مثالی از CQRS و Event Sourcing :

این پروژه یک وب سرویس است برای انجام کارهایی مربوط به حساب مانند: ایجاد حساب ،برداشت و واریز و گرفتن مانده نوشت شده است ضنما این پروژه با تکنولووژی asp.net .net core 8 با دیتابیس SQLLite توسعه یافت است .

بخش اول – توضیحات جامع درباره CQRS (Command Query Responsibility Segregation) و Event Sourcing:

CQRS (تفکیک مسئولیت‌های دستور و پرسش)

CQRSیک الگوی معماری است که در آن وظایف مربوط به خواندن داده‌ها و نوشتن داده‌ها از یکدیگر جدا می‌شوند. این الگو به طور ویژه در سیستم‌هایی که مقیاس‌پذیری و عملکرد بالا نیاز دارند، سودمند است.

اصول:CQRS

1. **تفکیک وظایف:**
   - **Commands:وظیفه آن‌ها تغییر وضعیت سیستم است. این‌ها اغلب شامل عملیات‌هایی مانند ایجاد، به‌روزرسانی یا حذف داده‌ها هستند.**
   - **Queries:وظیفه آن‌ها بازیابی داده‌ها از سیستم است. این‌ها عملیات‌هایی هستند که به داده‌ها دسترسی دارند بدون تغییر وضعیت آن‌ها.**
2. **مدل‌های مختلف داده:**
   - **درCQRS ، معمولاً از مدل‌های متفاوتی برای خواندن و نوشتن داده‌ها استفاده می‌شود. این می‌تواند به بهینه‌سازی عملکرد و مقیاس‌پذیری کمک کند.**
   - **برای مثال، مدل نوشتن ممکن است کاملاً نرمال‌سازی شده باشد، در حالی که مدل خواندن می‌تواند انباشته شده و بهینه باشد.**
3. **بهبود مقیاس‌پذیری:**
   - **با جدا کردن وظایف خواندن و نوشتن، می‌توان هر کدام را به صورت جداگانه مقیاس‌پذیر کرد، به عنوان مثال، با استفاده از بارگذاری متوازن یا کش.**

مزایای:CQRS

- **مقیاس‌پذیری بالا :امکان مقیاس‌گذاری مستقل بخش‌های فرمان و پرسش.**
- **تنظیم بهینه عملکرد :بهینه‌سازی جداگانه برای خواندن و نوشتن.**
- **ساده‌سازی کد :درک بهتر ساختار سیستم و تسهیل نگهداری کد.**
- **قابلیت تغییر :تحمل پذیرتر در برابر تغییرات کسب‌وکار و به روز رسانی‌های آینده.**

معایب:CQRS

- پیچیدگی بیشتر :جداسازی بخش‌های خواندن و نوشتن می‌تواند پیچیدگی معماری را افزایش دهد.
- نیاز به هماهنگی :نیاز به هماهنگی بین دو بخش در صورت تغییر وضعیت سیستم.

Event Sourcing(ذخیره‌سازی رویداد)

Event Sourcingیک الگوی معماری است که در آن وضعیت سیستم به واسطه رویدادهایی که باعث تغییر آن وضعیت شده‌اند، ذخیره می‌شود. به عبارت دیگر، به جای ذخیره وضعیت کنونی سیستم، تمام تغییرات (رویدادها) ثبت می‌شوند.

اصول:Event Sourcing

1. ذخیره‌سازی رویدادها:
   - تمامی تغییرات به عنوان مجموعه‌ای از رویدادها ذخیره می‌شوند. هر رویداد نمایانگر یک تغییر در وضعیت سیستم است.
   - این رویدادها به ترتیب زمان ثبت می‌شوند و می‌توانند به راحتی بازسازی وضعیت کنونی سیستم را فراهم کنند.
2. بازسازی وضعیت:
   - برای به‌دست آوردن وضعیت جاری سیستم، باید تمام رویدادها را از ابتدا تا کنون بروزرسانی کرد. این کار با replay کردن رویدادها امکان‌پذیر است.
3. تضمین یکپارچگی:
   - از آنجا که هر تغییر در سیستم به صورت یک رویداد ثبت می‌شود، این باعث می‌شود که تاریخچه کاملی از تغییرات در دسترس باشد، که این خود موجب افزایش شفافیت و یکپارچگی می‌شود.

مزایای:Event Sourcing

- تاریخچه کامل تغییرات :امکان بررسی تمامی تغییرات و وضعیت‌های پیشین.
- آسانی در بازیابی و تنقیح داده‌ها :بازسازی وضعیت‌ها در صورت نیاز.
- امکان پردازش‌های پیچیده :جداسازی لایه‌ها و استفاده از پردازش‌های متفاوت بر روی رویدادها.

معایب:Event Sourcing

- پیچیدگی بیشتر :نیاز به مدیریت و پردازش تعداد زیادی رویداد.
- اندازه بزرگ داده‌ها :ذخیره تمام رویدادها می‌تواند منجر به حجم بالای داده شود.
- نقص در تفسیر رویدادها :نیاز به استاندارد سازی و تفسیر صحیح رویدادها برای اطمینان از درک درست آن‌ها.

رابطه بین CQRS وEvent Sourcing

CQRSو Event Sourcing معمولاً با هم ترکیب می‌شوند. در این ترکیب، می‌توان رویدادها را به عنوان روش ذخیره‌سازی داده‌ها در بخش نوشتن CQRS استفاده کرد. وقتی که دستوراتی به سیستم وارد می‌شوند، رویدادهای مربوط به آن دستور ذخیره می‌شوند و از طریق آن‌ها می‌توان وضعیت کنونی را بازسازی کرد.

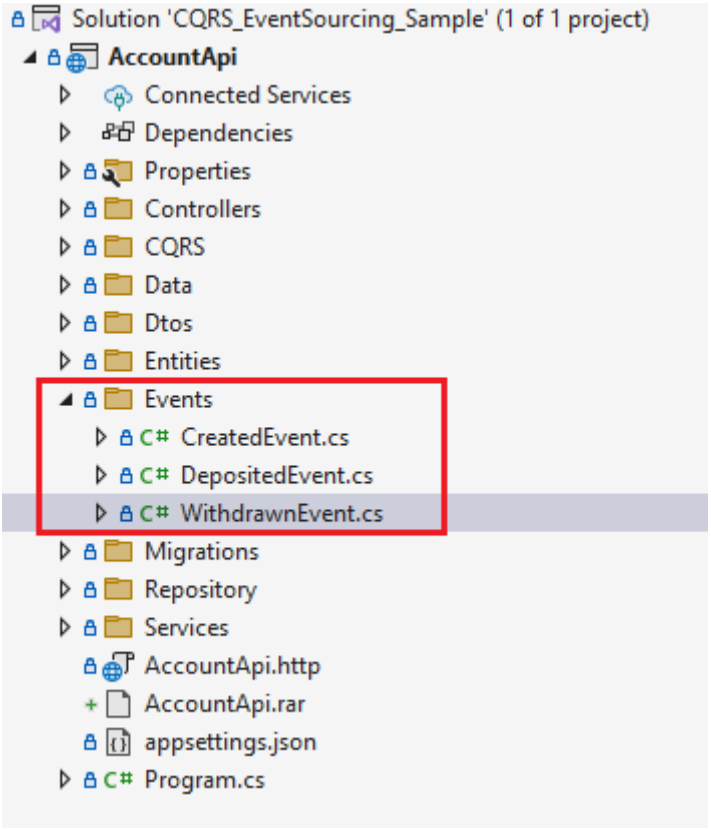
بخش دوم –استفاده از CQRSو Event Sourcing ، ما به یک برنامه مدیریت حساب‌های بانکی خواهیم پرداخت. در این مثال، ساختار برنامه شامل بخش‌های زیر است:

1. مدل رویداد :(Events) شامل رویدادها مانند واریز، برداشت، و ایجاد حساب.
2. ایجاد Commands و Queries :
    1. دستورات :(Commands) شامل عملیاتی برای تغییر وضعیت سیستم، مانند واریز و برداشت.
    2. پرسش‌ها :(Queries) شامل درخواست‌های برای دریافت داده‌ها، مانند دریافت موجودی حساب.
3. حساب :(Account) پیاده‌سازی حساب که وضعیت را بر اساس رویدادها پیگیری می‌کند.
4. مدل event
5. ذخیره (Repository) حساب ها
6. پیاده سازی پردازش Commands و Queries – برای ذخیره حساب و رویدادها
7. مدیریت رویداد
8. api

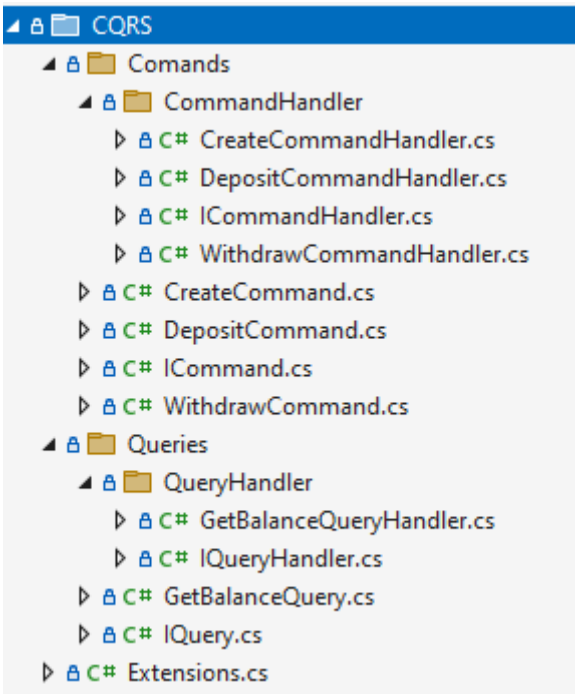
1. تعریف مدل‌های رویداد

مدل رویداد برای ذخیره‌سازی تغییرات وضعیت:

```
public class CreatedEvent
{
    public string Name { get; set; }
}

public class DepositedEvent
{
    public decimal Amount { get; set; }
}

public class WithdrawnEvent
{
    public decimal Amount { get; set; }
}
```



2. ایجاد دستورات و پرسش‌ها



به عنوان مثال برای ایجاد حساب

```
public class CreateCommand : ICommand<Guid>
{

    public string Name { get; set; }

    public decimal InitialBalance { get; set; }
}
```

```
1 reference | Ghamari.Vahid, 6 days ago | 1 author, 1 change
public class CreateCommandHandler : ICommandHandler<CreateCommand, Guid>
{
    private readonly IAccountService accountService;

    0 references | Ghamari.Vahid, 6 days ago | 1 author, 1 change
    public CreateCommandHandler(IAccountService accountService)
    {
        this.accountService = accountService;
    }

    4 references | Ghamari.Vahid, 6 days ago | 1 author, 1 change
    public async Task<Guid> HandlerAsync(CreateCommand request)
    {
        var result = await accountService.CreateAsync(request);
        return result;
    }
}
```

3. پیاده‌سازی حساب

کلاس Account که شامل منطق CQRS و Event Sourcing است:

```
4 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
public Guid Id { get; private set; }
7 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
public decimal Balance { get; private set; }
2 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
public string Name { get; private set; }


1 reference | Ghamari.Vahid, 7 days ago | 1 author, 3 changes
private Account(string name, decimal initialBalance)
{
    Name = name;
    Id = Guid.NewGuid();
    Balance = initialBalance;
}
```

```
1 reference | Ghamari.Vahid, 7 days ago | 1 author, 2 changes
public static async Task<Account> CreateAccountAsync(string name, decimal initialBalance,
    IGenericRepository<Event> eventStore)
{
    var account = new Account(name, initialBalance);

    var createdEvent = new CreatedEvent { Name = name };
    account.ApplyChange(createdEvent);

    var eventEntity = account.CreateEventEntity(nameof(CreatedEvent), createdEvent);
    await eventStore.AddAsync(eventEntity);

    return account;
}

1 reference | Ghamari.Vahid, 7 days ago | 1 author, 1 change
public async Task DepositAsync(decimal amount, IGenericRepository<Event> eventStore)
{
    if (amount <= 0)
        throw new ArgumentException("Deposit amount must be greater than zero.");

    Balance += amount;
    ApplyChange(new DepositedEvent { Amount = amount });

    var eventEntity = CreateEventEntity(nameof(DepositedEvent), new { Amount = amount });
    await eventStore.AddAsync(eventEntity);
}

1 reference | Ghamari.Vahid, 7 days ago | 1 author, 1 change
public async Task WithdrawAsync(decimal amount, IGenericRepository<Event> eventStore)
{
    if (amount <= 0)
        throw new ArgumentException("Withdrawal amount must be greater than zero.");
    if (amount > Balance)
        throw new InvalidOperationException("Insufficient funds.");

    Balance -= amount;
    ApplyChange(new WithdrawnEvent { Amount = amount });

    var eventEntity = CreateEventEntity(nameof(WithdrawnEvent), new { Amount = amount });
    await eventStore.AddAsync(eventEntity);
}
```

```
0 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
public void ClearUncommittedChanges() => _changes.Clear();
0 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
public IEnumerable<object> GetChanges() => _changes;

3 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
private void ApplyChange(object @event)
{
    _changes.Add(@event);
}

3 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
private Event CreateEventEntity(string eventType, object eventData)
{
    return new Event
    {
        Id = Guid.NewGuid(),
        OccurredOn = DateTime.UtcNow,
        AggregateId = this.Id,
        EventType = eventType,
        EventData = JsonSerializer.Serialize(eventData)
    };
}

1 reference | Ghamari.Vahid, 7 days ago | 1 author, 1 change
public void LoadFromEvents(IEnumerable<object> events)
{
    foreach (var @event in events)
    {
        switch (@event)
        {
            case CreatedEvent createdEvent:
                Name = createdEvent.Name;
                break;
            case DepositedEvent depositedEvent:
                Balance += depositedEvent.Amount;
                break;
            case WithdrawnEvent withdrawnEvent:
                Balance -= withdrawnEvent.Amount;
                break;
        }
    }
}
```

4- رویدار:در اینجا مقدار هر event و تایپ اون رو ذخیره میکنم

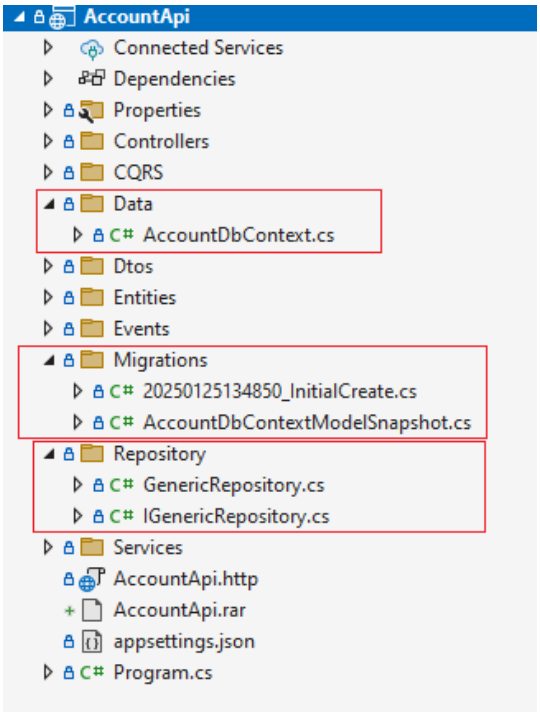
```
public class EventEntity
{
    public Guid Id { get; set; }
    public DateTime OccurredOn { get; set; }
    public Guid AggregateId { get; set; }
    public string EventType { get; set; }
    public string EventData { get; set; }
}
```

5-ریپازتوری

این ریپازتوری به شکل generic پیاده سازی شده تا هم event وهم ذخیره حساب انجام شود.

```
public interface IGenericRepository<T> where T : class
{
    1 reference | Ghamari.Vahid, 7 days ago | 1 author, 1 change
    Task<IEnumerable<T>> GetAllAsync();
    3 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
    Task<T> GetByIdAsync(Guid id);
    5 references | Ghamari.Vahid, 7 days ago | 1 author, 2 changes
    Task<T> AddAsync(T entity);
    3 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
    Task UpdateAsync(T entity);
    1 reference | Ghamari.Vahid, 7 days ago | 1 author, 1 change
    Task DeleteAsync(Guid id);
    2 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
    Task<IEnumerable<T>> FindAsync(Expression<Func<T, bool>> predicate);
}
```

```
3 references | Ghamari.Vahid, 7 days ago | 1 author, 3 changes
public class GenericRepository<T> : IGenericRepository<T> where T : class
{
    protected readonly AccountDbContext _context;
    private readonly DbSet<T> _dbSet;

    0 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
    public GenericRepository(AccountDbContext context)
    {
        _context = context;
        _dbSet = context.Set<T>();
    }

    1 reference | Ghamari.Vahid, 7 days ago | 1 author, 1 change
    public async Task<IEnumerable<T>> GetAllAsync() => await _dbSet.ToListAsync();

    3 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
    public async Task<T> GetByIdAsync(Guid id) => await _dbSet.FindAsync(id);

    5 references | Ghamari.Vahid, 7 days ago | 1 author, 2 changes
    public async Task<T> AddAsync(T entity)
    {
        await _dbSet.AddAsync(entity);
        await _context.SaveChangesAsync();
        return entity;
    }

    3 references | Ghamari.Vahid, 7 days ago | 1 author, 2 changes
    public async Task UpdateAsync(T entity)
    {
        _dbSet.Update(entity);
        await _context.SaveChangesAsync();
    }

    1 reference | Ghamari.Vahid, 7 days ago | 1 author, 2 changes
    public async Task DeleteAsync(Guid id)
    {
        var entity = await GetByIdAsync(id);
        if (entity != null)
        {
            _dbSet.Remove(entity);
            await _context.SaveChangesAsync();
        }

    }

    2 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
    public async Task<IEnumerable<T>> FindAsync(Expression<Func<T, bool>> predicate)
        => await _dbSet.Where(predicate).ToListAsync();
}
```



6- پیاده‌سازی پردازش دستورات و پرسش‌ها در یک سرویس

```
public class AccountService : IAccountService
{
    private readonly IGenericRepository<Account> accountrepository;
    private readonly IGenericRepository<Event> eventrepository;

    0 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
    public AccountService(IGenericRepository<Account> Accountrepository,
        IGenericRepository<Event> Eventrepository)
    {
        accountrepository = Accountrepository;
        eventrepository = Eventrepository;
    }

    2 references | Ghamari.Vahid, 7 days ago | 1 author, 2 changes
    public async Task DepositAsync(DepositCommand command)
    {
        var account = await LoadAccountAsync(command.AccountId);
        await account.DepositAsync(command.Amount, eventrepository);
        await accountrepository.UpdateAsync(account);

    }

    2 references | Ghamari.Vahid, 7 days ago | 1 author, 2 changes
    public async Task WithdrawAsync(WithdrawCommand command)
    {
        var account = await LoadAccountAsync(command.AccountId);
        await account.WithdrawAsync(command.Amount, eventrepository);
      await  accountrepository.UpdateAsync(account);
    }

    2 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
    public async Task<decimal> GetBalanceAsync(GetBalanceQuery query)
    {
        var account = await LoadAccountAsync(query.AccountId);
        return account.Balance;
    }

    3 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
    private async Task<Account> LoadAccountAsync(Guid accountId)
    {
        var events = await eventrepository.FindAsync(c => c.AggregateId == accountId);
        var account = await accountrepository.GetByIdAsync(accountId);
        account.LoadFromEvents(events);
        return account;
    }

    2 references | Ghamari.Vahid, 6 days ago | 1 author, 2 changes
    public async Task<Guid> CreateAsync(CreateCommand createCommand)
    {
        var account = await (new AccountFactory(eventrepository) { }).
            CreateAccountAsync(createCommand.Name, createCommand.InitialBalance);
        var result = await accountrepository.AddAsync(account);
        return result.Id;
    }
}
```

7-مدیریت رویدادها

متد **LoadFromEvents** به‌منظور بارگذاری و بازیابی وضعیت داخلی کلاس **Account** از روی رویدادهایی که در طول زمان ذخیره شده‌اند، طراحی شده است. معمولاً این متد باید در درون کلاس مدل اصلی که نمایانگر موجودیت شماست، به نام **Account** در این مورد، تعریف شود.

شما باید متد **LoadFromEvents** را به کلاس **Account** اضافه کنید. این متد به هر رویدادی که به حساب مربوط می‌شود پاسخ می‌دهد و وضعیت **Account** را بر اساس آن رویدادها به‌روزرسانی می‌کند.

```
1 reference | Ghamari.Vahid, 7 days ago | 1 author, 1 change
public void LoadFromEvents(IEnumerable<object> events)
{
    foreach (var @event in events)
    {
        switch (@event)
        {
            case CreatedEvent createdEvent:
                Name = createdEvent.Name;
                break;
            case DepositedEvent depositedEvent:
                Balance += depositedEvent.Amount;
                break;
            case WithdrawnEvent withdrawnEvent:
                Balance -= withdrawnEvent.Amount;
                break;
        }
    }
}
```

استفاده از متد**LoadFromEvents**

وقتی شما یک حساب را بارگذاری می‌کنید، دیگر نیاز نیست که به تمام رویدادها روی بیاورید. می‌توانید از متد **LoadFromEvents** برای به‌روزرسانی وضعیت حساب به‌کار ببرید. به این شکل:

```
public async Task<Guid> CreateAsync(CreateCommand createCommand)
{
    var account = await (new AccountFactory(eventrepository) { }).
        CreateAccountAsync(createCommand.Name, createCommand.InitialBalance);
    var result = await accountrepository.AddAsync(account);
    return result.Id;
}
```

این ترتیب به شما این امکان را می‌دهد که وضعیت حساب را در هنگام بارگذاری از روی رویدادها به‌راحتی مدیریت کنید. هر بار که یک حساب بارگذاری می‌شود، وضعیت آن بر اساس رویدادهای ذخیره‌شده به‌روز می‌شود.

نکته : ما در این پروژه از **AccountFactory** استفاده کرده ایم زیرا لازم بود در زمان ایجاد از سرویس های استفاده کنم که **async** بوده :

```
2 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
public class AccountFactory
{
    private readonly IGenericRepository<Event> _eventStore;

    1 reference | Ghamari.Vahid, 7 days ago | 1 author, 1 change
    public AccountFactory(IGenericRepository<Event> eventStore)
    {
        _eventStore = eventStore;
    }

    1 reference | Ghamari.Vahid, 7 days ago | 1 author, 1 change
    public async Task<Account> CreateAccountAsync(string name, decimal initialBalance)
    {
        var account = await Account.CreateAccountAsync(name, initialBalance, _eventStore);
        return account;
    }
}
```

دیگر ما contractor به شکل public نداریم

```
public static async Task<Account> CreateAccountAsync(string name, decimal initialBalance,
    IGenericRepository<Event> eventStore)
{
    var account = new Account(name, initialBalance);

    var createdEvent = new CreatedEvent { Name = name };
    account.ApplyChange(createdEvent);

    var eventEntity = account.CreateEventEntity(nameof(CreatedEvent), createdEvent);
    await eventStore.AddAsync(eventEntity);

    return account;
}
```

- **اگر فقط در حال ایجاد حساب هستید و رویدادهای جدید را ذخیره می‌کنید، نیازی به LoadFromEvents نیست.**
- **اگر بخواهید یک حساب موجود را از روی رویدادهایش بازسازی کنید، می‌توانید از LoadFromEvents استفاده کنید و این متد مرتبط با بارگذاری وضعیت حساب از روی رویدادها خواهد بود.**

api -8

```
[ApiController]
[Route("[controller]")]

public class AccountController : ControllerBase
{


    [HttpPost]

    public async Task<IActionResult> CreateAccunt([FromBody] RequestAccuont request, [FromServices] ICommandHandler<CreateCommand, Guid> command)
    {
        var dd = await command.HandlerAsync(new CreateCommand() { InitialBalance = request.Amount, Name = request.Name });
        return Ok(dd);
    }


    [HttpPut("DepositAsync")]

    public async Task<IActionResult> DepositAsync([FromBody] RequestAccuont request, [FromServices] ICommandHandler<DepositCommand, bool> command)
    {
        var result = await command.HandlerAsync(new DepositCommand() { AccountId = request.Id, Amount = request.Amount });
        return Ok(result);
    }

    [HttpPut("WithdrawAsync")]

    public async Task<IActionResult> WithdrawAsync([FromBody] RequestAccuont request, [FromServices] ICommandHandler<WithdrawCommand, bool> command)
    {
        var result = await command.HandlerAsync(new WithdrawCommand() { AccountId = request.Id, Amount = request.Amount });
        return Ok(result);
    }

    [HttpGet("GetBalanceAsync")]

    public async Task<IActionResult> GetBalanceAsync([FromQuery] Guid accountId, [FromServices] IQueryHandler<GetBalanceQuery, decimal> command)
    {
        var dd = await command.HandlerAsync(new GetBalanceQuery() { AccountId = accountId });
        return Ok(dd);
    }
```

با این طراحی:

- **CQRS:** شما دستورات را از پرسش‌ها جدا کرده‌اید.
- **Event Sourcing:** با استفاده از رویدادها وضعیت سیستم را بازیابی می‌کنید.
- می‌توانید به‌راحتی تغییرات را پیگیری کرده و منطق پیچیده‌تری را ایجاد کنید.

نکته تکمیلی Program

```
2 references | Ghamari.Vahid, 6 days ago | 1 author, 4 changes
public class Program
{
    0 references | Ghamari.Vahid, 6 days ago | 1 author, 4 changes
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);


        builder.Services.AddControllers();
        builder.Services.AddEndpointsApiExplorer();
        builder.Services.AddSwaggerGen();
        builder.Services.AddDbContext<AccountDbContext>();
        builder.Services.AddScoped<IGenericRepository<Account>, GenericRepository<Account>>();
        builder.Services.AddScoped<IGenericRepository<Event>, GenericRepository<Event>>();
        builder.Services.AddScoped<IAccountService, AccountService>();
        builder.Services.AddCommandHandlers(typeof(Program));
        builder.Services.AddQueryHandlers(typeof(Program));


        var app = builder.Build();
        app.UseSwagger();
        app.UseSwaggerUI();
        app.UseAuthorization();
        app.MapControllers();
        app.Run();
    }
}
```

بخش سوم :

در پیاده‌سازی‌ای که انجام شد، فقط رویدادها ذخیره می‌شوند و وضعیت کامل حساب (Account) به‌طور مستقیم ذخیره نمی‌شود. به همین دلیل، هر بار که می‌خواهید در مورد حساب‌ها اطلاعات جدیدی کسب کنید، باید همه رویدادهای مربوط به آن حساب را بارگذاری کنید و از آن‌ها برای بازسازی وضعیت استفاده کنید.

برای کامل کردن این فرآیند، می‌توانیم به دو روش پیش برویم:

1- ذخیره وضعیت حساب(Snapshot)

در این روش، علاوه بر ذخیره‌سازی رویدادها، یک **snapshot** از وضعیت فعلی حساب نیز ذخیره می‌کنیم تا بازیابی سریع‌تری انجام دهیم. این کار می‌تواند عملکرد را بهبود بخشد، مخصوصاً اگر تعداد زیادی رویداد مربوط به یک حساب وجود داشته باشد.

1- تعریف مدل Snapshot

```
public class AccountSnapshot
{
    public Guid Id { get; set; }
    public decimal Balance { get; set; }
    public string Name { get; set; }
    public DateTime CreatedAt { get; set; }
}
```

2- ذخیره Snapshot پس از تشکیل رویداد

در متد **DepositAsync** و**WithdrawAsync** ، پس از ذخیره‌سازی رویداد، می‌توانید **Snapshot** را نیز ذخیره کنید. اگر از تنظیمات سخت‌افزاری یا نرم‌افزاری خاصی استفاده می‌کنید، پیشنهاد می‌شود در یک دوره معین (مثلاً هر ۵۰ رویداد)، **Snapshot**را ذخیره کنید.

```
private async Task SaveSnapshotAsync(IEventStore eventStore)
{
    var snapshot = new AccountSnapshot
    {
        Id = this.Id,
        Balance = this.Balance,
        Name = this.Name,
        CreatedAt = DateTime.UtcNow
    };

    // را ذخیره کند snapshot اضافه کنید که IEventStore باید یک متد جدید به
    await eventStore.SaveSnapshotAsync(snapshot);
}
```

3-کار با Snapshot

وقتی که وضعیت حساب از روی رویدادها بارگذاری می‌شود، ابتدا می‌توانید از **Snapshot** استفاده کنید:

```
private async Task<Account> LoadAccountAsync(Guid accountId)
{
    // بارگذاری Snapshot
    var snapshot = await _eventStore.GetSnapshotAsync(accountId);

    var account = new Account
    {
        Id = snapshot.Id,
        Balance = snapshot.Balance,
        Name = snapshot.Name
    };

    // بارگذاری رویدادها برای بروزرسانی وضعیت از Snapshot
    var events = await _eventStore.GetEventsAsync(accountId);
    account.LoadFromEvents(events);

    return account;
}
```

2- ایجاد یک منبع داده برای حساب‌ها

می‌توانیم یک منبع داده (مانند پایگاه داده) برای ذخیره‌سازی وضعیت فعلی حساب‌ها داشته باشیم. در این حالت، آنچه ذخیره می‌کنیم، رویدادها هستند و در صورت نیاز، وضعیت حساب‌ها از روی رویدادها بازیابی می‌شود.

در پروژه که انجام دادیم ما از این روش استفاده کردم مثال :

```
2 references | Ghamari.Vahid, 6 days ago | 1 author, 2 changes
public async Task<Guid> CreateAsync(CreateCommand createCommand)
{
    var account = await (new AccountFactory(eventrepository) { }).
        CreateAccountAsync(createCommand.Name, createCommand.InitialBalance);
    var result = await accountrepository.AddAsync(account);
    return result.Id;
}
```

```
2 references | Ghamari.Vahid, 7 days ago | 1 author, 2 changes
public async Task DepositAsync(DepositCommand command)
{
    var account = await LoadAccountAsync(command.AccountId);
    await account.DepositAsync(command.Amount, eventrepository);
    await accountrepository.UpdateAsync(account);

}
```

```
2 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
public async Task<decimal> GetBalanceAsync(GetBalanceQuery query)
{
    var account = await LoadAccountAsync(query.AccountId);
    return account.Balance;
}
```

```
3 references | Ghamari.Vahid, 7 days ago | 1 author, 1 change
private async Task<Account> LoadAccountAsync(Guid accountId)
{
    var events = await eventrepository.FindAsync(c => c.AggregateId == accountId);
    var account = await accountrepository.GetByIdAsync(accountId);
    account.LoadFromEvents(events);
    return account;
}
```