📘Understanding Domain Models in ASP.NET Core Web API

1. 🔍What is a Domain Model?

A **domain model** represents the core business concepts and rules of your application. It encapsulates data and behavior related to the domain (business logic).

Example Entities:

- E-commerce: `Product`, `Order`, `Customer`
- Banking: `Account`, `Transaction`, `User`

> Domain models are not DTOs. They include business logic and are used within the application, not exposed directly to the outside world.

---

2. 🧱Components of Domain Modeling

a. Entities

Objects with a distinct identity that persist over time.

```
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Order> Orders { get; set; }
}
```

b. Value Objects

Objects that have no identity and are defined only by their attributes.

```
public class Address
{
    public string Street { get; set; }
    public string City { get; set; }
    public string ZipCode { get; set; }
}
```

c. Aggregates & Root

An **aggregate** is a cluster of domain objects treated as a single unit. The **aggregate root** is the entry point.

```
public class Order
{
    public int Id { get; set; }
    public Customer Customer { get; set; }
    public List<OrderItem> Items { get; set; }

    public decimal Total => Items.Sum(i => i.Price * i.Quantity);
}
```

---

3. 🕐Interaction Flow: Controller to Domain Model

```
Client (Request: JSON)
  ↓
Controller (Receives DTO)
  ↓
Application Service (Maps DTO → Domain Model)
  ↓
Domain Model (Executes business logic)
  ↓
Service (persists to DB or external system)
  ↓
Controller (Returns DTO response)
```

---

4. 🧰Example with Code and Flow

🕰️CreateOrderDto (DTO)

```
public class CreateOrderDto
{
    public int CustomerId { get; set; }
    public List<int> ProductIds { get; set; }
}
```

🕰️OrderService (App Layer)

```
public class OrderService
{
    public Order CreateOrder(CreateOrderDto dto)
    {
        var customer = _repo.GetCustomer(dto.CustomerId);
        var products = _repo.GetProducts(dto.ProductIds);

        var order = new Order { Customer = customer };
        order.AddItems(products);
        _repo.SaveOrder(order);
        return order;
    }
}
```

🕰️OrdersController

```
[HttpPost]
public IActionResult CreateOrder(CreateOrderDto dto)
{
    var order = _orderService.CreateOrder(dto);
    return CreatedAtAction(nameof(GetOrder), new { id = order.Id }, order);
}
```

---

5. Benefits of Using Domain Models

Benefit	Description
Separation of Concerns	Keeps business logic out of controllers
Reusability	Models and rules reusable across layers
Maintainability	Easier to update business rules
Testability	Domain logic is testable in isolation
Security	Keeps internal logic away from API surface

---

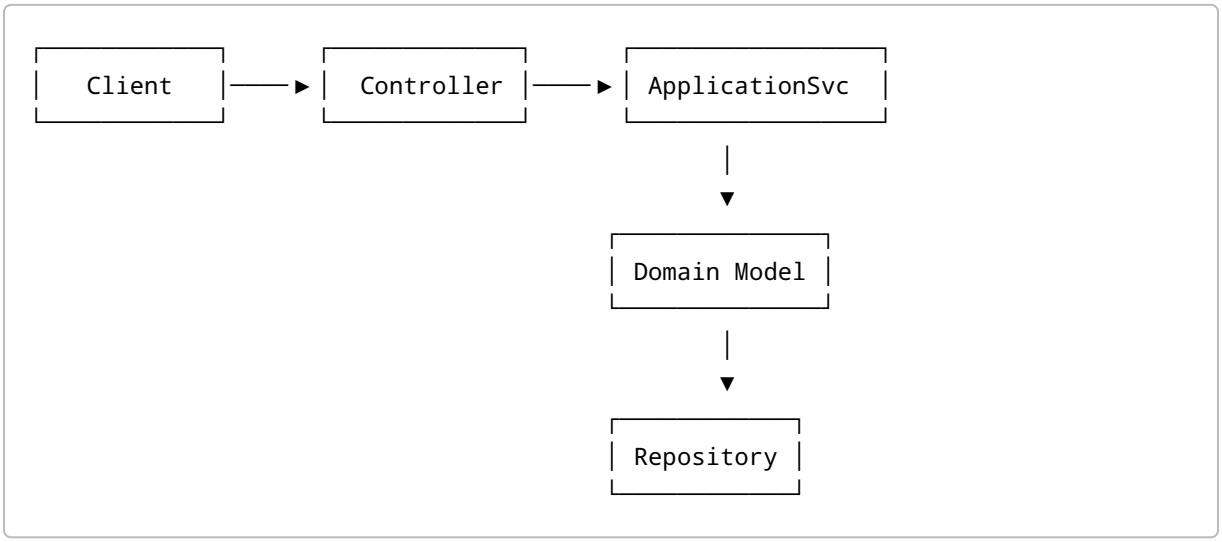
6. 🔧Best Practices

- Use **DTOs** to interact with the outside world.
- Keep **business logic in domain models**, not controllers.

- Create **application services** to mediate between controllers and domain.
- **Map** DTOs to/from domain models explicitly or with AutoMapper.

---

7. 📊Visual Flow Diagram



🔗Summary

- Domain models encapsulate your **core business logic**.
- They are **not** your API contracts (DTOs).
- Controllers **delegate** to services that use domain models.
- This pattern ensures **scalability**, **clarity**, and **maintainability**.