//// ✅ 23. Refactorización en ASP.NET Core

/\*

📌 ¿Qué es refactorizar?

- Reestructurar el código sin cambiar su comportamiento externo.

- Objetivo: mayor legibilidad, mantenibilidad y escalabilidad.

- Evita duplicidad de código y mezcla de responsabilidades.

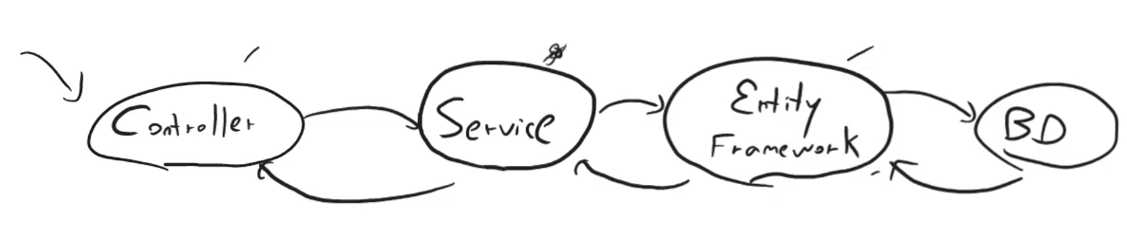
\*/

/// 🔹 Capas y responsabilidades

- \*\*Controller\*\* → Maneja solicitudes y respuestas HTTP (ActionResult, códigos 200/404/201, etc.).

- \*\*Service\*\* → Contiene reglas de negocio y lógica con la BD (vía EF Core).

- \*\*Entity Framework (Context)\*\* → Acceso real a la base de datos.



/// 🔹 Interface de Servicio

- Define contrato con métodos asíncronos (Task).

- Métodos típicos: GetAll, GetById, Add, Update, Delete.

- NO devuelve ActionResult → eso es responsabilidad del Controller.

- Se implementa en la clase de servicio y se inyecta en Program.cs.

/// 🔹 Inyección de Dependencias

```csharp

builder.Services.AddScoped<IBeerService, BeerService>();

* El Controller recibe la interfaz en el constructor:

private readonly IBeerService \_service;

public BeersController(IBeerService service) => \_service = service;

/// 🔹 Refactorización CRUD

* **Read**: Controller → invoca \_service.GetAll / GetById.
* **Create**: Controller valida → Service crea entidad, guarda y retorna DTO con ID.
* **Update**: Service actualiza entidad; Controller maneja null (NotFound) u Ok.
* **Delete**: Service elimina y retorna DTO borrado; Controller devuelve Ok(dto) o NotFound.

/// 🔹 Generics en Interfaces

* Permiten crear un contrato reutilizable para varios servicios.

public interface IService<TDto, TInsert, TUpdate>

{

Task<IEnumerable<TDto>> Get();

Task<TDto?> GetById(int id);

Task<TDto> Add(TInsert dto);

Task<TDto?> Update(int id, TUpdate dto);

Task<TDto?> Delete(int id);

}

* Cada servicio implementa con sus propios DTOs (ej: BeerDto, BeerInsertDto, BeerUpdateDto).
* Ventaja: menos interfaces duplicadas y mayor reutilización.