Data Transfer Objects (DTOs)

A **Data Transfer Object (DTO)** is a simple class used to transfer data between layers of an application (e.g., between the controller and the client). DTOs:

- Contain only data and no business logic.
- Are used to control what data is exposed to the client.
- Help decouple internal models from external APIs.

Why Use DTOs?

DTOs are helpful for organizing how your API communicates with clients. Here's why you need them, explained simply:

1. **Data Shaping**
   - Imagine your database stores a lot of details about a product: name, price, internal codes, supplier info, etc.
   - If the client only needs the **name** and **price**, you use a DTO to "shape" (select) just those fields to send.
   - **Example**:
     Instead of sending everything:

```
{

    "id": 1,

    "name": "Laptop",

    "price": 1500,

    "internalCode": "INT123",

    "supplier": "TechCorp"

}
```

You send only this:

```
{
	"name": "Laptop",
	"price": 1500
}
```

Security

- By hiding unnecessary fields, you prevent sensitive information (like `internalCode` or `password`) from being exposed to users.
- **Example**: If your API accidentally sends internal codes or admin data, someone might misuse it. DTOs protect this.

Validation

- DTOs help ensure the data sent by the client is correct before saving it in your database.
- **Example**: You can add rules like “price cannot be negative” or “name is required” in the DTO. If the client sends invalid data, the API will reject it.

Decoupling

- **Decoupling** means separating your internal database structure from what the client sees.
- If your database structure changes later (e.g., you add/remove columns), your API doesn’t break because the client only knows the DTO structure.
- **Example**: A database column `ProductCost` might change to `ProductPrice`, but the client only cares about `price` defined in the DTO.

Example of Using DTOs in .NET Core Web API

Scenario

Suppose you have a database model `Product` and you want to expose only specific fields (e.g., `Name` and `Price`) in your API.

1. **Database Model**

```
public class Product
{
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
        public string InternalCode { get; set; } // Should not be exposed
}
```

DTO Class

```
public class ProductDto

{

        public string Name { get; set; }

        public decimal Price { get; set; }

}
```

Controller Example:

```
[ApiController]

[Route("api/products")]

public class ProductsController : ControllerBase

{           private readonly List<Product> _products = new()

            {           new Product { Id = 1, Name = "Laptop", Price = 1500, InternalCode = "INT123" },

            new Product { Id = 2, Name = "Phone", Price = 800, InternalCode = "INT456" }

            };

            [HttpGet]

            public ActionResult<IEnumerable<ProductDto>> GetProducts()

            {           // Map Product to ProductDto

            var productDtos = _products.Select(p => new ProductDto

            {           Name = p.Name,                  Price = p.Price           }).ToList();

            return Ok(productDtos);       }}
```

Explanation

1. **Model**: `Product` represents the actual database entity with fields like `Id` and `InternalCode`.
2. **DTO**: `ProductDto` contains only the fields you want to expose (`Name` and `Price`).
3. **Controller**: Maps the `Product` model to `ProductDto` before sending it to the client.

Output of the API

When a client makes a `GET /api/products` request, the response will look like this:

```
[
        {
        "name": "Laptop",
        "price": 1500
        },
        {
        "name": "Phone",
        "price": 800
        }
]
```

Notice: Fields like `Id` and `InternalCode` are not exposed.

Benefits of Using DTOs

1. **Improves Security**: Hides sensitive or irrelevant fields.
2. **Flexible API**: Makes it easier to customize API responses.
3. **Reduces Coupling**: Keeps your internal models separate from API contracts.
4. **Better Validation**: DTOs can have validation attributes to ensure incoming data is valid.

What is AutoMapper?

AutoMapper is a tool that makes it easier to convert one object to another, like converting a **database model** into a **DTO**. Instead of writing repetitive code to "map" fields, AutoMapper does it for you.

Without AutoMapper

You have to manually map fields like this:

```
var productDto = new ProductDto

{

    Name = product.Name,

    Price = product.Price

};
```

With AutoMapper

AutoMapper simplifies this by automatically mapping fields with the same names:

1. **Setup AutoMapper**
   Create a "mapping profile" to tell AutoMapper how to map your objects:

```
public class MappingProfile : Profile

{

    public MappingProfile()

    {

    CreateMap<Product, ProductDto>();

    }

}
```

Use AutoMapper in Code

Use AutoMapper to map your list of `Product` objects to `ProductDto` objects:

var productDtos = _mapper.Map<IEnumerable<ProductDto>>(products);

AutoMapper will handle the mapping for you, saving you from writing repetitive code for every DTO.

Why Use AutoMapper?

1. **Saves Time**: No need to manually write mapping code for every property.
2. **Less Error-Prone**: Prevents mistakes in mapping (e.g., forgetting to map a field).
3. **Cleaner Code**: Your code is shorter and easier to maintain.