**¿Qué es MediatR?**

**MediatR** es una biblioteca de .NET que implementa el patrón **Mediator**, el cual se basa en **desacoplar** las partes del sistema que se comunican entre sí.

En lugar de que un **controlador** llame directamente a una clase de servicio, este envía un **mensaje** (llamado Request) al **Mediador**, y este se encarga de buscar quién maneja ese mensaje (Handler) y ejecutar la lógica correspondiente.

**¿Qué resuelve?**

* Desacopla el controlador de la lógica de negocio.
* Hace que las acciones sean independientes y fáciles de probar.
* Centraliza la lógica de flujo, permitiendo agregar validaciones, logs, etc., con **pipelines**.

**Conceptos Clave (con analogías prácticas)**

| **Concepto** | **Explicación práctica** |
| --- | --- |
| **Request** | Una **orden** o **pregunta**. Puede ser algo como “Registrar un empleado” o “Obtener todos los clientes”. |
| **Handler** | El **encargado** de ejecutar esa orden o responder esa pregunta. |
| **Response** | Lo que el handler devuelve: un resultado, mensaje, lista, error, etc. |
| **Command** | Una orden que **cambia el estado** del sistema. Ej: crear, actualizar, eliminar. |
| **Query** | Una pregunta que **solo consulta** datos. Ej: listar, buscar. |
| **Pipeline** | Una **cadena de procesos** que se ejecutan antes o después del handler (como validación, logging, manejo de errores, etc). |
| **TRequest** y **TResponse** | Son simplemente **tipos genéricos**: el tipo del mensaje y el tipo que devuelve. Lo explico más abajo con código. |

**Ejemplo mínimo con MediatR**

**Escenario: Quiero registrar un nuevo producto**

**Paso 1: Crear el Command (Request)**

public class CrearProductoCommand : IRequest<string>

{

public string Nombre { get; set; }

public decimal Precio { get; set; }

}

IRequest<string> indica que este mensaje espera como respuesta un string (por ejemplo, “Producto creado con ID 45”).

**Paso 2: Crear el Handler**

public class CrearProductoHandler : IRequestHandler<CrearProductoCommand, string>

{

public async Task<string> Handle(CrearProductoCommand request, CancellationToken cancellationToken)

{

// Aquí iría tu lógica de guardar en la base de datos

var id = Guid.NewGuid().ToString();

Console.WriteLine($"Guardado producto {request.Nombre} con precio {request.Precio}");

return await Task.FromResult($"Producto creado con ID {id}");

}

}

Handle ejecuta la lógica para ese mensaje. Lee el request, actúa, y devuelve el resultado.

**Paso 3: Usarlo desde el Controller**

[ApiController]

[Route("api/[controller]")]

public class ProductosController : ControllerBase

{

private readonly IMediator \_mediator;

public ProductosController(IMediator mediator)

{

\_mediator = mediator;

}

[HttpPost]

public async Task<IActionResult> CrearProducto([FromBody] CrearProductoCommand command)

{

var resultado = await \_mediator.Send(command);

return Ok(resultado);

}

}

**¿Y los Queries?**

Son igual que los Commands, pero no modifican nada, solo consultan:

public class ObtenerProductosQuery : IRequest<List<ProductoDto>> { }

public class ObtenerProductosHandler : IRequestHandler<ObtenerProductosQuery, List<ProductoDto>>

{

public async Task<List<ProductoDto>> Handle(ObtenerProductosQuery request, CancellationToken cancellationToken)

{

var productos = new List<ProductoDto>

{

new ProductoDto { Nombre = "Laptop", Precio = 2500 },

new ProductoDto { Nombre = "Teclado", Precio = 150 }

};

return await Task.FromResult(productos);

}

}

**¿Qué es el Pipeline?**

Un **Pipeline** son pasos extra que ocurren alrededor del handler, como si fueran **filtros**.

Por ejemplo:

* Validar los datos del request.
* Registrar logs.
* Medir rendimiento.
* Manejar errores.

Ejemplo usando **FluentValidation**:

public class CrearProductoValidator : AbstractValidator<CrearProductoCommand>

{

public CrearProductoValidator()

{

RuleFor(x => x.Nombre).NotEmpty();

RuleFor(x => x.Precio).GreaterThan(0);

}

}

Para que eso se aplique automáticamente, MediatR permite usar IPipelineBehavior<TRequest, TResponse> para agregar lógica antes/después del handler.

**¿Qué es TRequest y TResponse?**

Solo son genéricos, como decir:

IRequest<string> // El Request espera un string como respuesta.

IRequest<Empleado> // El Request espera un Empleado.

IRequestHandler<CrearProductoCommand, string> // Maneja ese tipo de comando y devuelve string.

**¿Qué necesito en mi proyecto?**

**Instalar NuGet:**

dotnet add package MediatR.Extensions.Microsoft.DependencyInjection

Y en Program.cs:

builder.Services.AddMediatR(cfg =>

cfg.RegisterServicesFromAssembly(typeof(Program).Assembly));

**Recomendaciones**

1. **Crea carpetas**: Commands, Queries, Handlers, DTOs.
2. **Nombrado claro**: Usa nombres como CrearProductoCommand, ObtenerClientesQuery.
3. **Cada handler debe hacer solo una cosa**.

**¿Seguimos con un ejemplo completo basado en tu ERP?**

Puedo crear uno como:

* CrearEmpleadoCommand
* ObtenerEmpleadosQuery
* ActualizarClienteCommand
* Y usar FluentValidation + PipelineBehavior.

Perfecto, vamos a hacerlo paso a paso como propones. 💪  
Vamos a trabajar en **tres versiones** del proyecto. Aquí va la planificación y luego comenzamos con la **Versión 1**:

## 🧱 PLAN DEL PROYECTO

### ✅ ****Versión 1:****

Guardar un empleado (simulado, sin base de datos real) usando MediatR.

* 🎯 Crear empleado usando un Command + Handler.
* 🎯 Llamarlo desde un Controller.

### 🔜 ****Versión 2:****

Agregar la consulta de empleados:

* 🎯 Query para obtener la lista de empleados (simulada, en memoria).
* 🎯 Otro endpoint en el controller.

### 🔜 ****Versión 3:****

Agregar validación y pipeline:

* 🎯 FluentValidation para validar el empleado antes de guardar.
* 🎯 PipelineBehavior para aplicar validación automáticamente.

# 🚀 VERSIÓN 1: Crear Empleado (Simulado)

## 1️⃣ Crear el proyecto

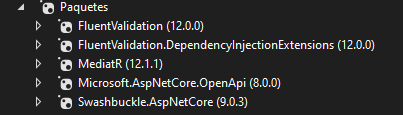
Abre la terminal o consola en tu carpeta de trabajo y ejecuta:

dotnet new webapi -n EmpleadoApp

cd EmpleadoApp

## 2️⃣ Instalar paquete MediatR

dotnet add package MediatR.Extensions.Microsoft.DependencyInjection



## 3️⃣ Estructura de carpetas

Crea estas carpetas:

📁 EmpleadoApp

┣ 📁 Features

┃ ┗ 📁 Empleados

┃ ┣ 📁 Commands

┃ ┗ 📁 Handlers

## 4️⃣ Crear clase CrearEmpleadoCommand

📄 Ruta: Features/Empleados/Commands/CrearEmpleadoCommand.cs

using MediatR;

namespace EmpleadoApp.Features.Empleados.Commands

{

public class CrearEmpleadoCommand : IRequest<string>

{

public string Nombre { get; set; }

public string Cargo { get; set; }

public decimal Salario { get; set; }

}

}

## 5️⃣ Crear CrearEmpleadoHandler

📄 Ruta: Features/Empleados/Handlers/CrearEmpleadoHandler.cs

using MediatR;

using EmpleadoApp.Features.Empleados.Commands;

namespace EmpleadoApp.Features.Empleados.Handlers

{

public class CrearEmpleadoHandler : IRequestHandler<CrearEmpleadoCommand, string>

{

public async Task<string> Handle(CrearEmpleadoCommand request, CancellationToken cancellationToken)

{

// Simulamos guardar el empleado

var idGenerado = Guid.NewGuid().ToString();

Console.WriteLine($"Empleado creado: {request.Nombre}, Cargo: {request.Cargo}, Salario: {request.Salario}");

return await Task.FromResult($"Empleado creado con ID: {idGenerado}");

}

}

}

## 6️⃣ Registrar MediatR en Program.cs

📄 Edita el archivo Program.cs así:

using MediatR;

using System.Reflection;

var builder = WebApplication.CreateBuilder(args);

// Agrega MediatR

builder.Services.AddMediatR(cfg =>

cfg.RegisterServicesFromAssembly(Assembly.GetExecutingAssembly()));

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

app.UseAuthorization();

app.MapControllers();

app.Run();

## 7️⃣ Crear el EmpleadoController

📄 Ruta: Controllers/EmpleadoController.cs

using EmpleadoApp.Features.Empleados.Commands;

using MediatR;

using Microsoft.AspNetCore.Mvc;

namespace EmpleadoApp.Controllers

{

[ApiController]

[Route("api/[controller]")]

public class EmpleadoController : ControllerBase

{

private readonly IMediator \_mediator;

public EmpleadoController(IMediator mediator)

{

\_mediator = mediator;

}

[HttpPost]

public async Task<IActionResult> CrearEmpleado([FromBody] CrearEmpleadoCommand command)

{

var resultado = await \_mediator.Send(command);

return Ok(resultado);

}

}

}

## ✅ Probando la API

Puedes correr el proyecto y probar con **Swagger** o Postman:

### Endpoint:

POST /api/empleado

### JSON de ejemplo:

{

"nombre": "Ana López",

"cargo": "Contadora",

"salario": 3000

}

### Respuesta esperada:

"Empleado creado con ID: 8e347cf5-64d4-482a-aabb-bf0f5c12df20"

# ✅ VERSIÓN 2: Consultar Lista de Empleados

## 🎯 Objetivo

* Agregar una **Query** para obtener empleados.
* Implementar su **Handler** con datos simulados.
* Agregar un nuevo **endpoint GET** en el EmpleadoController.

## 1️⃣ Crear carpeta Queries

Dentro de Features/Empleados, crea esta nueva carpeta:

📁 Features

┗ 📁 Empleados

┣ 📁 Commands

┣ 📁 Handlers

┗ 📁 Queries

## 2️⃣ Crear clase ObtenerEmpleadosQuery

📄 Ruta: Features/Empleados/Queries/ObtenerEmpleadosQuery.cs

using MediatR;

namespace EmpleadoApp.Features.Empleados.Queries

{

public class ObtenerEmpleadosQuery : IRequest<List<EmpleadoDto>>

{

}

public class EmpleadoDto

{

public string Nombre { get; set; }

public string Cargo { get; set; }

public decimal Salario { get; set; }

}

}

🎯 Este Query no recibe parámetros, y devuelve una lista de EmpleadoDto.

## 3️⃣ Crear ObtenerEmpleadosHandler

📄 Ruta: Features/Empleados/Handlers/ObtenerEmpleadosHandler.cs

using EmpleadoApp.Features.Empleados.Queries;

using MediatR;

namespace EmpleadoApp.Features.Empleados.Handlers

{

public class ObtenerEmpleadosHandler : IRequestHandler<ObtenerEmpleadosQuery, List<EmpleadoDto>>

{

public async Task<List<EmpleadoDto>> Handle(ObtenerEmpleadosQuery request, CancellationToken cancellationToken)

{

// Simulación de datos en memoria

var empleados = new List<EmpleadoDto>

{

new EmpleadoDto { Nombre = "Carlos Ruiz", Cargo = "Ingeniero de Software", Salario = 4500 },

new EmpleadoDto { Nombre = "Laura Gómez", Cargo = "Analista de Datos", Salario = 3800 },

new EmpleadoDto { Nombre = "Pedro Martínez", Cargo = "Soporte Técnico", Salario = 2800 }

};

return await Task.FromResult(empleados);

}

}

}

## 4️⃣ Agregar nuevo método GET al EmpleadoController

📄 Edita Controllers/EmpleadoController.cs y agrega esto:

using EmpleadoApp.Features.Empleados.Queries;

[HttpGet]

public async Task<IActionResult> ObtenerEmpleados()

{

var empleados = await \_mediator.Send(new ObtenerEmpleadosQuery());

return Ok(empleados);

}

## ✅ Prueba los dos endpoints

### 1. Crear empleado (POST)

* **Ruta:** POST /api/empleado
* **Body ejemplo:**

{

"nombre": "Ana López",

"cargo": "Contadora",

"salario": 3000

}

### 2. Consultar empleados (GET)

* **Ruta:** GET /api/empleado

**Respuesta simulada esperada:**

[

{

"nombre": "Carlos Ruiz",

"cargo": "Ingeniero de Software",

"salario": 4500

},

{

"nombre": "Laura Gómez",

"cargo": "Analista de Datos",

"salario": 3800

},

{

"nombre": "Pedro Martínez",

"cargo": "Soporte Técnico",

"salario": 2800

}

]

¿Continuamos con la **Versión 3**, donde integramos FluentValidation + PipelineBehavior para validar el comando CrearEmpleadoCommand antes de ejecutarlo?

# ✅ VERSIÓN 3: Validación automática con FluentValidation y PipelineBehavior

## 🎯 Objetivo

* Agregar **validación automática** para CrearEmpleadoCommand.
* Usar **FluentValidation** para definir reglas.
* Aplicar las validaciones usando un **Pipeline** de MediatR.

## 📦 1️⃣ Instalar paquete FluentValidation

En tu terminal, desde la raíz del proyecto:

dotnet add package FluentValidation

dotnet add package FluentValidation.DependencyInjectionExtensions

## 📂 2️⃣ Crear carpeta Validators

Dentro de tu carpeta Features/Empleados, crea:

📁 Features

┗ 📁 Empleados

┣ 📁 Commands

┣ 📁 Handlers

┣ 📁 Queries

┗ 📁 Validators

## ✅ 3️⃣ Crear validador para CrearEmpleadoCommand

📄 Ruta: Features/Empleados/Validators/CrearEmpleadoValidator.cs

using FluentValidation;

using EmpleadoApp.Features.Empleados.Commands;

namespace EmpleadoApp.Features.Empleados.Validators

{

public class CrearEmpleadoValidator : AbstractValidator<CrearEmpleadoCommand>

{

public CrearEmpleadoValidator()

{

RuleFor(x => x.Nombre)

.NotEmpty().WithMessage("El nombre es obligatorio.")

.MinimumLength(3).WithMessage("El nombre debe tener al menos 3 caracteres.");

RuleFor(x => x.Cargo)

.NotEmpty().WithMessage("El cargo es obligatorio.");

RuleFor(x => x.Salario)

.GreaterThan(0).WithMessage("El salario debe ser mayor a 0.");

}

}

}

## 🔁 4️⃣ Crear Pipeline de validación

📄 Ruta: Middleware/ValidationBehavior.cs

Primero, crea la carpeta Middleware en la raíz del proyecto (al nivel de Controllers).

mkdir Middleware

Luego crea el archivo:

📄 Middleware/ValidationBehavior.cs

using FluentValidation;

using MediatR;

namespace EmpleadoApp.Middleware

{

public class ValidationBehavior<TRequest, TResponse> : IPipelineBehavior<TRequest, TResponse>

where TRequest : IRequest<TResponse>

{

private readonly IEnumerable<IValidator<TRequest>> \_validators;

public ValidationBehavior(IEnumerable<IValidator<TRequest>> validators)

{

\_validators = validators;

}

public async Task<TResponse> Handle(TRequest request, RequestHandlerDelegate<TResponse> next, CancellationToken cancellationToken)

{

if (\_validators.Any())

{

var context = new ValidationContext<TRequest>(request);

var errores = \_validators

.Select(v => v.Validate(context))

.SelectMany(result => result.Errors)

.Where(f => f != null)

.ToList();

if (errores.Any())

{

var mensajes = errores.Select(e => e.ErrorMessage).ToList();

throw new ValidationException(mensajes);

}

}

return await next();

}

}

}

## ⚙️ 5️⃣ Configurar FluentValidation y el Pipeline en Program.cs

Abre Program.cs y actualiza para incluir lo siguiente:

using FluentValidation;

using EmpleadoApp.Middleware;

var builder = WebApplication.CreateBuilder(args);

// MediatR

builder.Services.AddMediatR(cfg =>

cfg.RegisterServicesFromAssembly(Assembly.GetExecutingAssembly()));

// FluentValidation

builder.Services.AddValidatorsFromAssembly(Assembly.GetExecutingAssembly());

// Pipeline de validación

builder.Services.AddTransient(typeof(IPipelineBehavior<,>), typeof(ValidationBehavior<,>));

## ❌ 6️⃣ Manejar errores de validación (opcional)

Para evitar que FluentValidation lance una excepción cruda, puedes envolver los errores en una respuesta limpia si lo deseas. Por ahora puedes dejarlo así, y verás un error como este:

{

"errors": [

"El nombre es obligatorio.",

"El salario debe ser mayor a 0."

]

}

## ✅ 7️⃣ Probar validación

### Enviar una petición ****inválida****:

{

"nombre": "",

"cargo": "",

"salario": -10

}

### Resultado esperado (HTTP 400):

{

"errors": [

"El nombre es obligatorio.",

"El cargo es obligatorio.",

"El salario debe ser mayor a 0."

]

}

### Enviar una petición ****válida****:

{

"nombre": "María Pérez",

"cargo": "Administradora",

"salario": 4000

}

👉 Se procesará y verás:

"Empleado creado con ID: <GUID>"

# 🏁 ¡Proyecto final listo!

Has creado un mini proyecto MediatR con:

* ✅ Commands y Handlers para crear empleados.
* ✅ Queries para consultar empleados.
* ✅ FluentValidation para validar reglas.
* ✅ PipelineBehavior para aplicar validaciones automáticamente.

¿Quieres que ahora empaquetemos este ejemplo como un ZIP? ¿O quieres extenderlo (por ejemplo, agregar actualización o eliminación)?