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

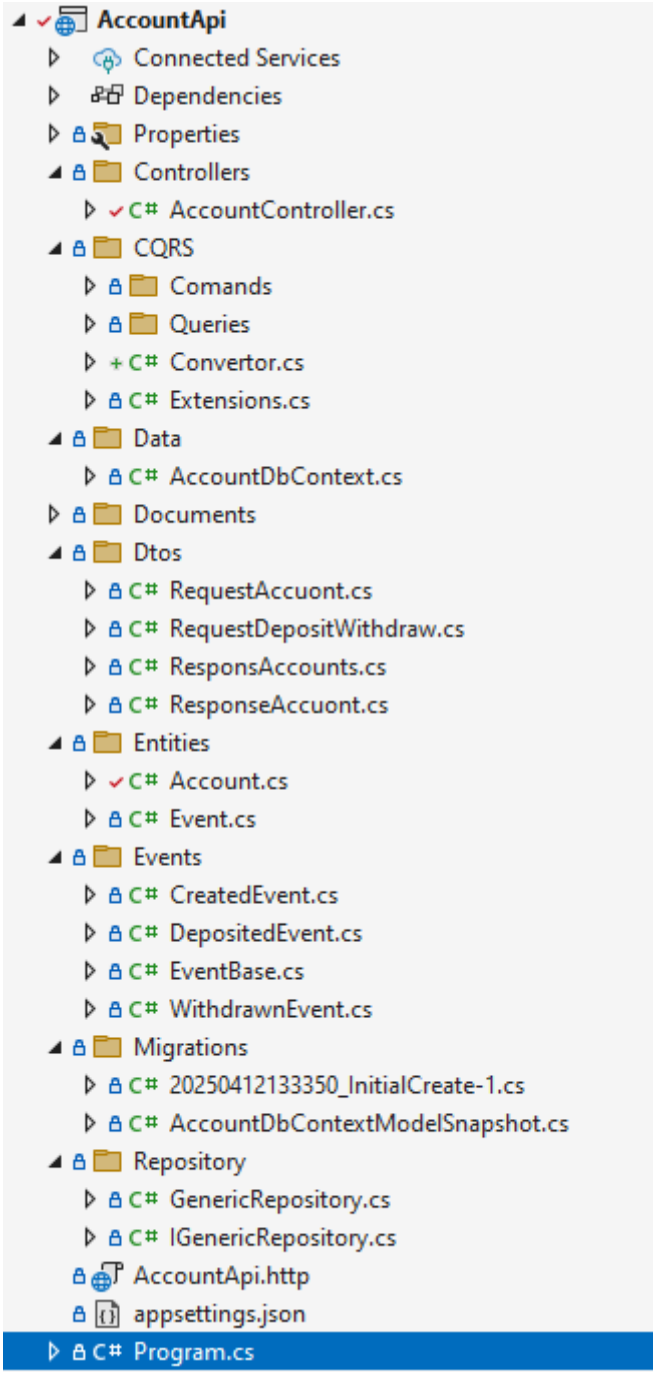
- **پیچیدگی بیشتر :نیاز به مدیریت و پردازش تعداد زیادی رویداد.**
- **اندازه بزرگ داده‌ها :ذخیره تمام رویدادها می‌تواند منجر به حجم بالای داده شود.**

- **نقص در تفسیر رویدادها :نیاز به استاندارد سازی و تفسیر صحیح رویدادها برای اطمینان از درک درست آن‌ها.**

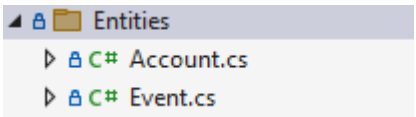
رابطه بین CQRS وEvent Sourcing

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 CQRSوEvent Sourcing ، ما به یک برنامه مدیریت حساب‌های بانکی خواهیم پرداخت. در این مثال، ساختار برنامه شامل بخش‌های زیر است:



1. اضافه کردن مدل های **account** و **Event** هست که در پوشه **Entities** اضافه شده است.



- ✓ کلاس **Account** که برای انجام عملیات های ایجاد ،واریز و برداشت و مانده گیری انجام می شود:

- تنها نکته این پراپرتی ها **Changes** هست که بعد درموردش صحبت میکنم :

```
8 references | Ghamari.Vahid, 18 hours ago | 1 author, 2 changes
public Guid Id { get; protected set; }
9 references | Ghamari.Vahid, 18 hours ago | 1 author, 2 changes
public decimal Balance { get; protected set; }
3 references | Ghamari.Vahid, 18 hours ago | 1 author, 2 changes
public string Name { get; protected set; }
1 reference | Ghamari.Vahid, 18 hours ago | 1 author, 1 change
public bool IsActive { get; protected set; }
1 reference | Ghamari.Vahid, 18 hours ago | 1 author, 1 change
public DateTime Created { get; protected set; }
4 references | Ghamari.Vahid, 18 hours ago | 1 author, 1 change
public List<EventBase> Changes => changes;
```

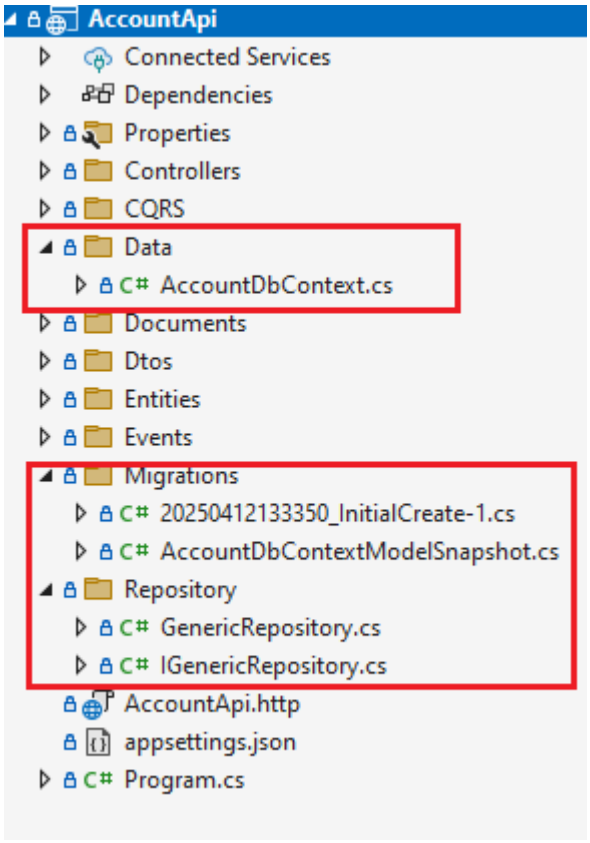
✓ کلاس **Event** هم هر رویدادی که وجود دارد را ذخیره میکند :که کل رویداد رو در پرارتی **EvnetData** به صورت **json** ذخیره میکنم

```
public class Event
{

    public Guid Id { get; set; }

    public DateTime OccurredOn { get; set; }

    public Guid AggregateId { get; set; }

    public string EventType { get; set; }
    21 references | - changes | -authors, -changes
    public string EventData { get; set; }
}
```

2. اضافه شدن **DBContext**، **Repository**و **Migration** برای **Entity** های مورد نظر:

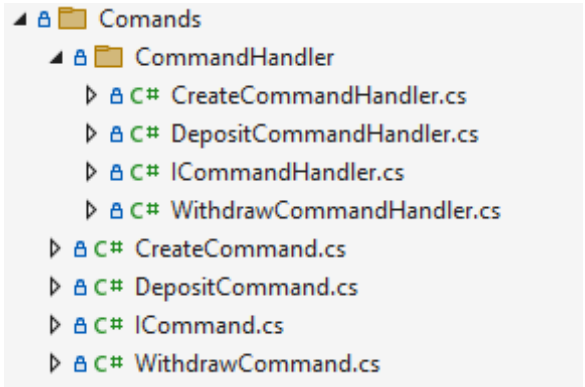
✓ نکته مهم در **DBContext** برای مدل **Account** هست با توجه به اینکه پراپرتی **Changes** لازم به ذخیره سازی در دیتابیس نیست ان را باید **Ignor** میکنم

```
0 references | Ghamari.Vahid, 19 hours ago | 1 author, 3 changes
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Account>(entity =>
    {
        entity.ToTable("Accounts");
        entity.HasKey(e => e.Id);
        entity.Ignore(e => e.Changes);

    });

    modelBuilder.Entity<Event>(entity =>
    {
        entity.ToTable("Events");
        entity.HasKey(e => e.Id);

    });

}
```



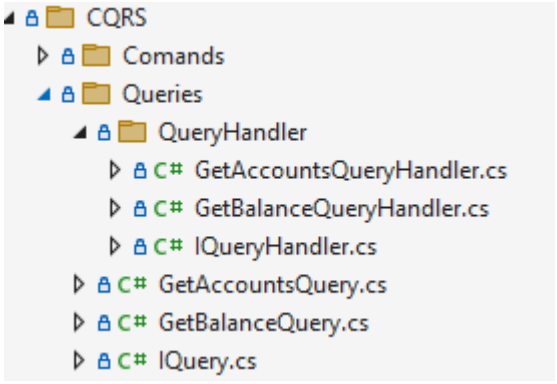
3. ایجاد Commands و Queries :

- ✓ **دستورات :(Commands) شامل عملیاتی برای تغییر وضعیت سیستم، مانند واریز و برداشت.**
- ✓ **پرسش‌ها :(Queries) شامل درخواست‌های برای دریافت داده‌ها، مانند دریافت موجودی حساب.**
- ✓ **مشخص کردن Command های مورد نظر در پوشه CQRS :ابتدا باید کامند های (Handler) سیستم مشخص شود که همزمان با ان چون قصد داریم این کامند ها خروجی هم داشته باشند DTO منتظرهم تعریف میشوند:در این مثال سه کامند مهم ایجاد CreateCommand ،واریز DepositCommand و برداشت WithdrawCommand مشخص شده است :**



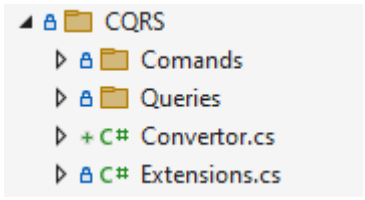
در پیاده سازی هندلرهام نکته ای خاصی نیست فقط نکته مهم این است که ما در واریز و برداشت بعد از اپدیت حساب و ایونت ها متد LoadFromEvents رو کال میکنن که بعد ان را توضیح میدهیم

- ✓ **مشخص کردن Query های مورد نظر :**



نکته مهم در پیاده سازی این دومودر استفاده از **LoadAccountAsync** است که با توجه به استفاده از **Event sourcing** لازم هست بازسازی زمان اجرا کامند(ویرایش و خذف) و کوئری ها انجام شود تا یکپارچگی سیستم درست انجام شود .

کلاس**Extentions** هست که برای افزودن کامند و کوئری ها ی تعریف شده به **builder.Services** است .




```
public class Program
{
    0 references | Ghamari.Vahid, 19 hours ago | 1 author, 5 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.AddCommandHandlers(typeof(Program));
        builder.Services.AddQueryHandlers(typeof(Program));


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

4. پیدا کردن رویدادهای سیستم که در پوشه **Events** شامل رویدادها مانند واریز، برداشت، و ایجاد حساب. ما یک کلاس بیس **Eventbase** در نظر گرفتیم و سه کلاس از آن ارث بری میکنند:

```
10 references | Ghamari.Vahid, 19 hours ago | 1 author, 3 changes
public class CreatedEvent : EventBase
{
    9 references | Ghamari.Vahid, 77 days ago | 1 author, 1 change
    public string Name { get; set; }
    9 references | Ghamari.Vahid, 4 days ago | 1 author, 1 change
    public decimal initialBalance { get; set; }
}
```

```
10 references | Ghamari.Vahid, 19 hours ago | 1 author, 2 changes
public class DepositedEvent : EventBase
{
    9 references | Ghamari.Vahid, 77 days ago | 1 author, 1 change
    public decimal Amount { get; set; }
}
```

```
public class WithdrawnEvent : EventBase
{

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

نکات بسیار مهم :

- ✓ **حالا باید به کلاس Account که رویداد ها دران اتفاق می افتند شما تغییرات شود که اول تغییر اضافه شد لیست از رویداد ها است :**

```
2 references | Ghamari.Vahid, 19 hours ago | 1 author, 1 change
public List<EventBase> Changes => changes;
```

- ✓ **مرحله دوم اضافه شدن متد ApplyChangeاز این به بعد تمام رویداها و بایندهای داخل کلاس از این متد رد می شود که با توجه به نوع event مشخص میش شود که چکاری انجام شود**
- ✓ **مرحله سوم اضافه شدن متد LoadFromEvents است که کار اصلی ان بازسازی account دریافتی از دیتابیس با توجه به event ها است.(این مورد در کامند(وازیر و برداشت) و کوئری ها استفاده می شود)**
- ✓ **مرحله چهارم اضافه شدن به لیست ایونت ها**

```
3 references | Ghamari.Vahid, 1 hour ago | 1 author, 1 change
private void ApplyChange(EventBase @event)
{
    switch (@event)
    {
        case CreatedEvent createdEvent:
            Id = Guid.NewGuid();
            Name = createdEvent.Name;
            Balance = createdEvent.initialBalance;
            IsActive = true;
            Created = DateTime.Now;
            break;
        case DepositedEvent depositedEvent:
            Balance += depositedEvent.Amount;
            break;
        case WithdrawnEvent withdrawnEvent:
            Balance -= withdrawnEvent.Amount;
            break;
    }
}
```

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

```
1 reference | Ghamari.Vahid, 17 hours ago | 1 author, 3 changes
public Account(string name, decimal balance)
{
    var @event = new CreatedEvent { Name = name, initialBalance = balance, NameOf = "Account" };
    ApplyChange(@event);
    changes.Add(@event);

}

2 references | Ghamari.Vahid, 19 hours ago | 1 author, 1 change
public void Deposit(decimal amount)
{
    if (amount <= 0)
        throw new ArgumentException("Deposit amount must be greater than zero.");

    var @event = new DepositedEvent { Amount = amount, NameOf = "Deposit" };
    ApplyChange(@event);
    changes.Add(@event);
}

1 reference | Ghamari.Vahid, 19 hours ago | 1 author, 1 change
public void Withdraw(decimal amount)
{
    if (amount <= 0)
        throw new ArgumentException("Withdrawal amount must be greater than zero.");
    if (amount > Balance)
        throw new InvalidOperationException("Insufficient funds.");

    var @event = new WithdrawnEvent { Amount = amount, NameOf = "Withdraw" };
    ApplyChange(@event);
    changes.Add(@event);
}
```

5. کنترل :در این قسمت api را تعیرف میکنم که در اینجا از ما از command و query ها استفاده میکنم :

```
[HttpPost]
0 references | Ghamari.Vahid, 18 hours ago | 1 author, 2 changes
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, Bonus = request.Bonus });
    return Ok(dd);
}


[HttpPut("DepositAsync/{AccountId}")]
0 references | Ghamari.Vahid, 18 hours ago | 1 author, 1 change
public async Task<IActionResult> DepositAsync([FromRoute] Guid AccountId, [FromBody] RequestDepositWithdraw request,
    [FromServices] ICommandHandler<DepositCommand, bool> command)
{
    var result = await command.HandlerAsync(new DepositCommand() { Amount = request.Amount, AccountId = AccountId });
    return Ok(result);
}

[HttpPut("WithdrawAsync/{AccountId}")]
0 references | Ghamari.Vahid, 18 hours ago | 1 author, 1 change
public async Task<IActionResult> WithdrawAsync([FromRoute] Guid AccountId, [FromBody] RequestDepositWithdraw request,
    [FromServices] ICommandHandler<WithdrawCommand, bool> command)
{
    var result = await command.HandlerAsync(new WithdrawCommand() { Amount = request.Amount, AccountId = AccountId });
    return Ok(result);
}

[HttpGet("GetBalanceAsync")]
0 references | Ghamari.Vahid, 76 days ago | 1 author, 1 change
public async Task<IActionResult> GetBalanceAsync([FromQuery] Guid accountId, [FromServices] IQueryHandler<GetBalanceQuery, decimal> query)
{
    var dd = await query.HandlerAsync(new GetBalanceQuery() { AccountId = accountId });
    return Ok(dd);
}

[HttpGet("GetAccountsAsync")]
0 references | Ghamari.Vahid, 18 hours ago | 1 author, 1 change
public async Task<IActionResult> GetAccountsAsync( [FromServices] IQueryHandler<GetAccountsQuery, ResponsAccounts> query)
{
    var dd = await query.HandlerAsync(new GetAccountsQuery());
    return Ok(dd);
}
```

نکته تکمیلی Program

```
2 references | Ghamari.Vahid, 20 hours ago | 1 author, 5 changes
public class Program
{
    0 references | Ghamari.Vahid, 20 hours ago | 1 author, 5 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.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 مدلSnapshot

```
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 وWithdrawAsync ، 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)، Snapshotرا 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 باSnapshot

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