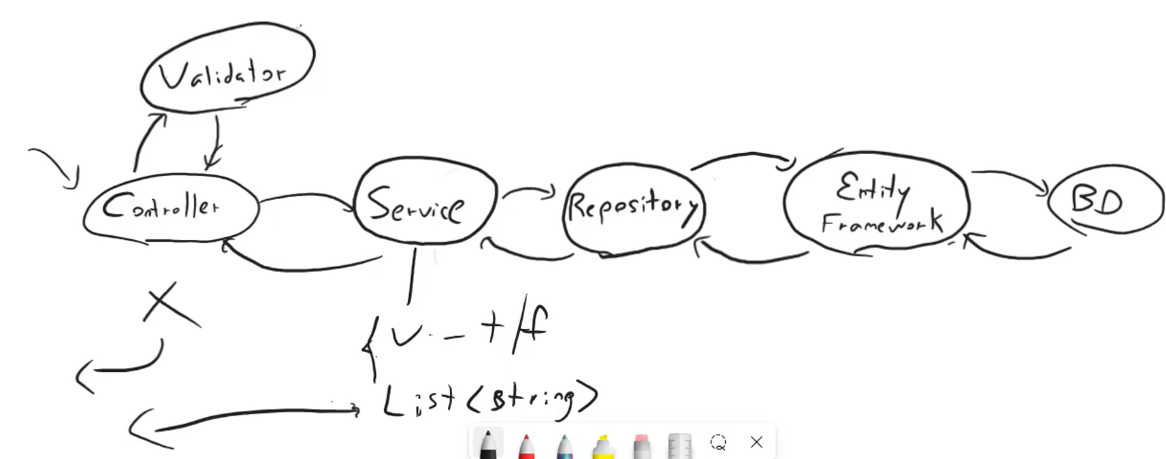
**📌 Sección 14 – Manejo de Errores**



**🔹 76. Propuestas para el manejo de errores**

* Los **errores de negocio** (ej: venta sin inventario, nombre repetido de cerveza) no siempre deben manejarse como excepciones.
* **Alternativas posibles:**
  1. **Excepciones en el servicio:** lanzar throw y capturar en el controlador con try-catch. Útil para errores excepcionales (BD caída, fallo de servicio). No ideal para reglas de negocio porque afecta el rendimiento.
  2. **Capa intermedia de validaciones:** entre controlador y servicio. Separa reglas de negocio (ej: descuentos, unicidad en BD) de validaciones de formato.
  3. **Validaciones en el validador existente:** mezclar validaciones de formato con reglas de negocio. Problema: muchas veces hay que consultar repositorio.
  4. **En la capa de servicio (opción elegida):**
     + Método Validate() que devuelve true/false.
     + Lista pública Errors con mensajes descriptivos.
     + El controlador consulta Validate().
       - Si falla: retorna 400 Bad Request + detalle de errores.
       - Si pasa: continúa con la operación.

**🔹 77. Definición de métodos de validación**

* Se agregan métodos Validate() en la **interfaz del servicio** (ICommandService).
* Se usa **sobrecarga**:
  + Validate(BeerInsertDto dto)
  + Validate(BeerUpdateDto dto)
* También se define la propiedad pública:

public List<string> Errors { get; }

* Inicialmente los métodos devuelven true.
* En el **controlador** se usa:

if (!service.Validate(dto)) return BadRequest(service.Errors);

* Esto permite que cada servicio concrete implemente sus propias reglas.

**🔹 78. Método de búsqueda en Repositorio**

* Regla de negocio: **nombre de cerveza no debe repetirse**.
* No se usa constraint UNIQUE porque no sirve con **borrado lógico**.
* Se implementa un método **Search** en repositorio:

IEnumerable<TEntity> Search(Func<TEntity, bool> filter);

* Este método recibe un **Func** (función de primera clase) que define la condición (WHERE) y devuelve los registros que cumplen.
* Se declara en la **interfaz IRepository**, así los servicios pueden acceder sin importar la implementación.
* Ventaja: búsqueda **genérica y reutilizable**, no atada a un campo fijo.

**🔹 79. Implementación en capa Servicio**

* En BeerService, Validate() usa Search() para verificar unicidad:

**✔️ Insert**

var result = \_repository.Search(b => b.Name == dto.Name);

if (result.Count() > 0) {

Errors.Add("No puede existir una cerveza con un nombre ya existente.");

return false;

}

return true;

**✔️ Update**

* Similar al Insert, pero se **ignora el mismo registro** (mismo Id):

var result = \_repository.Search(b => b.Name == dto.Name && b.Id != dto.Id);

if (result.Count() > 0) {

Errors.Add("No puede existir una cerveza con un nombre ya existente.");

return false;

}

return true;

**✔️ Controlador**

* No se modifica: solo invoca Validate().
* Si falla → retorna **400 + lista de errores**.
* Si pasa → ejecuta normalmente (200 OK).

**✅ Conclusiones**

* Existen **múltiples enfoques** para manejar errores de negocio.
* En este caso, se implementa en la **capa de servicio**, centralizando las reglas y retornando errores detallados.
* Se usa:
  + Validate() con sobrecarga.
  + Errors como lista de mensajes.
  + Search(Func) en repositorio para búsquedas dinámicas.
* Ventaja: **claridad, flexibilidad y reutilización**.
* Es solo una alternativa: la elección depende de la arquitectura y políticas de la empresa.