

◆ SECTION 6: MODEL BINDING & VALIDATION (ASP.NET Core) ◆

⚠ This section is extremely important for interviews because:

- Many real production bugs happen here
- Interviewers use this to judge API maturity
- Security + correctness + usability all connect here

1 What is Model Binding?

🧠 Model binding is how ASP.NET Core takes data from a request and puts it into your C# objects.

Example:

```
{  
    "name": "Ashik",  
    "age": 28  
}
```

ASP.NET Core automatically creates:

```
Employee { Name = "Ashik", Age = 28 }
```

You don't write this mapping manually — model binding does it.

⌚ Model binding is the process where ASP.NET Core:

- Reads data from request sources
- Converts data to .NET types
- Populates action parameters and models

Sources include:

- **Route values**
- **Query string**
- **Headers**
- **Body**
- **Form data**

📌 Happens **before controller action executes**.

➡ Q: Does model binding happen before filters?

✓ Before action filters, after authorization filters

2 Model Binding Sources (VERY IMPORTANT)

◆ Route Data

```
[HttpGet("{id}")]  
public IActionResult Get(int id)
```

URL: /api/employees/10

📌 Route binding has highest priority.

◆ Query String

```
public IActionResult Get([FromQuery] int page)
```

URL: /api/employees?page=2

◆ Request Body

```
public IActionResult Create([FromBody] EmployeeDto dto)
```

📌 Only one [FromBody] parameter allowed per action.

◆ Headers

```
public IActionResult Get([FromHeader(Name="x-api-key")] string apiKey)
```

◆ Form Data

Used in file uploads & HTML forms.

💡 Q: What if [FromX] is not specified?

✓ ASP.NET Core uses default binding rules

3 Binding Priority Order (INTERVIEW FAVORITE)

Order: 1 Route → 2 Query → 3 Body → 4 Headers

📌 If same name exists in multiple places, route wins.

4 What is Model Validation?

🧠 Validation checks: “Is the incoming data correct and safe?”

Example:

- Required fields
- Max length
- Range checks

🎯 Model validation:

- Runs after model binding
- Uses Data Annotations or custom validators
- Populates `ModelState`

If validation fails:

- `ModelState.IsValid = false`
- ASP.NET Core can automatically return 400 Bad Request

📌 **Validation prevents bad data reaching business logic.**

5 Data Annotations (COMMON INTERVIEW AREA)

Examples:

```
public class EmployeeDto
{
    [Required]
    public string Name { get; set; }

    [Range(18, 60)]
    public int Age { get; set; }

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

📌 **“Data annotations provide declarative validation rules.”**

6 Automatic Model Validation (ApiController)

💡 ASP.NET Core checks validation for you automatically.

🎯 When `[ApiController]` is used:

- `ModelState` is automatically validated
- Returns **400 with error details**
- No need to check `ModelState.IsValid` manually

📌 This improves consistency and reduces boilerplate.

❓ Q: Can we disable automatic validation?

✓ Yes, via `ApiBehaviorOptions`

7 Custom Validation (ADVANCED)

◆ `IValidatableObject`

```
public class EmployeeDto : IValidatableObject
{
    public int Age { get; set; }

    public IEnumerable<ValidationResult> Validate(ValidationContext context)
    {
        if (Age < 18)
            yield return new ValidationResult("Age must be >= 18");
    }
}
```

◆ `Custom Validation Attribute`

```
public class EvenNumberAttribute : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
ValidationContext context)
    {
        if ((int)value % 2 != 0)
            return new ValidationResult("Number must be even");
        return ValidationResult.Success;
    }
}
```

📌 Custom validators handle business-specific rules.

8 ModelState (VERY IMPORTANT)

ModelState stores:

- Binding errors
- Validation errors

ModelState:

- Is a dictionary
- Contains field-level errors
- Used to generate validation responses

 With [\[ApiController\]](#), framework checks it automatically.

9 Common Production Bugs (INTERVIEW GOLD)

-  Missing [FromBody] → Causes null model
-  Multiple [FromBody] → Causes runtime exception
-  Trusting client input → Security risk
-  Not validating nested objects → Leads to inconsistent data

10 Best Practices (INTERVIEW MUST SAY)

- Use DTOs, not entities
- Always validate input
- Keep validation close to models
- Return meaningful validation errors
- Never trust client input

ONE-LINE INTERVIEW ANSWER (MEMORIZE)

“Model binding maps request data to action parameters, and model validation ensures incoming data meets defined rules before business logic executes.”

YOU NOW UNDERSTAND

- | | | |
|-------------------------|---------------------|-----------------------|
| ✓ Model binding sources | ✓ Binding priority | ✓ Validation pipeline |
| ✓ Data annotations | ✓ Custom validation | ✓ ModelState behavior |
| ✓ Real-world bugs | | |

HANDS-ON 6: MODEL BINDING & VALIDATION

STEP 0: Create / Use Project

```
dotnet new webapi -n BindingValidationDemo  
cd BindingValidationDemo
```

STEP 1: Create DTO (BEST PRACTICE)

Create folder: Dtos

```
public class CreateEmployeeDto  
{  
    [Required]  
    [StringLength(50)]  
    public string Name { get; set; }  
    [Range(18, 60)]  
    public int Age { get; set; }  
    [EmailAddress]  
    public string Email { get; set; }  
}
```

📌 “We use DTOs instead of entities to protect domain models and control input.”

STEP 2: Create Controller

Controllers/EmployeesController.cs

```
[ApiController]  
[Route("api/employees")]  
public class EmployeesController : ControllerBase  
{  
    [HttpPost]  
    public IActionResult Create(CreateEmployeeDto dto)  
    {  
        return Ok("Employee created");  
    }  
}
```

📌 No **[FromBody]** needed because **[ApiController]** infers it

STEP 3: Test VALID Request

Request:

```
{  
    "name": "Ashik",  
    "age": 28,  
    "email": "ashik@test.com"  
}
```

Response: **200 OK**

 Model binding + validation passed successfully.

STEP 4: Test INVALID Request (Automatic Validation)

Request:

```
Json  
{  
    "name": "",  
    "age": 15,  
    "email": "wrong-email"  
}
```

Response: **400 Bad Request**

Body:

```
Json  
{  
    "errors": {  
        "Name": ["The Name field is required."],  
        "Age": ["The field Age must be between 18 and 60."],  
        "Email": ["The Email field is not a valid e-mail address."]
    }
}
```

 **[ApiController]** automatically validates ModelState and returns 400.

STEP 5: Disable Automatic Validation (Interview Scenario)

In Program.cs:

```
builder.Services.Configure<ApiBehaviorOptions>(options =>
{
    options.SuppressModelStateInvalidFilter = true;
});
```

Now update controller:

```
[HttpPost]
public IActionResult Create(CreateEmployeeDto dto)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    return Ok("Employee created");
}
```

 **Manual validation gives more control over error response.**

STEP 6: Binding from Route, Query, Header

```
[HttpGet("{id}")]
public IActionResult GetEmployee(
    [FromRoute] int id,
    [FromQuery] string department,
    [FromHeader(Name = "x-request-id")] string requestId)
{
    return Ok(new { id, department, requestId });
}
```

Test URL:

```
GET /api/employees/10?department=IT
```

Header:

```
x-request-id: abc123
```

 **“ASP.NET Core can bind data from multiple sources in one action.”**

STEP 7: Binding Priority DEMO

Request:

```
GET /api/employees/5?id=99
```

Result:

```
id = 5
```

⚠ Route data has higher priority than query string.

STEP 8: Multiple [FromBody] ✖ (COMMON MISTAKE)

```
public IActionResult Create(  
    [FromBody] CreateEmployeeDto dto,  
    [FromBody] string extra)
```

⚠ Result:

```
InvalidOperationException
```

⚠ Only one parameter can be bound from request body.

STEP 9: Custom Validation (Business Rule)

Using `IValidatableObject`

```
public class CreateEmployeeDto : IValidatableObject  
{  
    public int Age { get; set; }  
  
    public IEnumerable<ValidationResult> Validate(ValidationContext context)  
    {  
        if (Age % 2 != 0)  
        {  
            yield return new ValidationResult(  
                "Age must be even number",  
                new[] { nameof(Age) });  
        }  
    }  
}
```

⚠ Use this when validation depends on multiple fields or business rules.

STEP 10: Nested Model Validation (IMPORTANT)

```
public class AddressDto
{
    [Required]
    public string City { get; set; }
}

public class CreateEmployeeDto
{
    [Required]
    public AddressDto Address { get; set; }
}
```

 **Nested objects are validated automatically if decorated.**

STEP 11: Security Best Practices (INTERVIEW MUST)

- Never trust client input
- Validate everything
- Limit string lengths
- Avoid binding entities directly
- Prevent over-posting

FINAL INTERVIEW SUMMARY (MEMORIZE)

“ASP.NET Core model binding maps request data from routes, query strings, headers, and body into action parameters, while validation ensures data correctness using data annotations and custom rules before business logic executes.”

YOU NOW MASTER (HANDS-ON)

- ✓ Automatic & manual validation ✓ Binding from all sources ✓ Binding priority
✓ Common runtime errors ✓ Custom business validation ✓ Security best practices