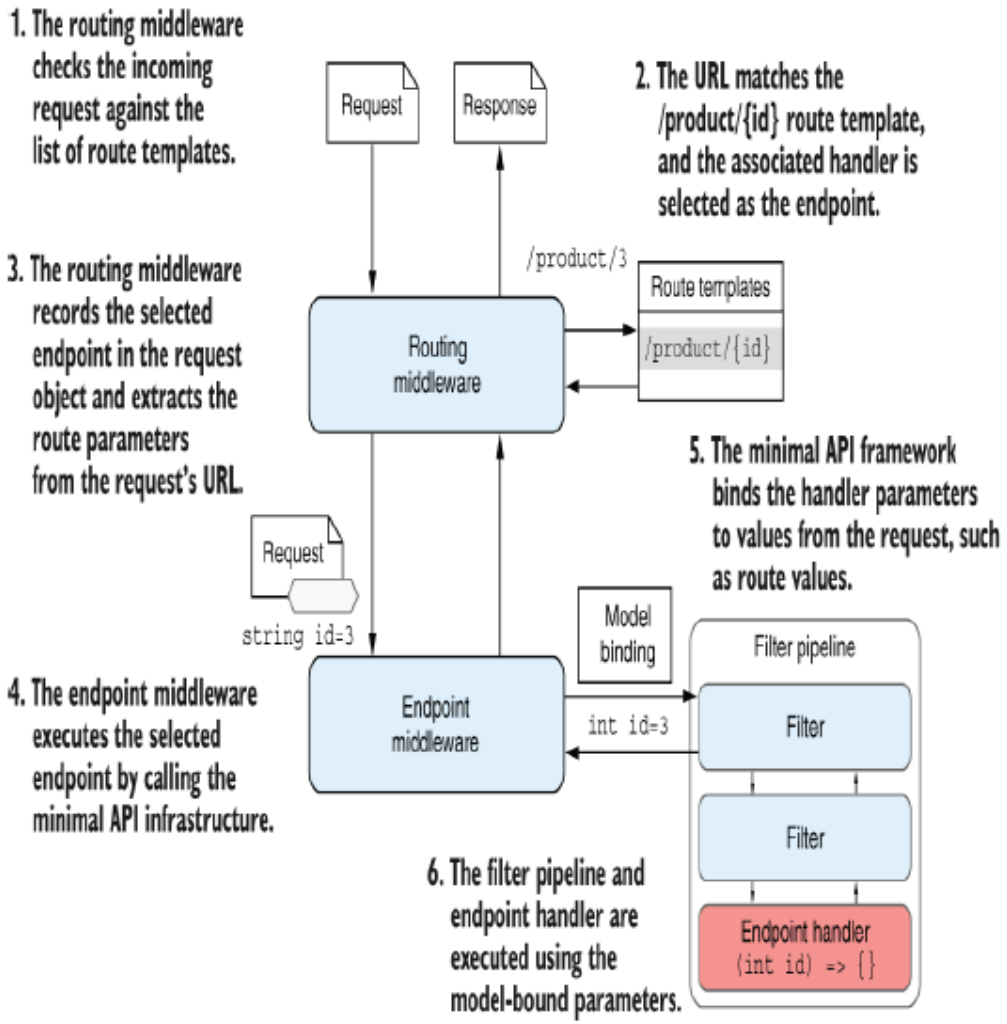
EXTRACTING VALUES FROM A REQUEST WITH MODEL BINDING

- Model binding extracts values **from a request** and uses them to create .NET objects.
  - These objects are **passed as method parameters** to the endpoint **handler** being executed.
- The **model binder** is responsible for looking **through the request** that comes in and finding values to use.
  - It creates objects of the **appropriate type** and assigns these values to your model in a process called **binding**.

EXTRACTING VALUES FROM A REQUEST WITH MODEL BINDING

- Model binding in minimal APIs (and in Razor Pages and Model-View-Controller [MVC]) is a **one-way population** of objects from the request, not the **two-way data binding** that desktop or mobile development sometimes uses.
- ASP.NET Core automatically creates the **arguments that are passed to your handler** by using the **request's properties**
  - the request URL, headers sent in the HTTP request, any data POSTed in the request body, and so on.
- Model binding happens **before** the filter pipeline and your endpoint handler execute.

EXTRACTING VALUES FROM A REQUEST WITH MODEL BINDING



EXTRACTING VALUES FROM A REQUEST WITH MODEL BINDING

Binding sources to create the handler arguments :

- **Route values**—These values are obtained from URL segments
- **Query string values**—These values are passed at the end of the URL, not used during routing.
- **Header values**—Header values are provided in the HTTP request.

EXTRACTING VALUES FROM A REQUEST WITH MODEL BINDING

Binding sources to create the handler arguments :

- **Body JSON**—A single parameter may be bound to the JSON body of a request.
- **Dependency injected services**—Services available through dependency injection can be used as endpoint handler arguments.
- **Custom binding**—ASP.NET Core exposes methods for you to customize how a type is bound by providing access to the HttpRequest object.

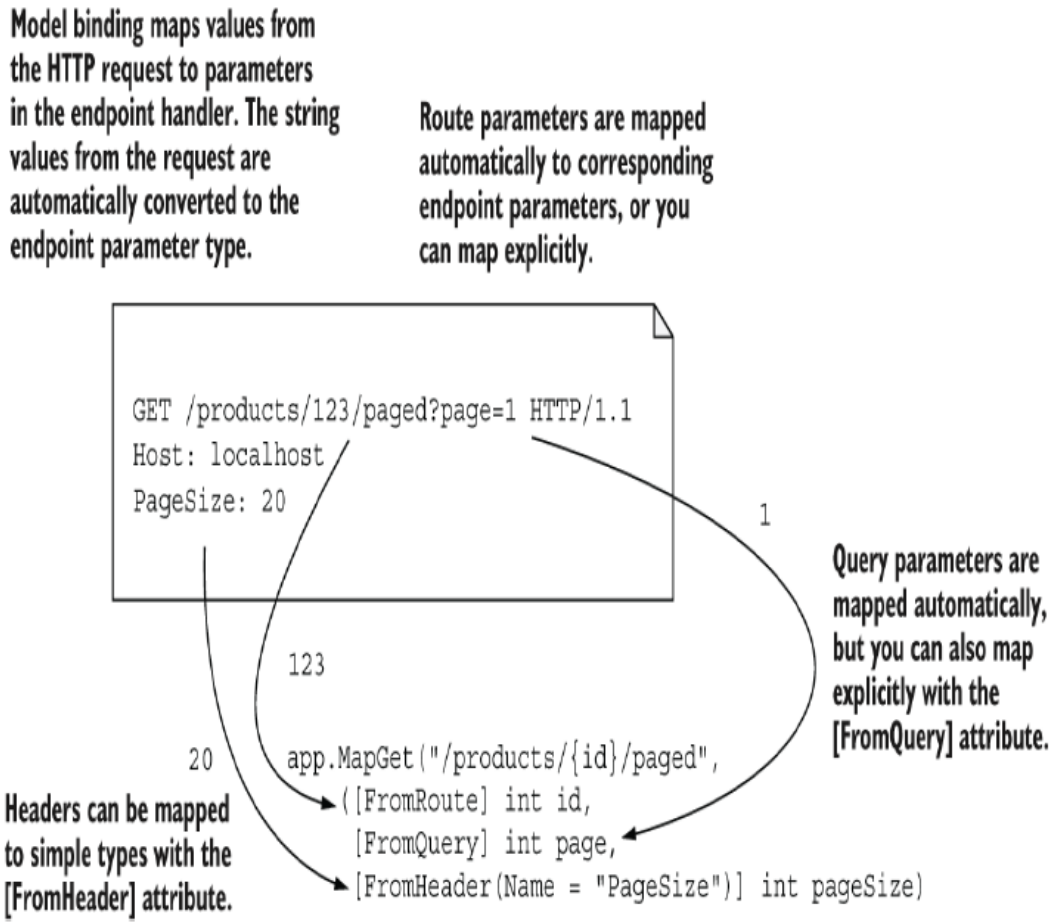
BINDING SIMPLE TYPES TO A REQUEST

- ASP.NET Core automatically tries to bind the value to a route parameter, or a query string value:
- **If the name of the handler parameter matches the name of a route parameter in the route template,** ASP.NET Core binds to the associated route value.
- If the name of the handler parameter **doesn't match** any parameters in the route template, ASP.NET Core tries to **bind to a query string value**.

BINDING SIMPLE TYPES TO A REQUEST

- In addition to this "**automatic**" inference, you can force ASP.NET Core to bind from a specific source by adding **attributes** to the parameters.
- [**FromRoute**] explicitly binds to route parameters
- [**FromQuery**] to the query string
- [**FromHeader**] to header values

BINDING SIMPLE TYPES TO A REQUEST



BINDING SIMPLE TYPES TO A REQUEST

```
using Microsoft.AspNetCore.Mvc;                                    ❶

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/products/{id}/paged",
    ([FromRoute] int id,                                           ❷
     [FromQuery] int page,                                         ❸
     [FromHeader(Name = "PageSize")] int pageSize)                 ❹
     => $"Received id {id}, page {page}, pageSize {pageSize}");

app.Run();
```

❶ All the [From*] attributes are in this namespace.

❷ [FromRoute] forces the argument to bind to the route value.

❸ [FromQuery] forces the argument to bind to the query string.

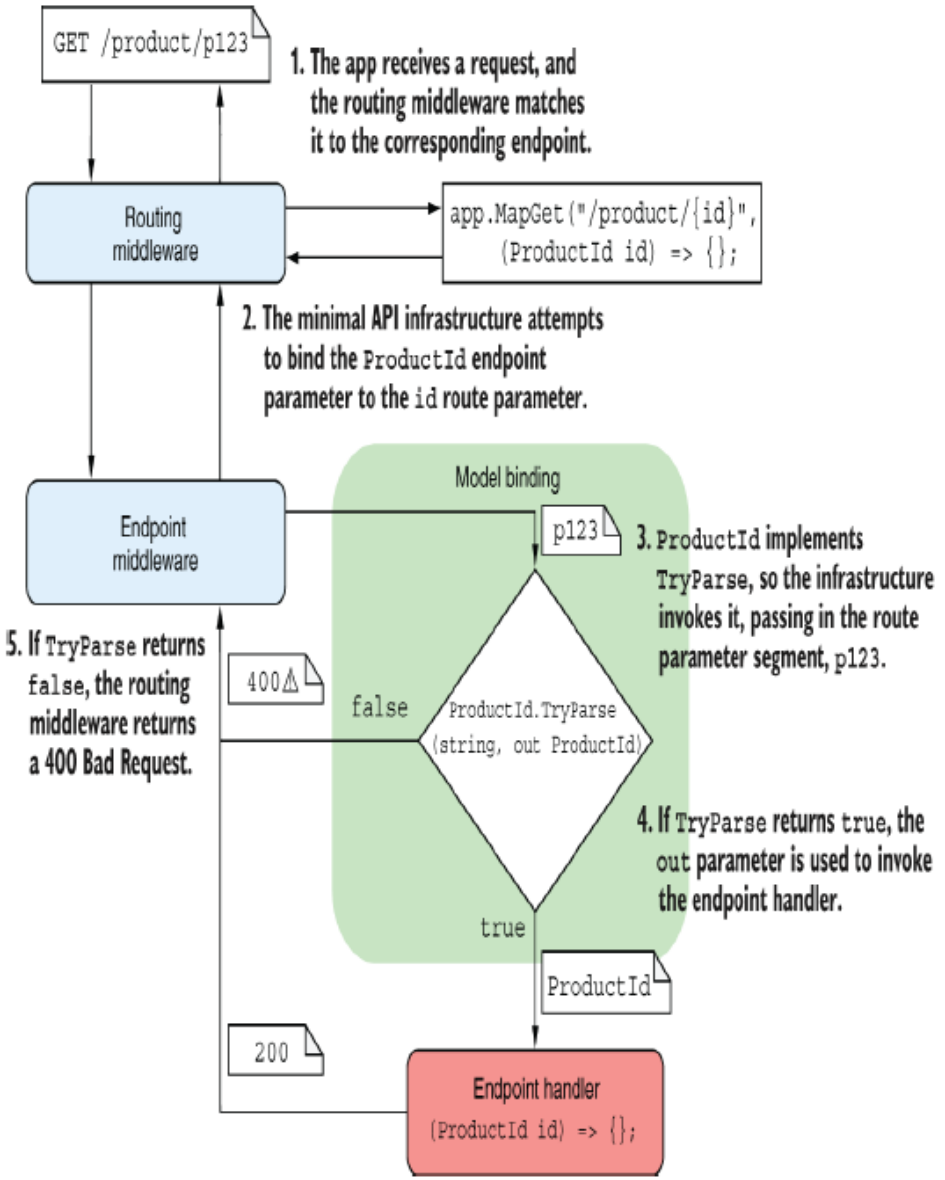
❹ [FromHeader] binds the argument to the specified header.

NOTE

- When the minimal API infrastructure fails to bind a handler parameter due to an **incompatible format**, it throws a BadHttpRequestException and returns a 400 Bad Request response.
- A **simple type** is defined as any type that contains either of the following **TryParse** methods, where T is the implementing type.

```
public static bool TryParse(string value, out T result);
public static bool TryParse(
    string value, IFormatProvider provider, out T result);
```

TRYPARSE


GET /product/p123
1. The app receives a request, and the routing middleware matches it to the corresponding endpoint.
Routing middleware
app.MapGet("/product/{id}", (ProductId id) => {};
2. The minimal API infrastructure attempts to bind the ProductId endpoint parameter to the id route parameter.
Model binding
Endpoint middleware
p123
3. ProductId implements TryParse, so the infrastructure invokes it, passing in the route parameter segment, p123.
5. If TryParse returns false, the routing middleware returns a 400 Bad Request.
400
false
ProductId.TryParse (string, out ProductId)
4. If TryParse returns true, the out parameter is used to invoke the endpoint handler.
true
ProductId
200
Endpoint handler
(ProductId id) => {};

TRYPARSE

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/product/{id}", (ProductId id) => $"Received {id}");     ❶
app.Run();

readonly record struct ProductId(int Id)                             ❷
{
    public static bool TryParse(string? s, out ProductId result)    ❸
    {
        if(s is not null                                             ❹
           && s.StartsWith('p')                                      ❹
           && int.TryParse(                                          ❺
                s.AsSpan().Slice(1),                                 ❻
                out int id))                                         ❼
        {
            result = new ProductId(id);                              ❽
            return true;                                             ❽
        }

        result = default;                                            ❾
        return false;                                                ❾
    }
}
```

BINDING COMPLEX TYPES TO THE JSON BODY

- Model binding in minimal APIs relies on certain **conventions** to simplify the code you need to write.
  - minimal API endpoints assume that requests will be sent using JSON.
- Minimal APIs can bind the body of a request to a **single complex type** in your endpoint handler by deserializing the request from JSON.

BINDING COMPLEX TYPES TO THE JSON BODY

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapPost("/product", (Product product) => $"Received {product}");  ❶

app.Run();

record Product(int Id, string Name, int Stock);                       ❷
```

❶ Product is a complex type, so it's bound to the JSON body of the request.

❷ Product doesn't implement TryParse, so it's a complex type.

```
{ "id": 1, "Name": "Shoes", "Stock": 12 }
```

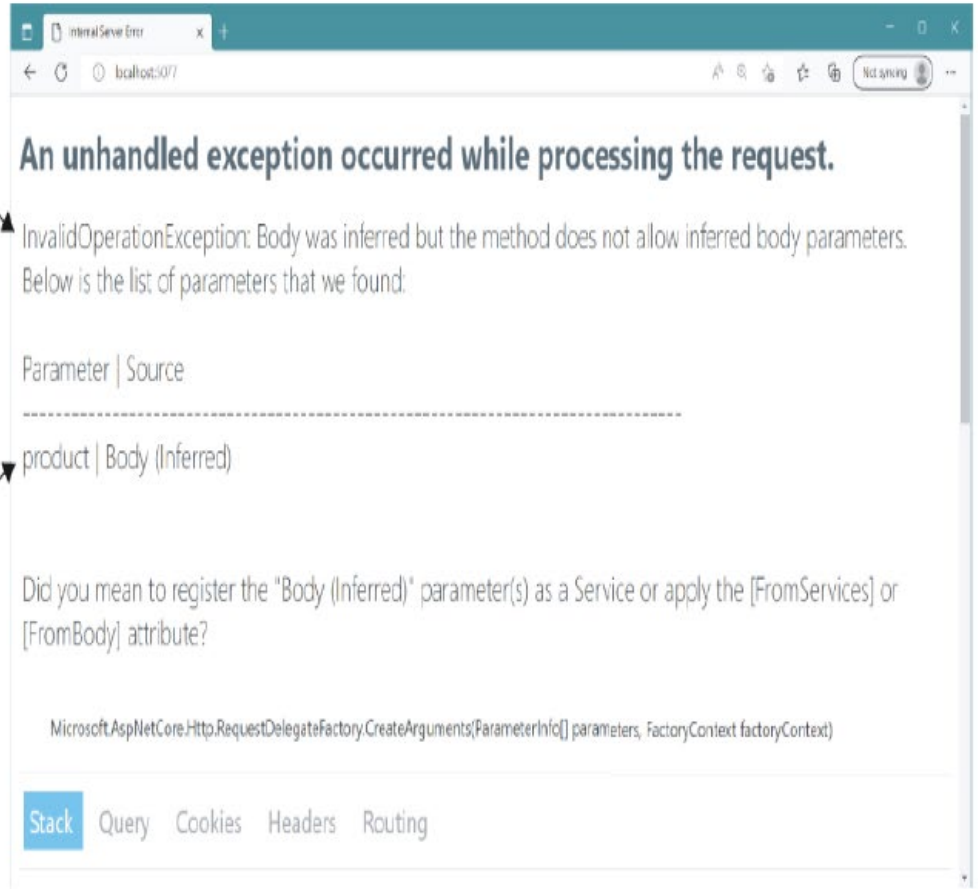
BINDING COMPLEX TYPES TO THE JSON BODY

- You can bind only a **single handler parameter** to the JSON body. If **more than one complex** parameter is eligible to bind to the body, you'll get **an exception at runtime** when the app receives its first request.
- If the request body **isn't JSON**, the endpoint handler won't run, and the EndpointMiddleware will return a **415 Unsupported Media Type response**.
- If you try to bind to the body for an HTTP verb that usually **doesn't send a body** (GET, HEAD, OPTIONS, DELETE, TRACE, and CONNECT), you'll get an exception at runtime.

BINDING COMPLEX TYPES TO THE JSON BODY

Attempting to read the body of a GET request causes an exception.

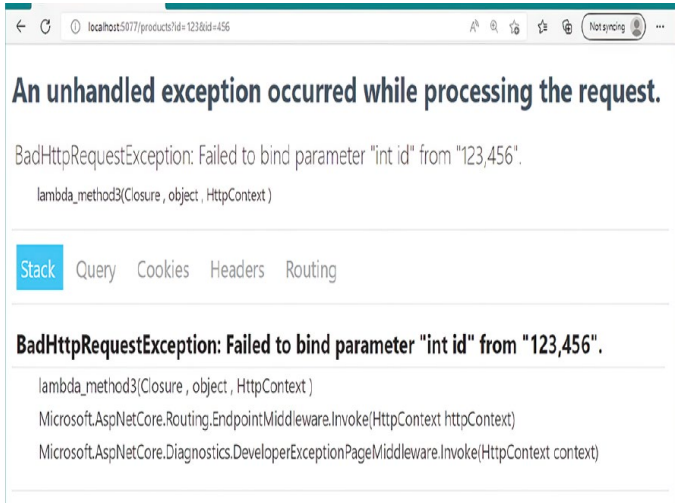
The exception indicates which parameter caused the exception and the binding source used.



ARRAYS: SIMPLE TYPES OR COMPLEX TYPES?

```
/products?id=123&id=456
```

URL is valid




```
app.MapGet("/products", (int id) => $"Received {id}");
```

- /products?id=123 would bind the id parameter to the query string
- /products?id=123&id=456, will cause a **runtime error**

ARRAYS: SIMPLE TYPES OR COMPLEX TYPES?

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/products/search",
    (int[] id) => $"Received {id.Length} ids");     ❶

app.Run();
```

❶ The array will bind to multiple instances of id in the query string.

```
app.MapGet("/products/search",
    ([FromQuery(Name = "id")] int[] ids) => $"Received {ids.Length} ids");
```

ARRAYS: SIMPLE TYPES OR COMPLEX TYPES?

Arrays work as described **only if** :

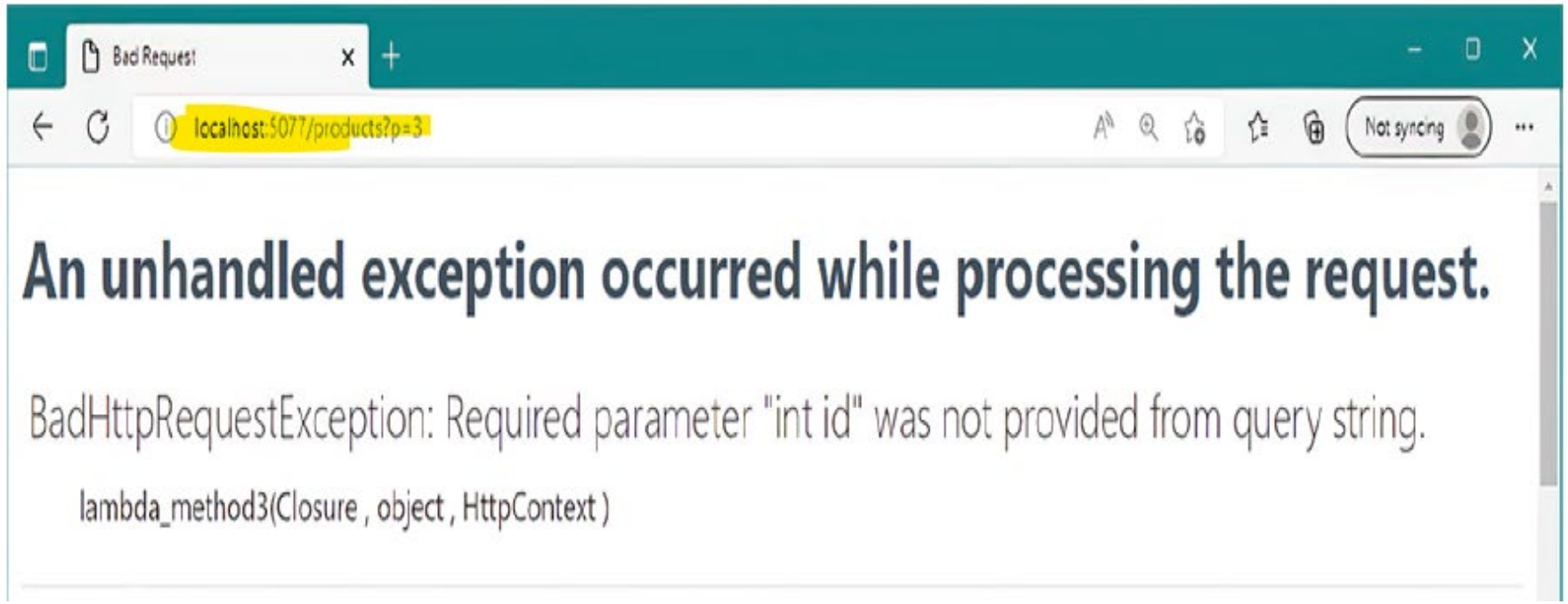
- You're using an HTTP verb that typically **doesn't include** a request body, such as GET, HEAD, or DELETE.
- The array is an array of simple types (or string[] or StringValues).

If either of these statements is not true, ASP.NET Core will attempt **to bind the array to the JSON body** of the request instead.

MAKING PARAMETERS OPTIONAL WITH NULLABLES

```
app.MapGet("/stock/{id?}", (int id) => $"Received {id}");
```

/stock/123 and /stock will invoke the handler



MAKING PARAMETERS OPTIONAL WITH NULLABLES

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/stock/{id?}", (int? id) => $"Received {id}");          ❶

app.MapGet("/stock2", (int? id) => $"Received {id}");               ❷

app.MapPost("/stock", (Product? product) => $"Received {product}"); ❸

app.Run();
```

❶ Uses a nullable simple type to indicate that the value is optional, so id is null when calling /stock

❷ This example binds to the query string. Id will be null for the request /stock2.

❸ A nullable complex type binds to the body if it's available; otherwise, it's null.

MAKING PARAMETERS OPTIONAL WITH NULLABLES

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/stock", StockWithDefaultValue);                    ❶
```

```
app.Run();

string StockWithDefaultValue(int id = 0) => $"Received {id}";     ❷
```

❶ The local function StockWithDefaultValue is the endpoint handler.

❷ The id parameter binds to the query string value if it's available; otherwise, it has the value 0.

BINDING SERVICES AND SPECIAL TYPES

Three types of parameters:

- **Well-known types**—that is, hard-coded types that ASP.NET Core knows about, such as HttpContext and HttpRequest
- **IFormFileCollection** and **IFormFile** for working with file uploads
- Application **services** registered in WebApplicationBuilder.Services

INJECTING WELL-KNOWN TYPES

Well-known types in your minimal API endpoint **handlers**:

- **HttpContext**—This type contains all the details on both the request and the response.
- **HttpRequest**—Equivalent to the property HttpContext.Request
- **HttpResponse**—Equivalent to the property HttpContext.Response
- **CancellationToken**—Equivalent to the property HttpContext.RequestAborted, this token is canceled if the client aborts the request.

INJECTING WELL-KNOWN TYPES

Well-known types in your minimal API endpoint **handlers**:

- **ClaimsPrincipal**—Equivalent to the property HttpContext.User, this type contains authentication information about the user.
- **Stream**—Equivalent to the property HttpRequest.Body, this parameter is a reference to the Stream object of the request.
- **PipeReader**—Equivalent to the property HttpContext.BodyReader, PipeReader provides a higher-level API compared with Stream,

INJECTING SERVICES

```
app.MapGet("/links", (LinkGenerator links) =>    ❶
{
    string link = links.GetPathByName("products");
    return $"View the product at {link}";
});
```

❶ The LinkGenerator can be used as a parameter because it's available in the DI container.

```
app.MapGet("/links", ([FromServices] LinkGenerator links) =>
```

BINDING FILE UPLOADS WITH IFORMFILE AND IFORMFILECOLLECTION

ASP.NET Core supports uploading files by exposing

- IFormFile interface

```
app.MapGet("/upload", (IFormFile file) => {});
```

- IFormFileCollection if you need to accept multiple files

```
app.MapGet("/upload", (IFormFileCollection files) =>
{
    foreach (IFormFile file in files)
    {
    }
});
```

BINDING FILE UPLOADS WITH IFORMFILE AND IFORMFILECOLLECTION

- The IFormFile object exposes several properties and utility methods for reading the contents of the uploaded file

```
public interface IFormFile
{
    string ContentType { get; }
    long Length { get; }
    string FileName { get; }
    Stream OpenReadStream();
}
```

CUSTOM BINDING WITH BINDASYNC

- To add **custom binding** for a parameter type, you must implement one of the following two static **BindAsync** methods in your type T:

```
public static ValueTask<T?> BindAsync(HttpContext context);
public static ValueTask<T?> BindAsync(
HttpContext context, ParameterInfo parameter);
```

CUSTOM BINDING WITH BINDASYNC

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapPost("/sizes", (SizeDetails size) => $"Received {size}");          ❶

app.Run();

public record SizeDetails(double height, double width)                    ❷
{                                                                         ❷
    public static async ValueTask<SizeDetails?> BindAsync(                ❷
        HttpContext context)                                              ❷
    {
        using var sr = new StreamReader(context.Request.Body);            ❸

        string? line1 = await sr.ReadLineAsync(context.RequestAborted);   ❹
        if (line1 is null) { return null; }                               ❺

        string? line2 = await sr.ReadLineAsync(context.RequestAborted);   ❹
        if (line2 is null) { return null; }                               ❺

        return double.TryParse(line1, out double height)                  ❻
            && double.TryParse(line2, out double width)                   ❻
            ? new SizeDetails(height, width)                              ❼
            : null;                                                       ❽
    }
}
```

❶ No extra attributes are needed for the SizeDetails parameter, as it has a BindAsync method.

❷ SizeDetails implements the static BindAsync method.

❸ Creates a StreamReader to read the request body

❹ Reads a line of text from the body

❺ If either line is null, indicating no content, stops processing

❻ Tries to parse the two lines as doubles

❼ If the parsing is successful, creates the SizeDetails model and returns it . . .

❽ . . . otherwise, returns null

CHOOSING A BINDING SOURCE

The **first binding source that matches is** the one it uses:

1. If the parameter defines an **explicit binding source** using attributes such as [FromRoute], [FromQuery], or [FromBody], the **parameter binds to that part** of the request.
2. If the parameter is a well-known type such as HttpContext, HttpRequest, Stream, or IFormFile, the parameter is bound to the corresponding value.

CHOOSING A BINDING SOURCE

3. If the parameter type has a **BindAsync()** method, use that method for binding.
4. If the parameter is a string or **has an appropriate TryParse()** method (so is a simple type):
   a. If the name of the parameter matches a **route parameter** name, bind to the route value.
   b. Otherwise, bind to **the query string**.

CHOOSING A BINDING SOURCE

5. If the parameter is an **array of simple types**, a string[], or StringValues, and the request is **a GET or similar HTTP verb** that **normally doesn't have a request body**, bind to the **query string**.
6. If the parameter is a **known service** type from the dependency injection container, bind by **injecting** the service from the container.
7. Finally, bind to the body by deserializing from JSON.

SIMPLIFYING HANDLERS WITH ASPARAMETERS

```
app.MapGet("/category/{id}", (int id, int page, [FromHeader(Name = "sort")]
➥ bool? sortAsc, [FromQuery(Name = "q")] string search) => { });
```

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/category/{id}",
    ([AsParameters] SearchModel model) => $"Received {model}");  ❶

app.Run();

record struct SearchModel(
    int id,                                                      ❷
    int page,                                                    ❷
    [FromHeader(Name = "sort")] bool? sortAsc,                   ❷
    [FromQuery(Name = "q")] string search);                      ❷
```

❶ [AsParameters] indicates that the constructor or properties of the type should be bound, not the type itself.

❷ Each parameter is bound as though it were written in the endpoint handler.

HANDLING USER INPUT WITH MODEL VALIDATION

- You should validate your endpoint handler parameters before you use them to do anything that touches your domain, anything that touches your infrastructure, or anything that could leak information to an attacker.
  - minimal API **filter pipeline**.

HANDLING USER INPUT WITH MODEL VALIDATION

Validation is needed to check for nonmalicious errors:

- Data should be **formatted correctly**. Email fields have a valid email format, for example.
- **Numbers** may need to be in a **particular range**. You can't buy –1 copies of this book!
- Some values may be required, but others are optional. Name may be required for a profile, but phone number is optional.
- **Values** must conform to your **business requirements**. You can't convert a currency to itself; it needs to be converted to a different currency.

USING DATAANNOTATIONS ATTRIBUTES FOR VALIDATION

- Validation attributes—more precisely, **DataAnnotations attributes**—allow you to specify **the rules that your parameters should conform to**.

USING DATAANNOTATIONS ATTRIBUTES FOR VALIDATION

```
using System.ComponentModel.DataAnnotations;                    ❶

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapPost("/users", (UserModel user) => user.ToString());     ❷
```

```
app.Run();

public record UserModel
{
    [Required]                                                  ❸
    [StringLength(100)]                                         ❹
    [Display(Name = "Your name")]                               ❺
    public string FirstName { get; set; }

    [Required]
    [StringLength(100)]
    [Display(Name = "Last name")]
    public string LastName { get; set; }

    [Required]
    [EmailAddress]                                              ❻
    public string Email { get; set; }

    [Phone]                                                     ❼
    [Display(Name = "Phone number")]
    public string PhoneNumber { get; set; }
}
```

USING DATAANNOTATIONS ATTRIBUTES FOR VALIDATION

- [**CreditCard**]—Validates that a property has a valid credit card format
- [**EmailAddress**]—Validates that a property has a valid email address format
- [**StringLength(max)**]—Validates that a string has at most max number of characters
- [**MinLength(min)**]—Validates that a collection has at least the min number of items
- [**Phone**]—Validates that a property has a valid phone number format

USING DATAANNOTATIONS ATTRIBUTES FOR VALIDATION

- [**Range(min, max)**]—Validates that a property has a value between min and max
- [**RegularExpression(regex)**]—Validates that a property conforms to the regex regular expression pattern
- [**Url**]—Validates that a property has a valid URL format
- [**Required**]—Indicates that the property must not be null
- [**Compare**]—Allows you to confirm that two properties have the same value (such as Email and ConfirmEmail)

USING DATAANNOTATIONS ATTRIBUTES FOR VALIDATION

- One common **limitation** with DataAnnotation attributes is that it's hard to **validate properties that depend on the values of other properties**.
- Implement **IValidatableObject** in your models instead of, or in addition to, using attributes.

USING DATAANNOTATIONS ATTRIBUTES FOR VALIDATION

```
using System.ComponentModel.DataAnnotations;

public record CreateUserModel : IValidatableObject                ❶
{
    [EmailAddress]                                                ❷
    public string Email { get; set; }

    [Phone]                                                       ❷
    public string PhoneNumber { get; set; }

    public IEnumerable<ValidationResult> Validate(                ❸
        ValidationContext validationContext)                      ❸
    {
        if(string.IsNullOrEmpty(Email)                            ❹
            && string.IsNullOrEmpty(PhoneNumber))                 ❹
        {
            yield return new ValidationResult(                    ❺
                "You must provide an Email or a PhoneNumber",     ❺
                New[] { nameof(Email), nameof(PhoneNumber) });    ❺
        }
    }
}
```

❶ Implements the IValidatableObject interface

❷ The DataAnnotation attributes continue to validate basic format requirements.

❸ Validate is the only function to implement in IValidatableObject.

❹ Checks whether the object is valid . . .

❺ . . . and if not, returns a result describing the error

ADDING A VALIDATION FILTER TO YOUR MINIMAL APIS

- Microsoft decided **not to include any dedicated validation** APIs in minimal APIs. By contrast, validation is a built-in core feature of Razor Pages and MVC.
  - Consequently, validation in minimal APIs typically relies on the filter pipeline.
- NuGet package called MinimalApis.Extensions provides the filter for you.
  - extension method called WithParameterValidation() that you can add to your endpoints.

ADDING A VALIDATION FILTER TO YOUR MINIMAL APIS

After the **UserModel** is bound to the JSON body of the request, the validation filter executes as part of the filter pipeline.

- If the user parameter is valid, execution passes to the endpoint handler.
- If the parameter is invalid, a 400 Bad Request Problem Details response is returned containing a description of the errors,

```
using System.ComponentModel.DataAnnotations;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapPost("/users", (UserModel user) => user.ToString())
        .WithParameterValidation();     ❶

app.Run();
```

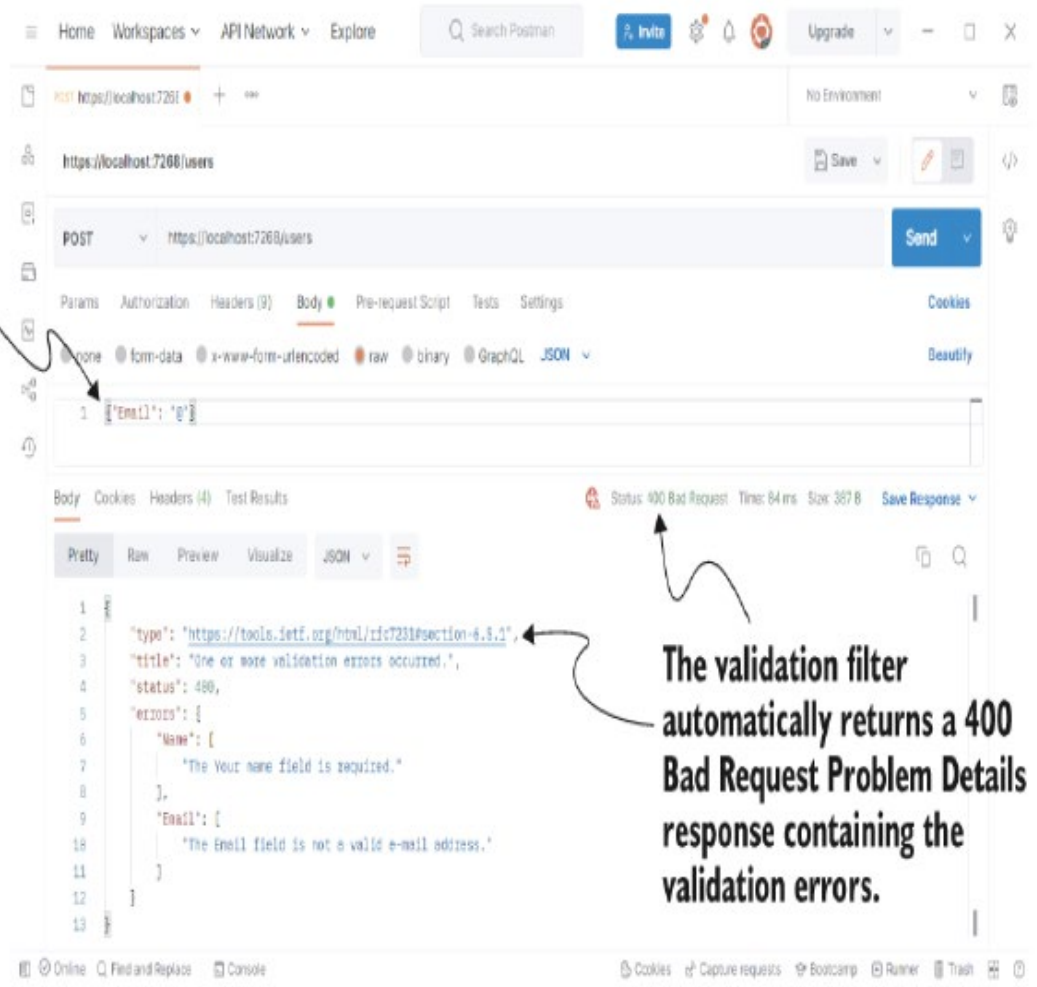
```
public record UserModel                 ❷
{
    [Required]
    [StringLength(100)]
    [Display(Name = "Your name")]
    public string Name { get; set; }

    [Required]
    [EmailAddress]
    public string Email { get; set; }
}
```

❶ Adds the validation filter to the endpoint

❷ The UserModel defines its validation requirements using DataAnnotations attributes.

This example sends invalid data in the body of the request.



The validation filter automatically returns a 400 Bad Request Problem Details response containing the validation errors.

ADDING A VALIDATION FILTER TO YOUR MINIMAL APIS

```
using System.ComponentModel.DataAnnotations;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapPost("/user/{id}",
    ([AsParameters] GetUserModel model) => $"Received {model.Id}")  ❶
     .WithParameterValidation();                                    ❷

app.Run();

struct GetUserModel
{
    [Range(1, 10)]                                                  ❸
    Public int Id { get; set; }                                     ❸
}
```

❶ Uses [AsParameters] to create a type than can be validated

❷ Adds the validation filter to the endpoint

❸ Adds validation attributes to your simple types