**Flujo de ejecución en la creación de un empleado**

**Flujo de la tarea de crear un Empleado:**

Cliente (POST JSON) ───▶ [HttpPost] Controller

└──▶ \_mediator.Send(command)

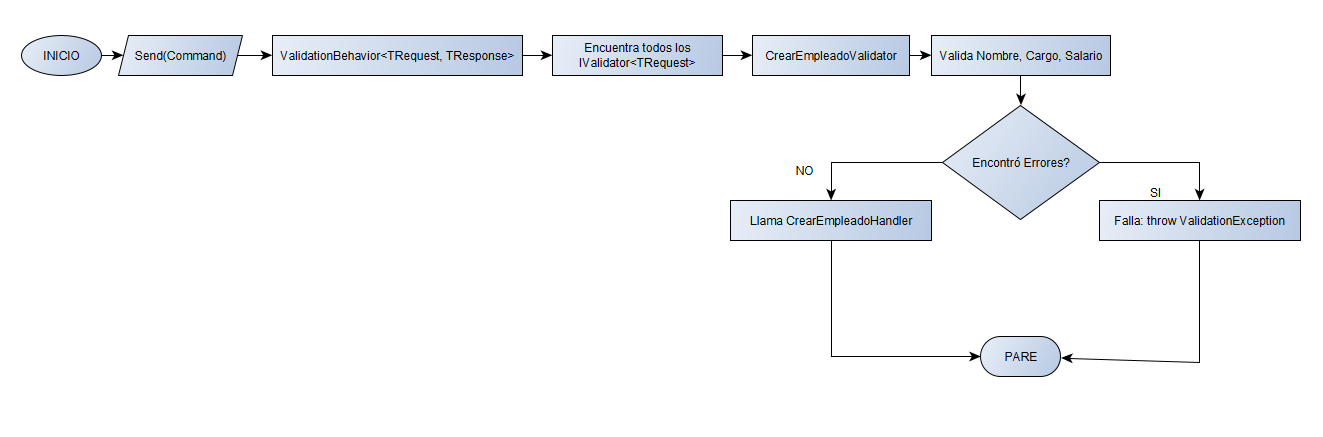
└──▶ ValidationBehavior (FluentValidation)

└── Si válido ─▶ CrearEmpleadoHandler

└── return "Empleado creado"

◀───────────────────────

◀── resultado

 ◀── return Ok("Empleado creado")

[HttpPost]

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

{

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

return Ok(resultado);

}

[HttpPost]

El atributo HttpPost como bien es sabido indica que este método debe responder a una petición HTTP de tipo POST. Recuerda que estamos creando un empleado.

Ejemplo:

POST http://localhost:5000/api/empleado

### async:

Permite que el método sea **asíncrono**, es decir, puede esperar operaciones que toman tiempo (como I/O o llamadas a servicios) sin bloquear el hilo principal.

### Task<IActionResult>:

Esto significa que el método devuelve una **tarea (Task),** que al completarse retorna un objeto del tipo **IActionResult**. Este tipo es parte de ASP.NET Core y representa cualquier tipo de **respuesta HTTP válida** (como Ok(), BadRequest(), NotFound(), etc.).

([FromBody] CrearEmpleadoCommand command)

### CrearEmpleadoCommand command:

Este es el objeto que representa los **datos del empleado que el cliente está enviando**. Por ejemplo, un JSON así:

{

"nombre": "Juan Pérez",

"cargo": "Ingeniero",

"salario": 5000000

}

ASP.NET Core automáticamente convierte ese JSON a un objeto CrearEmpleadoCommand.

### FromBody:

Este atributo le dice a ASP.NET: **“toma ese JSON desde el cuerpo del mensaje HTTP”**, y no de la URL ni de los parámetros de consulta.

Sin FromBody, por defecto intentaría buscar en la URL (lo que no aplica para POST).

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

Aquí es donde entra en acción **MediatR**.

### \_mediator.Send(command):

Esto **envía el comando** al "bus" de MediatR, el cual busca automáticamente **quién maneja ese tipo de comando** (CrearEmpleadoCommand).

MediatR entonces:

1. **Encuentra el handler** correspondiente, es decir:

class CrearEmpleadoHandler : IRequestHandler<**CrearEmpleadoCommand**, string>

1. **Antes de ir al handler**, MediatR pasa el comando por todos los middleware (IPipelineBehavior), como en este caso la clase (ValidationBehavior con FluentValidation).

ValidationBehavior es un **middleware personalizado de MediatR** que intercepta cualquier request (comando o consulta) **antes** de que llegue al handler.

Esta clase:

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

Este comportamiento se registra en Program.cs como:

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

Aquí está la magia. Esto le dice a MediatR:  
“Antes de enviar cualquier request al handler, pásala por esta clase ValidationBehavior.”

MediatR detecta que tenemos un IPipelineBehavior, así que ejecuta el metodo:

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

En ese momento:

* TRequest es CrearEmpleadoCommand
* TResponse es string

### ValidationBehavior busca validadores registrados

En esta parte del código:

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())

{

throw new ValidationException(errores);

}

}

Aquí es cuando se invoca **CrearEmpleadoValidator**.

¿Por qué?

Porque registramos los validadores así:

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

Esto busca automáticamente todas las clases que implementan:

AbstractValidator<T>

Y como tenemos:

public class CrearEmpleadoValidator : AbstractValidator<CrearEmpleadoCommand>

MediatR **ya sabe** que este validador aplica cuando TRequest es CrearEmpleadoCommand.

### Se ejecuta entonces CrearEmpleadoValidator

El método Validate(context) activa esto:

public CrearEmpleadoValidator()

{

RuleFor(e => e.Nombre)

.NotEmpty().WithMessage("No deje el nombre vacío")

.MinimumLength(5);

RuleFor(c => c.Cargo)

.NotEmpty().WithMessage("Proporcione un cargo");

RuleFor(s => s.Salario)

.GreaterThan(0)

.WithMessage("Salario debe ser mayor que cero (0)");

}

Si alguna regla NO se cumple, se genera una lista de errores.

throw new ValidationException(errores);

Y nunca se llega al Handler. La respuesta será un error HTTP 500 (por ahora), pero podríamos capturarla de mejor forma, si implementamos un middleware global de excepciones.

Si NO hay errores:

return await next();

Si la validación pasa, ejecuta **el** Handle() **del handler**, que contiene la lógica de negocio.

public class CrearEmpleadoHandler : IRequestHandler<CrearEmpleadoCommand, string>

{

public async Task<string> Handle(CrearEmpleadoCommand request,

CancellationToken cancellationToken)

{

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

request.Id = id;

Console.WriteLine($"Empleado {request.Nombre} creado");

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

}

}

1. Lo que el handler retorne (en este caso, un string) se devuelve a esta línea y se guarda en resultado. Del controlador.

## return Ok(resultado);

Esto devuelve una respuesta HTTP 200 OK al cliente con el contenido que produjo el handler.

Por ejemplo, si el handler retornó:

"Empleado con ID 5ec44... creado"

Entonces el cliente recibe algo como:

HTTP/1.1 200 OK

Content-Type: application/json

"Empleado con ID 5ec44... creado"