🔹 **CQRS در .NET – Command and Query Responsibility Segregation**  
🚀 CQRS یک الگوی معماری است که مسئولیت‌های **خواندن (Query)** و **نوشتن (Command)** را در سیستم از هم **جدا** می‌کند. این تفکیک باعث **بهبود عملکرد، مقیاس‌پذیری و نگهداری راحت‌تر** اپلیکیشن‌های پیچیده می‌شود.

**✅ CQRS چیست و چرا استفاده می‌شود؟**

🔹 **در معماری معمولی (CRUD):**

* همه‌ی عملیات خواندن (SELECT) و نوشتن (INSERT, UPDATE, DELETE) از **یک مدل و یک دیتابیس استفاده می‌کنند.**
* اگر سیستم پیچیده شود، دسترسی‌های زیاد ممکن است باعث **کندی خواندن و نوشتن شود.**

🔹 **در CQRS:**

* مدل **خواندن (Query Model)** و **نوشتن (Command Model)** کاملاً **جدا** هستند.
* پردازش درخواست‌های خواندن و نوشتن بهینه می‌شود.
* سیستم‌های با حجم بالای درخواست **مقیاس‌پذیرتر** خواهند بود.

**✅ کاربردهای CQRS**

🔹 CQRS معمولاً در سیستم‌هایی که:  
✅ **میزان خواندن و نوشتن نابرابر** دارند (مثلاً خواندن زیاد، ولی نوشتن کم).  
✅ **نیاز به پاسخ‌گویی سریع به درخواست‌های خواندن دارند** بدون اینکه عملیات نوشتن کند شود.  
✅ **سیستم‌های ماژولار و توزیع‌شده** دارند (مانند **Microservices**).  
✅ **تحلیل داده‌های پیچیده** و پردازش‌های بلادرنگ دارند.

**✅ مثال از CQRS در .NET**

📌 **سناریو: مدیریت کاربران (خواندن لیست کاربران و ثبت کاربر جدید)**  
🔹 **بدون CQRS (روش کلاسیک CRUD):**

public class UserController : Controller

{

private readonly UserRepository \_userRepository;

public UserController(UserRepository userRepository)

{

\_userRepository = userRepository;

}

// خواندن اطلاعات کاربران

public IActionResult GetUsers()

{

var users = \_userRepository.GetAllUsers();

return View(users);

}

// ثبت کاربر جدید

[HttpPost]

public IActionResult CreateUser(User user)

{

\_userRepository.AddUser(user);

return RedirectToAction("GetUsers");

}

}

✅ **مشکلات این روش:**

* مدل **خواندن و نوشتن یکی است** → کارایی ممکن است پایین بیاید.
* اگر خواندن اطلاعات کاربران زیاد باشد، نوشتن ممکن است **کند شود.**

🔹 **با CQRS (تفکیک Command و Query):**  
📌 **ابتدا دو مدل جداگانه برای خواندن و نوشتن تعریف کنیم:**

public class GetUsersQuery

{

public IEnumerable<UserDto> Execute()

{

using var db = new ApplicationDbContext();

return db.Users.Select(u => new UserDto { Id = u.Id, Name = u.Name }).ToList();

}

}

public class CreateUserCommand

{

public void Execute(User user)

{

using var db = new ApplicationDbContext();

db.Users.Add(user);

db.SaveChanges();

}

}

📌 **حالا کنترلر به این شکل می‌شود:**

public class UserController : Controller

{

private readonly GetUsersQuery \_getUsersQuery;

private readonly CreateUserCommand \_createUserCommand;

public UserController(GetUsersQuery getUsersQuery, CreateUserCommand createUserCommand)

{

\_getUsersQuery = getUsersQuery;

\_createUserCommand = createUserCommand;

}

public IActionResult GetUsers()

{

var users = \_getUsersQuery.Execute();

return View(users);

}

[HttpPost]

public IActionResult CreateUser(User user)

{

\_createUserCommand.Execute(user);

return RedirectToAction("GetUsers");

}

}

✅ **مزایای این روش:**  
🔹 **خواندن و نوشتن جدا شده‌اند** → بهینه‌تر اجرا می‌شوند!  
🔹 **پردازش خواندن می‌تواند از دیتابیس جداگانه یا کش (Cache) استفاده کند.**  
🔹 **بهبود مقیاس‌پذیری در پردازش‌های حجیم.**

**✅ CQRS را استفاده کنیم یا نه؟**

🚀 **چه زمانی مناسب است؟**  
✅ **زمانی که سیستم بزرگ باشد و نیاز به بهینه‌سازی داشته باشد.**  
✅ **زمانی که درخواست‌های خواندن بسیار زیاد باشد.**  
✅ **زمانی که سرعت پاسخ‌دهی به خواندن اطلاعات مهم باشد.**

🚨 **چه زمانی لازم نیست؟**  
❌ **برای اپلیکیشن‌های کوچک و ساده، CQRS پیچیدگی غیرضروری ایجاد می‌کند.**  
❌ **اگر میزان خواندن و نوشتن تقریباً برابر باشد، CQRS ممکن است کمکی نکند.**

**🔹 جمع‌بندی**

✅ **CQRS کمک می‌کند خواندن و نوشتن اطلاعات کاملاً جدا شوند، و سیستم کارآمدتر عمل کند.**  
✅ **اگر سیستم بزرگی داشته باشی که نیاز به خواندن سریع دارد، CQRS می‌تواند عالی باشد!**  
✅ **اگر اپلیکیشن کوچک باشد، CQRS ممکن است فقط پیچیدگی اضافه کند.**

🚀 **دوست داری روی پیاده‌سازی CQRS در پروژه‌ات کار کنیم؟** 😃  
یا سوال خاصی درباره‌ی این روش داری؟ 👌