PREGRADO

UNIDAD 4 | TRABAJO EN EQUIPOS MULTIDISCIPLINARIOS & WEB APPLICATION FEATURES

BEST PRACTICES

PART 1

SI653 | Aplicaciones Web

Al finalizar la unidad de aprendizaje, el estudiante comunica resultados y proceso de ingeniería aplicado para el desarrollo de una solución distribuida de aplicaciones web, con una arquitectura orientada a servicios, que satisface necesidades u oportunidades detectadas, en un ambiente ágil y colaborativo, bajo un enfoque dirigido por la innovación e inclusion.

AGENDA

.NET CORE & ASP.NET CORE

ASP.NET CORE & DDD PATTERNS

BEYOND ASP.NET CORE

Microsoft .NET Core

.NET Core is an open-source, general-purpose development platform.

You can create .NET Core apps for Windows, macOS, and Linux for x64, x86, ARM32, and ARM64 processors using multiple programming languages.

Frameworks and APIs are provided for cloud, IoT, client UI, and machine learning.

Microsoft ASP.NET Core

ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-enabled, Internet-connected apps. With ASP.NET Core, you can:

Build web apps and services, Internet of Things (IoT) apps, and mobile backends.

Use your favorite development tools on Windows, macOS, and Linux.

Deploy to the cloud or on-premises.

Run on .NET Core.

Microsoft ASP.NET Core

ASP.NET Core MVC provides features to build web APIs and web apps:

- The Model-View-Controller (MVC) pattern helps make your web APIs and web apps testable.
- Razor Pages is a page-based programming model that makes building web UI easier and more productive.
- Razor markup provides a productive syntax for Razor Pages and MVC views.
- Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files.
- Built-in support for multiple data formats and content negotiation lets your web APIs reach a broad range of clients, including browsers and mobile devices.
- Model binding automatically maps data from HTTP requests to action method parameters.
- Model validation automatically performs client-side and server-side validation.

Microsoft ASP.NET Core

ASP.NET Core supports creating RESTful services, also known as web APIs, using C#.

To handle requests, a web API uses controllers.

Controllers in a web API are classes that derive from ControllerBase.

Microsoft Entity Framework Core

Entity Framework (EF) Core is a lightweight, extensible, open source and cross-platform version of Entity Framework data access technology.

EF Core can serve as an object-relational mapper (O/RM), enabling .NET developers to work with a database using .NET objects, and eliminating the need for most of the data-access code they usually need to write.

EF Core supports many database engines

AGENDA

.NET CORE & ASP.NET CORE

ASP.NET CORE & DDD PATTERNS

BEYOND ASP.NET CORE

The Model

Data access is performed using a model. A model is made up of entity classes and a context object that represents a session with the database, allowing you to query and save data.

```
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;

namespace Intro
{
    public class BloggingContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Blogging;Integrated Security=True");
        }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Url { get; set; }
        public int Rating { get; set; }
        public List<Post> Posts { get; set; }
    }

    public class Post
    {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public Blog Blog { get; set; }
    }
}
```

Querying

Instances of entity classes are retrieved from the database using Language Integrated Query (LINQ).

```
using (var db = new BloggingContext())
{
    var blogs = db.Blogs
        .Where(b => b.Rating > 3)
        .OrderBy(b => b.Url)
        .ToList();
}
```

Saving

Data is created, deleted, and modified in the database using instances of entity classes.

```
using (var db = new BloggingContext())
{
    var blog = new Blog { Url = "http://sample.com" };
    db.Blogs.Add(blog);
    db.SaveChanges();
}
```

Domain-Driven Design

Domain-driven design (DDD) advocates modeling based on the reality of business as relevant to your use cases. In the context of building applications.

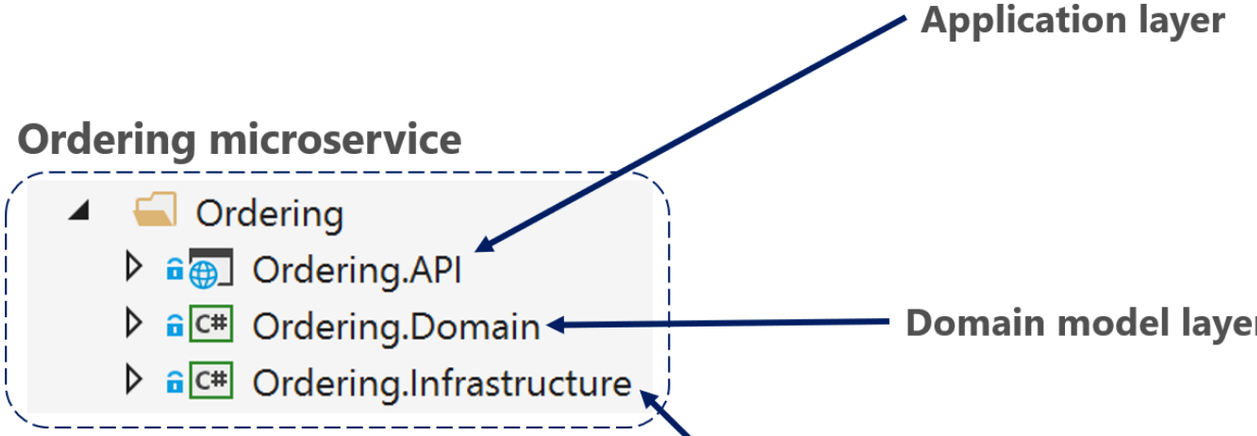
DDD talks about problems as domains. It describes independent problem areas as Bounded Contexts (each Bounded Context correlates to a microservice). It also emphasizes a common language to talk about these problems.

DDD also suggests many technical concepts and patterns, like domain entities with rich models (no anemic-domain model), value objects, aggregates and aggregate root (or root entity) rules to support the internal implementation.

.NET Core & DDD

Layered Design in .NET Core Application. Each Layer is a Visual Studio Project.

Layers in a Domain-Driven Design Microservice



Application layer

- ASP.NET Web API
- Network access to microservice
- API contracts/implementation
- Commands and command handlers
- Queries (when using an CQS approach)
  - Micro ORMs like Dapper

Domain model layer

- Domain entity model
- POCO entity classes (clean C# code)
- Domain entities with data + behavior
- DDD patterns:
  - Domain entity, aggregate
  - Aggregate root, value object
  - Repository contracts/interfaces

Infrastructure layer

- Data persistence infrastructure
  - Repository implementation
- Use of ORMs or data access API:
  - Entity Framework Core or any ORM
  - ADO.NET
  - Any NoSQL database API
- Other infrastructure implementation used from the application layer
  - Logging, cryptography, search engine, etc.

Domain Layer

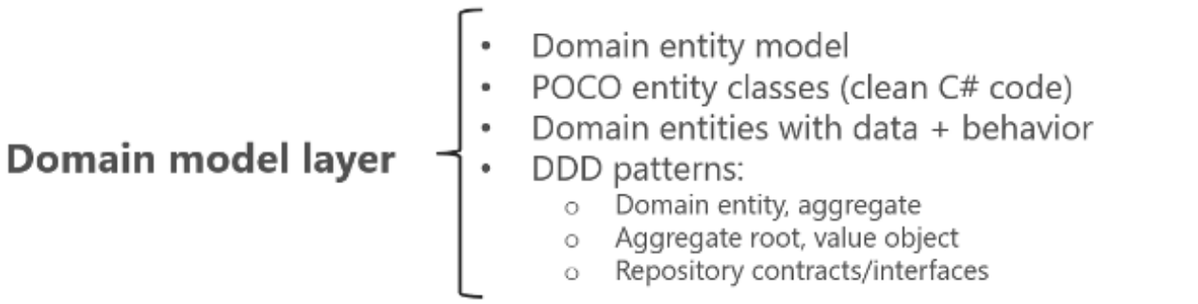
Domain Model Layer: Responsible for representing concepts of the business, information about the business situation, and business rules. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure. This layer is the heart of business software.

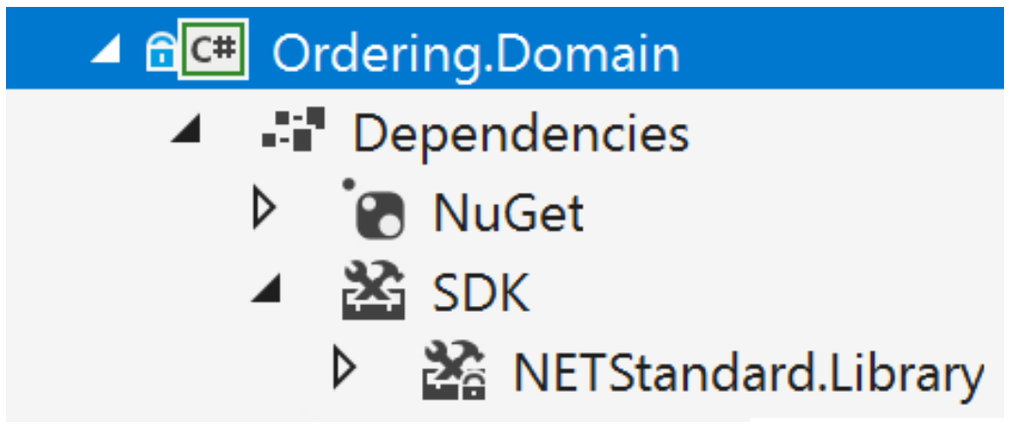
Implemented as Library.

Domain Model classes should be Plain Old CLR Objects (POCO).

Not dependent of another layer.

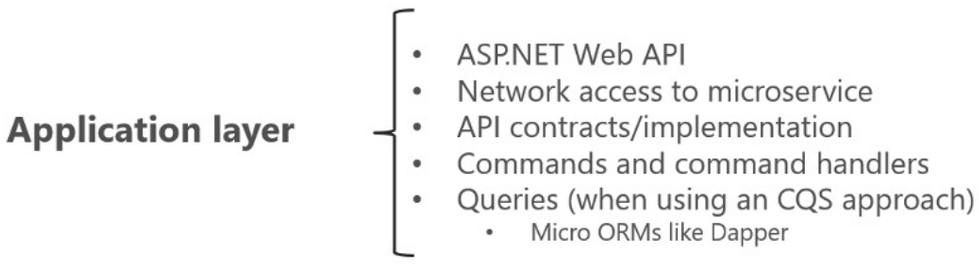
Only dependent on .NET Core Libraries or Nuget Packages, not custom libraries (data, persistence, etc.)





Application Layer

Application Layer: Defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems. The tasks this layer is responsible for are meaningful to the business or necessary for interaction with the application layers of other systems. This layer is kept thin. It does not contain business rules or knowledge, but only coordinates tasks and delegates work to collaborations of domain objects in the next layer down. It does not have state reflecting the business situation, but it can have state that reflects the progress of a task for the user or the program.

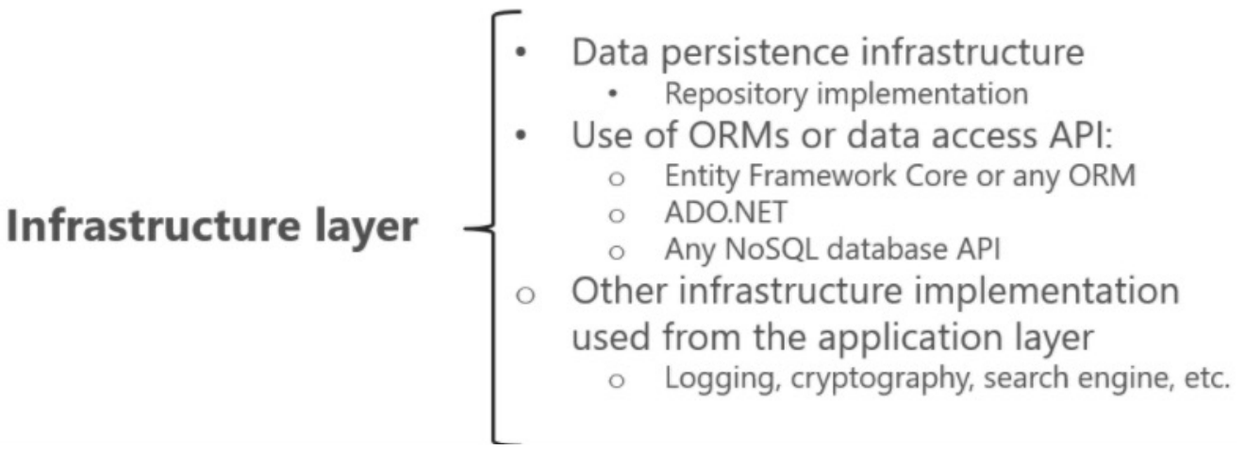


Application layer in .NET is commonly coded as an ASP.NET Core Web API project.

The ASP.NET Core Web API that represents the application layer delegates the execution of business rules to the domain model classes themselves (aggregate roots and domain entities), which will ultimately update the data within those domain entities.

Infrastructure Layer

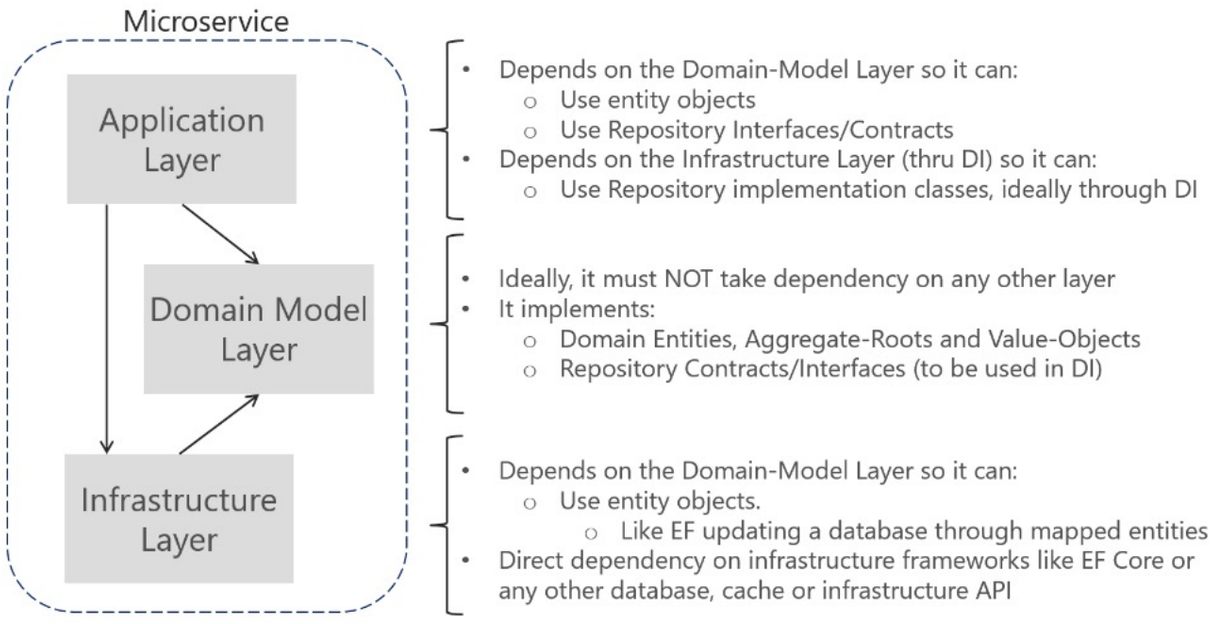
The **infrastructure layer** is how the data that is initially held in domain entities (in memory) is persisted in databases or another persistent store. An example is using Entity Framework Core code to implement the Repository pattern classes that use a DBContext to persist data in a relational database.



Dependencies between Layers

Dependencies in a DDD Service, the Application layer depends on Domain and Infrastructure, and Infrastructure depends on Domain, but Domain doesn't depend on any layer. This layer design should be independent for each microservice. As noted earlier, you can implement the most complex microservices following DDD patterns, while implementing simpler data-driven microservices (simple CRUD in a single layer) in a simpler way.

Dependencies between Layers in a Domain-Driven Design service



Domain Model

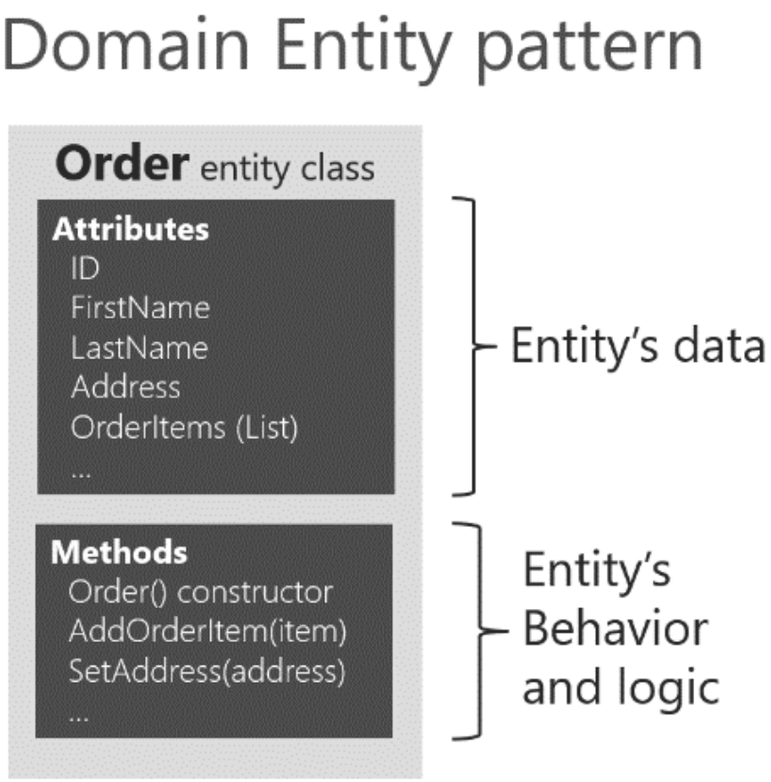
Domain Entity Pattern

> "An object primarily defined by its identity is called an Entity."
>
> - Eric Evans

Entities represent domain objects and are primarily defined by their identity, continuity, and persistence over time, and not only by the attributes that comprise them.

Domain entities must implement behavior in addition to implementing data attributes.

For example, as part of an order entity class you must have business logic and operations implemented as methods for tasks such as adding an order item, data validation, and total calculation.



Domain Model

The Aggregate Pattern

> “Aggregates should also be viewed from the perspective of being a ‘conceptual whole’ — they should reflect what designers see as a single thing.”
>
> - Eric Evans

A domain model contains clusters of different data entities and processes that can control a significant area of functionality, such as order fulfillment or inventory. A more fine-grained DDD unit is the aggregate, which describes a cluster or group of entities and behaviors that can be treated as a cohesive unit.

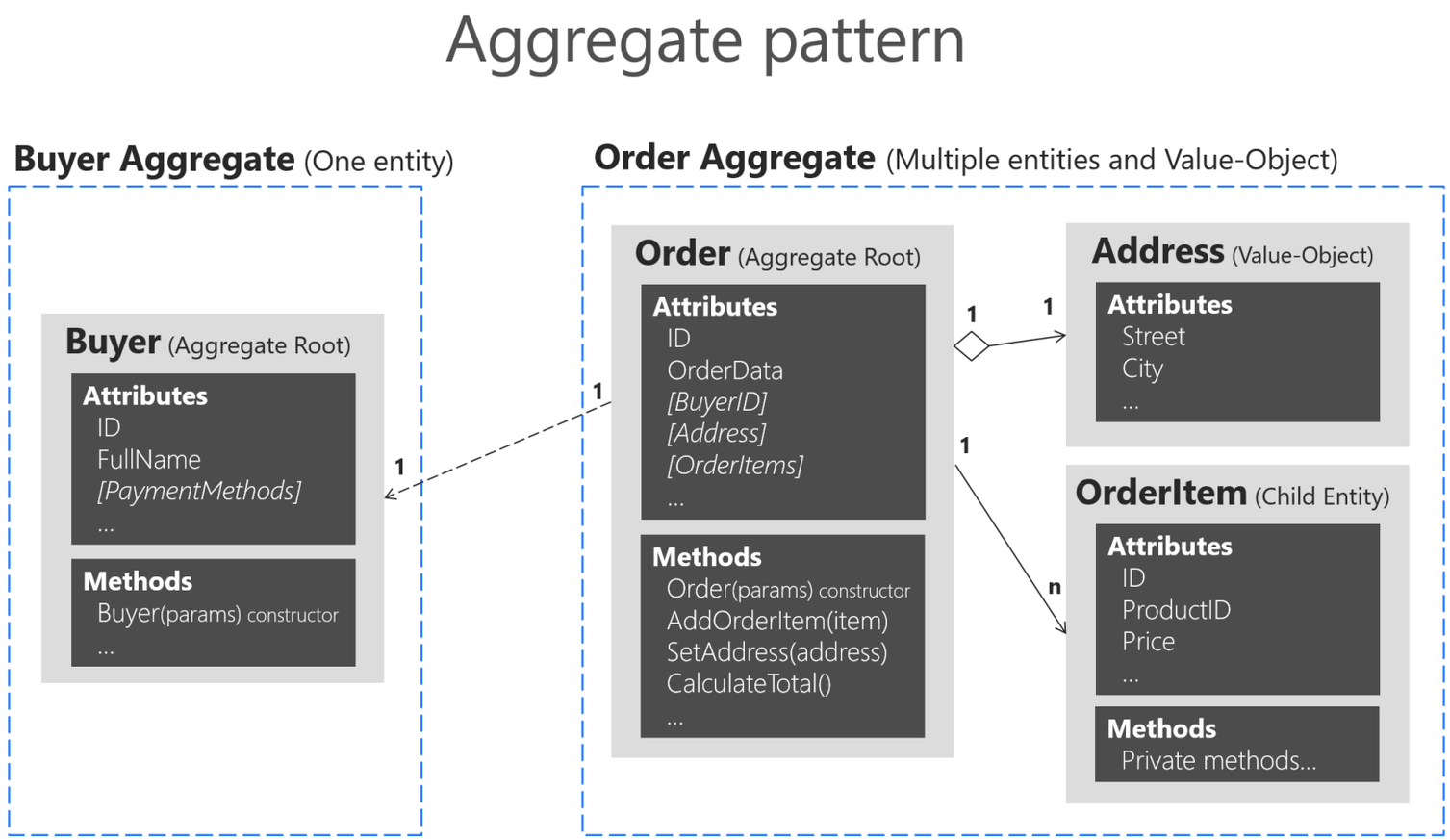
A classic example is an order that also contains a list of order items. An order item will usually be an entity. But it will be a child entity within the order aggregate, which will also contain the order entity as its root entity, typically called an aggregate root.

Domain Model

The Aggregate Root or Root Entity Pattern

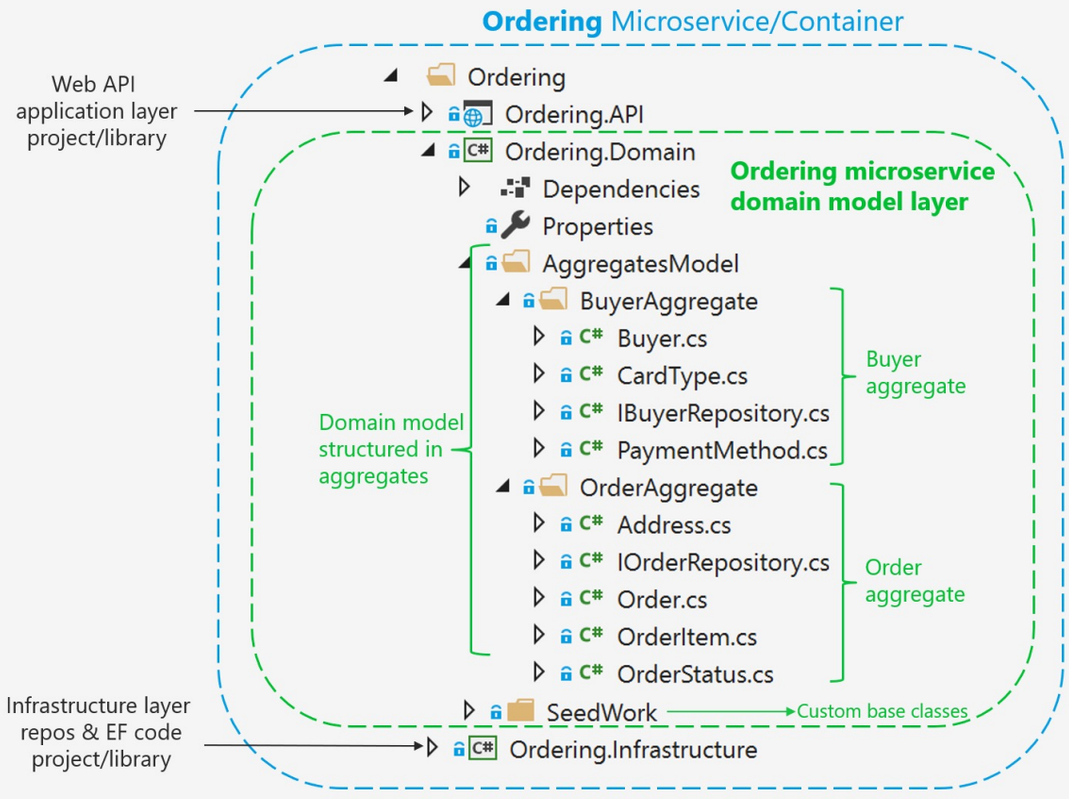
An aggregate is composed of at least one entity: the aggregate root, also called root entity or primary entity. Additionally, it can have multiple child entities and value objects, with all entities and objects working together to implement required behavior and transactions.

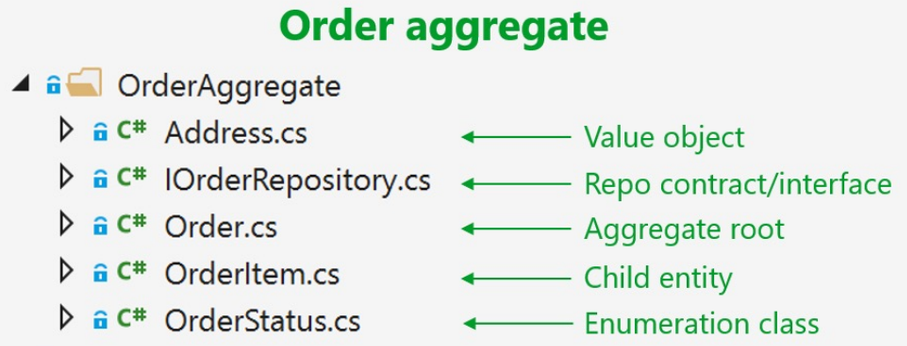
The purpose of an aggregate root is to ensure the consistency of the aggregate; it should be the only entry point for updates to the aggregate through methods or operations in the aggregate root class.



Domain Model Structure

In the ordering domain model there are two aggregates, the order aggregate and the buyer aggregate.
Each aggregate is a group of domain entities and value objects, although you could have an aggregate composed of a single domain entity (the aggregate root or root entity) as well.





Implementing domain entities as POCO classes

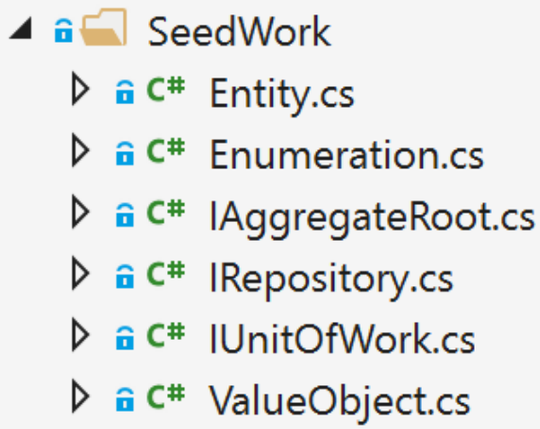
```
// Entity is a custom base class with the ID
public class Order : Entity, IAggregateRoot
{
    private DateTime _orderDate;
    public Address Address { get; private set; }
    private int? _buyerId;


    public OrderStatus OrderStatus { get; private set; }
    private int _orderStatusId;


    private string _description;
    private int? _paymentMethodId;


    private readonly List<OrderItem> _orderItems;
    public IReadOnlyCollection<OrderItem> OrderItems => _orderItems;


    public Order(string userId, Address address, int cardTypeId, string cardNumber, string cardSecurityNumber,
            string cardHolderName, DateTime cardExpiration, int? buyerId = null, int? paymentMethodId = null)
    {
        _orderItems = new List<OrderItem>();
        _buyerId = buyerId;
        _paymentMethodId = paymentMethodId;
        _orderStatusId = OrderStatus.Submitted.Id;
        _orderDate = DateTime.UtcNow;
        Address = address;


        // ...Additional code ...
    }


    public void AddOrderItem(int productId, string productName,
                            decimal unitPrice, decimal discount,
                            string pictureUrl, int units = 1)
    {
        //...
        // Domain rules/logic for adding the OrderItem to the order
        // ...


        var orderItem = new OrderItem(productId, productName, unitPrice, discount, pictureUrl, units);


        _orderItems.Add(orderItem);


    }
    // ...
    // Additional methods with domain rules/logic related to the Order aggregate
    // ...
}
```

Encapsulate data in Domain Entities

```
// WRONG ACCORDING TO DDD PATTERNS – CODE AT THE APPLICATION LAYER OR
// COMMAND HANDLERS
// Code in command handler methods or Web API controllers
//... (WRONG) Some code with business logic out of the domain classes ...
OrderItem myNewOrderItem = new OrderItem(orderId, productId, productName,
    pictureUrl, unitPrice, discount, units);


//... (WRONG) Accessing the OrderItems collection directly from the application layer
// or command handlers
myOrder.OrderItems.Add(myNewOrderItem);
//...



// RIGHT ACCORDING TO DDD--CODE AT THE APPLICATION LAYER OR COMMAND HANDLERS
// The code in command handlers or WebAPI controllers, related only to application stuff
// There is NO code here related to OrderItem object's business logic
myOrder.AddOrderItem(productId, productName, pictureUrl, unitPrice, discount, units);
// The code related to OrderItem params validations or domain rules should
// be WITHIN the AddOrderItem method.
//...
```

SeedWork

Also known as Common, SharedKernel, are base classes for Domain Model



Custom Entity Class

```
// COMPATIBLE WITH ENTITY FRAMEWORK CORE (1.1 and later)
public abstract class Entity
{
    int? _requestedHashCode;
    int _Id;
    private List<INotification> _domainEvents;
    public virtual int Id
    {
        get
        {
            return _Id;
        }
        protected set
        {
            _Id = value;
        }
    }


    public List<INotification> DomainEvents => _domainEvents;
    public void AddDomainEvent(INotification eventItem)
    {
        _domainEvents = _domainEvents ?? new List<INotification>();
        _domainEvents.Add(eventItem);
    }
    public void RemoveDomainEvent(INotification eventItem)
    {
        if (_domainEvents is null) return;
        _domainEvents.Remove(eventItem);
    }


    public bool IsTransient()
    {
        return this.Id == default(Int32);
    }


    public override bool Equals(object obj)
    {
        if (obj == null || !(obj is Entity))
            return false;
        if (Object.ReferenceEquals(this, obj))
            return true;
        if (this.GetType() != obj.GetType())
            return false;
        Entity item = (Entity)obj;
        if (item.IsTransient() || this.IsTransient())
            return false;
        else
            return item.Id == this.Id;
    }


    public override int GetHashCode()
    {
        if (!IsTransient())
        {
            if (!_requestedHashCode.HasValue)
                _requestedHashCode = this.Id.GetHashCode() ^ 31;
                        return _requestedHashCode.Value;
        }
        else
            return base.GetHashCode();
    }
    public static bool operator ==(Entity left, Entity right)
    {
        if (Object.Equals(left, null))
            return (Object.Equals(right, null));
        else
            return left.Equals(right);
    }
    public static bool operator !=(Entity left, Entity right)
    {
        return !(left == right);
    }
}
```

Repository contracts (interfaces)

Repository contracts are simply .NET interfaces that express the contract requirements of the repositories to be used for each aggregate.

The repositories themselves, with EF Core code or any other infrastructure dependencies and code (Linq, SQL, etc.), must not be implemented within the domain model

The repositories should only implement the interfaces you define in the domain model.

"Use Separated Interface to define an interface in one package but implement it in another. This way a client that needs the dependency to the interface can be completely unaware of the implementation."

- Martin Fowler

Repository contracts (interfaces)

```
// Defined at IOrderRepository.cs
public interface IOrderRepository : IRepository<Order>
{
    Order Add(Order order);

    void Update(Order order);

    Task<Order> GetAsync(int orderId);
}

// Defined at IRepository.cs (Part of the Domain Seedwork)
public interface IRepository<T> where T : IAggregateRoot
{
    IUnitOfWork UnitOfWork { get; }
}
```

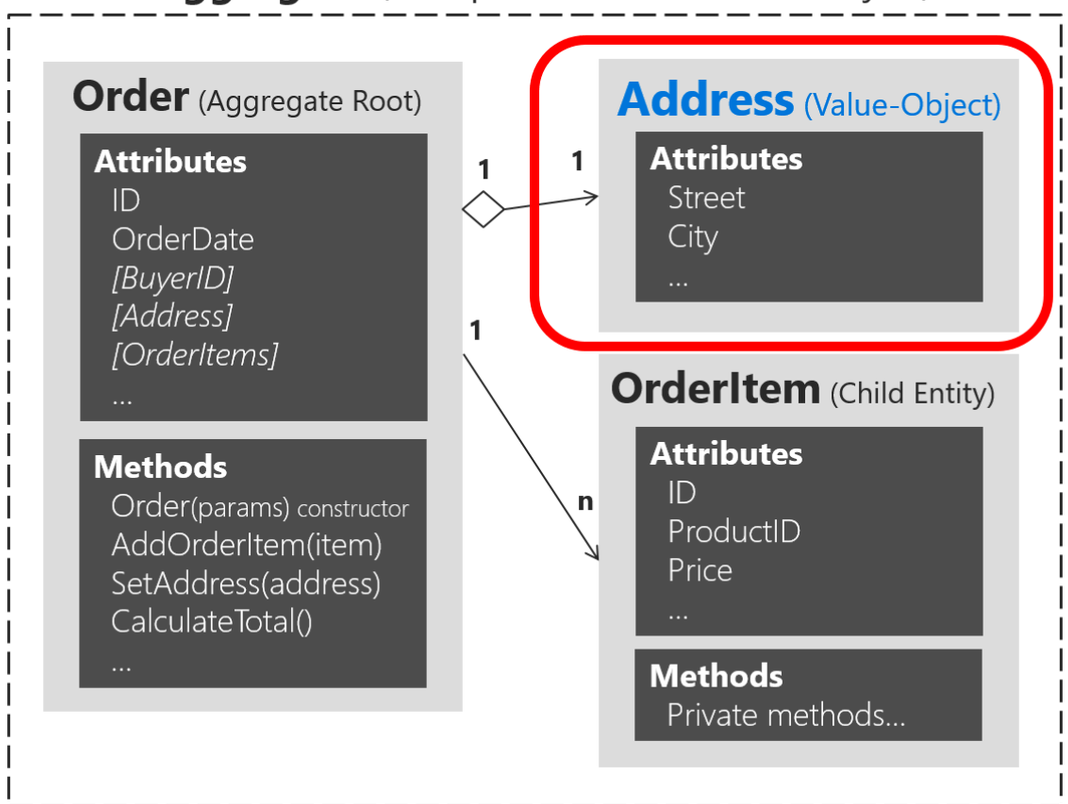
Value Objects

There are many objects and data items in a system that do not require an identity and identity tracking, such as **value objects.**

There are two main characteristics for value objects:

- They have no identity.
- They are immutable.

Value Object within Aggregate



Value Object base class implementation

```
public abstract class ValueObject
{
    protected static bool EqualOperator(ValueObject left, ValueObject right)
    {
        if (ReferenceEquals(left, null) ^ ReferenceEquals(right, null))
        {
            return false;
        }
        return ReferenceEquals(left, null) || left.Equals(right);
    }


    protected static bool NotEqualOperator(ValueObject left, ValueObject right)
    {
        return !(EqualOperator(left, right));
    }


    protected abstract IEnumerable<object> GetEqualityComponents();


    public override bool Equals(object obj)
    {
        if (obj == null || obj.GetType() != GetType())
        {
            return false;
        }


        var other = (ValueObject)obj;


        return this.GetEqualityComponents().SequenceEqual(other.GetEqualityComponents());
    }


    public override int GetHashCode()
    {
        return GetEqualityComponents()
            .Select(x => x != null ? x.GetHashCode() : 0)
            .Aggregate((x, y) => x ^ y);
    }
    // Other utility methods
}
```

Value Object base class implementation

Operator overload

```
public static bool operator ==(ValueObject one, ValueObject two)
{
    return EqualOperator(one, two);
}
```

```
public static bool operator !=(ValueObject one, ValueObject two)
{
    return NotEqualOperator(one, two);
}
```

Concrete Value Object implementation

```
public class Address : ValueObject
{
    public String Street { get; private set; }
    public String City { get; private set; }
    public String State { get; private set; }
    public String Country { get; private set; }
    public String ZipCode { get; private set; }


    private Address() { }


    public Address(string street, string city, string state, string country, string zipcode)
    {
        Street = street;
        City = city;
        State = state;
        Country = country;
        ZipCode = zipcode;
    }


    protected override IEnumerable<object> GetAtomicValues()
    {
        // Using a yield return statement to return each element one at a time
        yield return Street;
        yield return City;
        yield return State;
        yield return Country;
        yield return ZipCode;
    }
}
```

Value Objects Persistence

```
/ Part of the OrderingContext.cs class at the Ordering.Infrastructure project
//
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new ClientRequestEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new PaymentMethodEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new OrderEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new OrderItemEntityTypeConfiguration());
    //...Additional type configurations
}

// Part of the OrderEntityTypeConfiguration.cs class
//
public void Configure(EntityTypeBuilder<Order> orderConfiguration)
{
    orderConfiguration.ToTable("orders", OrderingContext.DEFAULT_SCHEMA);
    orderConfiguration.HasKey(o => o.Id);
    orderConfiguration.Ignore(b => b.DomainEvents);
    orderConfiguration.Property(o => o.Id)
        .ForSqlServerUseSequenceHiLo("orderseq", OrderingContext.DEFAULT_SCHEMA);


    //Address value object persisted as owned entity in EF Core 2.0
    orderConfiguration.OwnsOne(o => o.Address);

    //You can append the Property().HasColumnName() fluent method to rename Value Object columns.
    orderConfiguration.OwnsOne(p => p.Address)
                            .Property(p=>p.Street).HasColumnName("ShippingStreet");


    orderConfiguration.OwnsOne(p => p.Address)
                            .Property(p=>p.City).HasColumnName("ShippingCity");

    orderConfiguration.Property<DateTime>("OrderDate").IsRequired();


    //...Additional validations, constraints and code...
    //...
}
```

Value Objects Persistence

Chaining Value Objects

```
orderConfiguration.OwnsOne(p => p.OrderDetails, cb =>
    {
        cb.OwnsOne(c => c.BillingAddress);
        cb.OwnsOne(c => c.ShippingAddress);
    });
//...
//...
public class Order
{
    public int Id { get; set; }
    public OrderDetails OrderDetails { get; set; }
}


public class OrderDetails
{
    public Address BillingAddress { get; set; }
    public Address ShippingAddress { get; set; }
}


public class Address
{
    public string Street { get; set; }
    public string City { get; set; }
}
```

Implement validations in the domain model layer

Validate conditions and throw exceptions

```
public void SetAddress(string line1, string line2,
    string city, string state, int zip)
{
    _shippingAddress.line1 = line1 ?? throw new ...
    _shippingAddress.line2 = line2;
    _shippingAddress.city = city ?? throw new ...
    _shippingAddress.state = (IsValid(state) ? state : throw new …);
}
```

Domain events

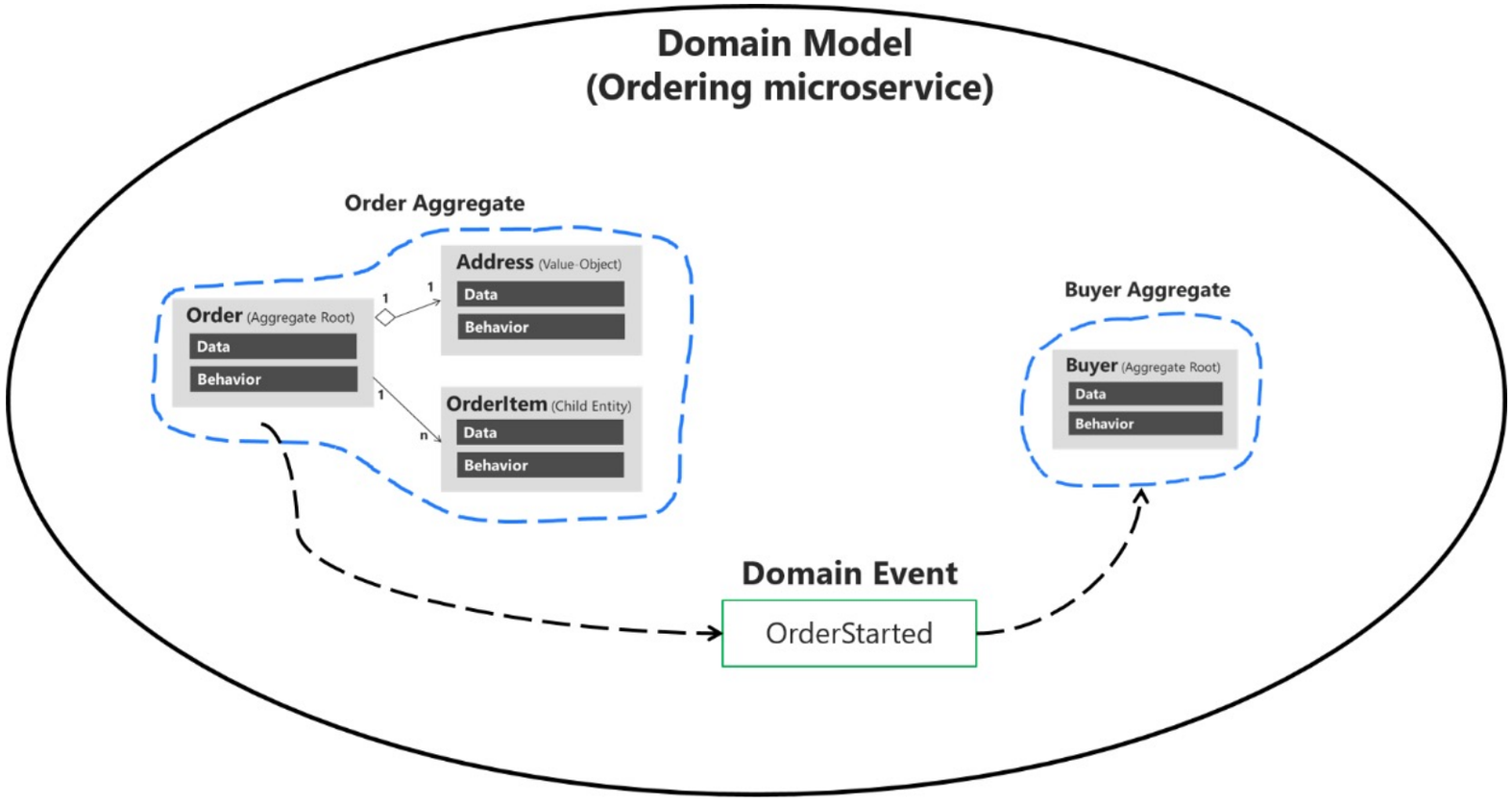
An event is something that has happened in the past.

A domain event is something that happened in the domain that you want other parts of the same domain (in-process) to be aware of.

The notified parts usually react somehow to the events.

When the user initiates an order, the Order Aggregate sends an OrderStarted domain event.

The OrderStarted domain event is handled by the Buyer Aggregate to create a Buyer object in the ordering microservice, based on the original user info from the identity microservice (with information provided in the CreateOrder command).



Domain events

With domain events, you can create a fine-grained and decoupled implementation by segregating responsibilities using this approach:

1. Send a command (for example, CreateOrder).
2. Receive the command in a command handler.

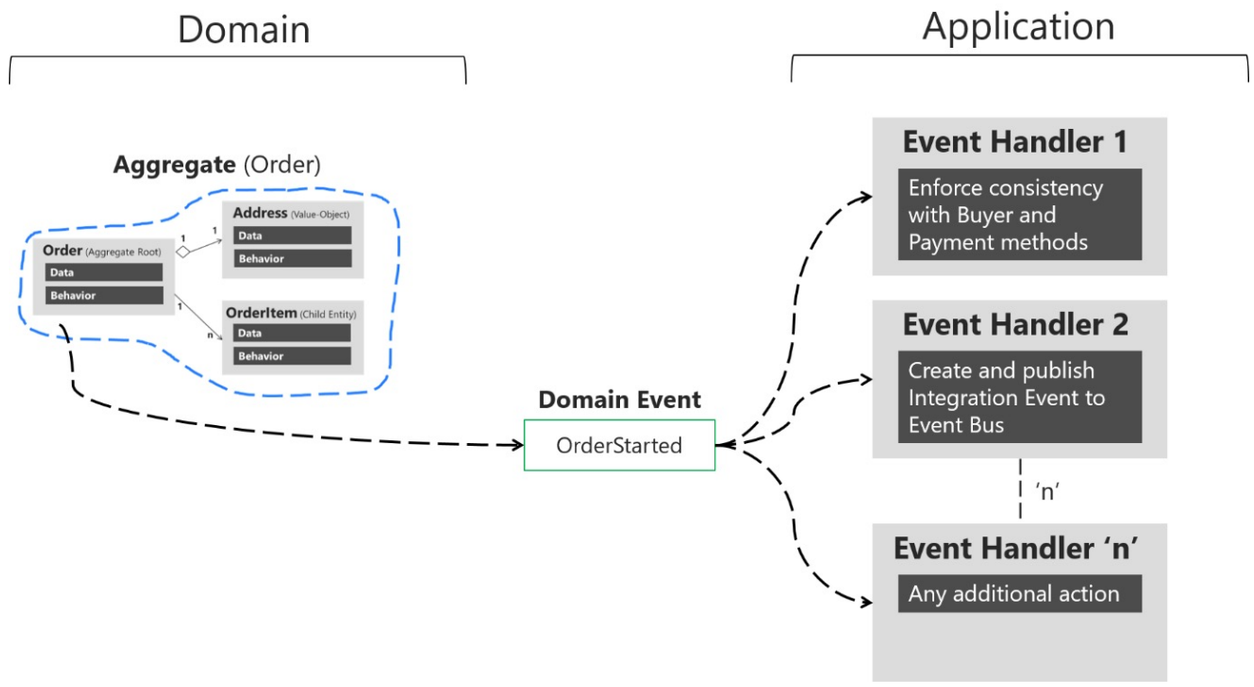
   Execute a single aggregate's transaction.

   (Optional) Raise domain events for side effects (for example, OrderStartedDomainEvent).
3. Handle domain events (within the current process) that will execute an open number of side effects in multiple aggregates or application actions. For example:

   Verify or create buyer and payment method.

   Create and send a related integration event to the event bus to propagate states across microservices or trigger external actions like sending an email to the buyer.

   Handle other side effects.



Implementing domain events

```
public class OrderStartedDomainEvent : INotification
{
    public string UserId { get; }
    public int CardTypeId { get; }
    public string CardNumber { get; }
    public string CardSecurityNumber { get; }
    public string CardHolderName { get; }
    public DateTime CardExpiration { get; }
    public Order Order { get; }

    public OrderStartedDomainEvent(Order order,
                                   int cardTypeId, string cardNumber,
                                   string cardSecurityNumber, string cardHolderName,
                                   DateTime cardExpiration)
    {
        Order = order;
        CardTypeId = cardTypeId;
        CardNumber = cardNumber;
        CardSecurityNumber = cardSecurityNumber;
        CardHolderName = cardHolderName;
        CardExpiration = cardExpiration;
    }
}
```

Deferred approach to raise and dispatch events

```
public abstract class Entity
{
    //...
    private List<INotification> _domainEvents;
    public List<INotification> DomainEvents => _domainEvents;


    public void AddDomainEvent(INotification eventItem)
    {
        _domainEvents = _domainEvents ?? new List<INotification>();
        _domainEvents.Add(eventItem);
    }


    public void RemoveDomainEvent(INotification eventItem)
    {
        _domainEvents?.Remove(eventItem);
    }
    //... Additional code
}
```

Deferred approach to raise and dispatch events

When you want to raise an event, you just add it to the event collection from code at any method of the aggregate-root entity.

```
var orderStartedDomainEvent = new OrderStartedDomainEvent(this, //Order object
                                                          cardTypeId, cardNumber,
                                                          cardSecurityNumber,
                                                          cardHolderName,
                                                          cardExpiration);
this.AddDomainEvent(orderStartedDomainEvent);
```

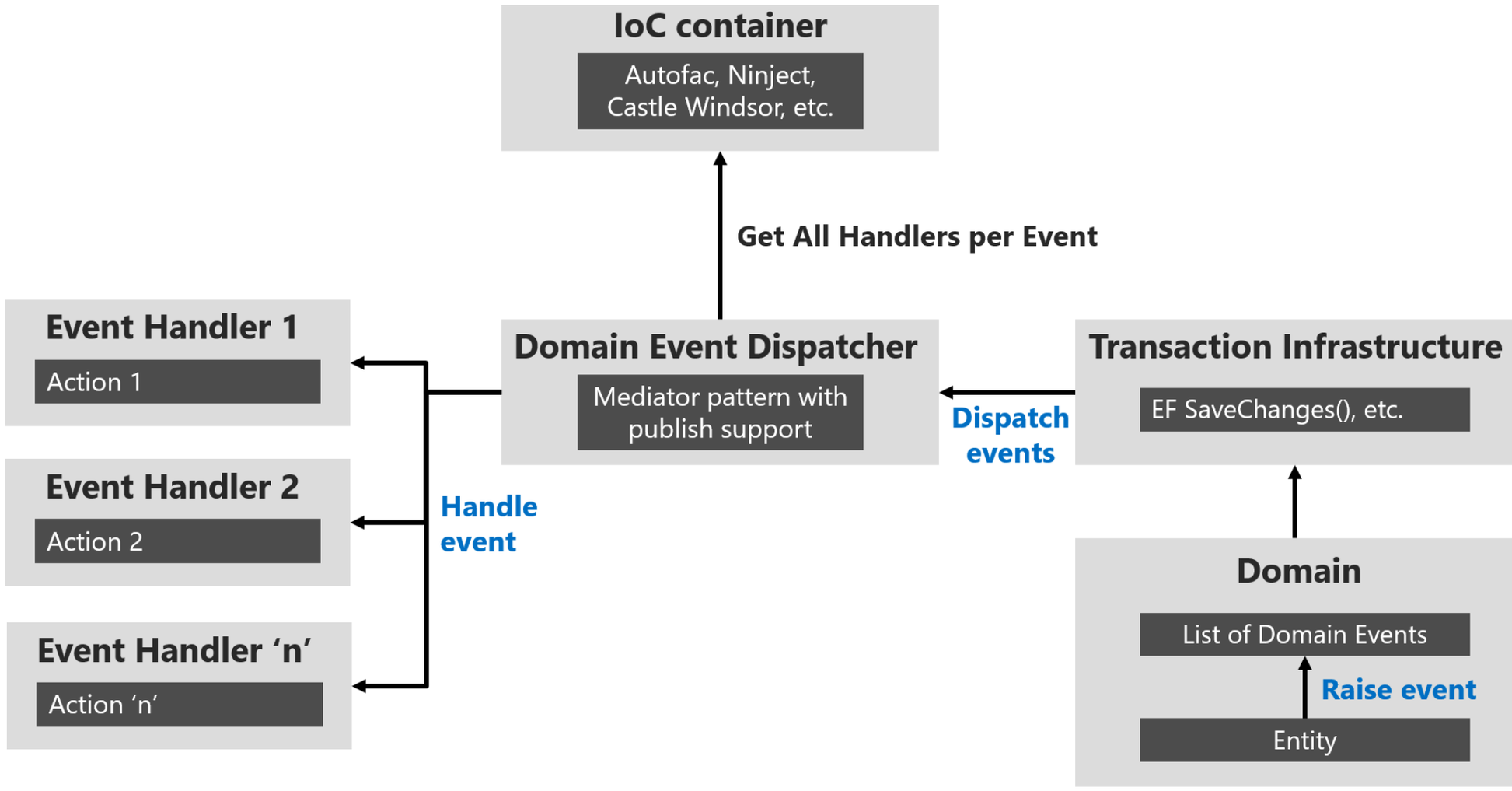
Deferred approach to raise and dispatch events

Dispatch the events later on, when you commit the transaction to the database.

```
// EF Core DbContext
public class OrderingContext : DbContext, IUnitOfWork
{
    // ...
    public async Task<bool> SaveEntitiesAsync(CancellationToken cancellationToken = default(CancellationToken))
    {
        // Dispatch Domain Events collection.
        // Choices:
        // A) Right BEFORE committing data (EF SaveChanges) into the DB. This makes
        // a single transaction including side effects from the domain event
        // handlers that are using the same DbContext with Scope lifetime
        // B) Right AFTER committing data (EF SaveChanges) into the DB. This makes
        // multiple transactions. You will need to handle eventual consistency and
        // compensatory actions in case of failures.
        await _mediator.DispatchDomainEventsAsync(this);


        // After this line runs, all the changes (from the Command Handler and Domain
        // event handlers) performed through the DbContext will be committed
        var result = await base.SaveChangesAsync();
    }
}
```

Domain event dispatcher

Mapping from events to event handlers.



Deferred approach to raise and dispatch events

Register the event handler types in your IoC container.

```
public class MediatorModule : Autofac.Module
{
    protected override void Load(ContainerBuilder builder)
    {
        // Other registrations ...
        // Register the DomainEventHandler classes (they implement IAsyncNotificationHandler<>)
        // in assembly holding the Domain Events
        builder.RegisterAssemblyTypes(typeof(ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler)
                                        .GetTypeInfo().Assembly)
                                          .AsClosedTypesOf(typeof(IAsyncNotificationHandler<>));
        // Other registrations ...
    }
}
```

Handling domain events

```
public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
                  : INotificationHandler<OrderStartedDomainEvent>
{
    private readonly ILoggerFactory _logger;
    private readonly IBuyerRepository<Buyer> _buyerRepository;
    private readonly IIdentityService _identityService;

    public ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler(
        ILoggerFactory logger,
        IBuyerRepository<Buyer> buyerRepository,
        IIdentityService identityService)
    {
        // ...Parameter validations...
    }

    public async Task Handle(OrderStartedDomainEvent orderStartedEvent)
    {
        var cardTypeId = (orderStartedEvent.CardTypeId != 0) ? orderStartedEvent.CardTypeId : 1;
        var userGuid = _identityService.GetUserIdentity();
        var buyer = await _buyerRepository.FindAsync(userGuid);
        bool buyerOriginallyExisted = (buyer == null) ? false : true;

        if (!buyerOriginallyExisted)
        {
            buyer = new Buyer(userGuid);
        }

        buyer.VerifyOrAddPaymentMethod(cardTypeId,
                                       $"Payment Method on {DateTime.UtcNow}",
                                       orderStartedEvent.CardNumber,
                                       orderStartedEvent.CardSecurityNumber,
                                       orderStartedEvent.CardHolderName,
                                       orderStartedEvent.CardExpiration,
                                       orderStartedEvent.Order.Id);

        var buyerUpdated = buyerOriginallyExisted ? _buyerRepository.Update(buyer)
                                                  : _buyerRepository.Add(buyer);

        await _buyerRepository.UnitOfWork
                .SaveEntitiesAsync();

        // Logging code using buyerUpdated info, etc.
    }
}
```

The event handler usually implements application layer code that uses infrastructure repositories to obtain the required additional aggregates and to execute side-effect domain logic.

Propagating domain events

Some alternatives

MediatR

In-process messaging with no dependencies

https://github.com/jbogard/MediatR

RabbitMQ

Message Broker

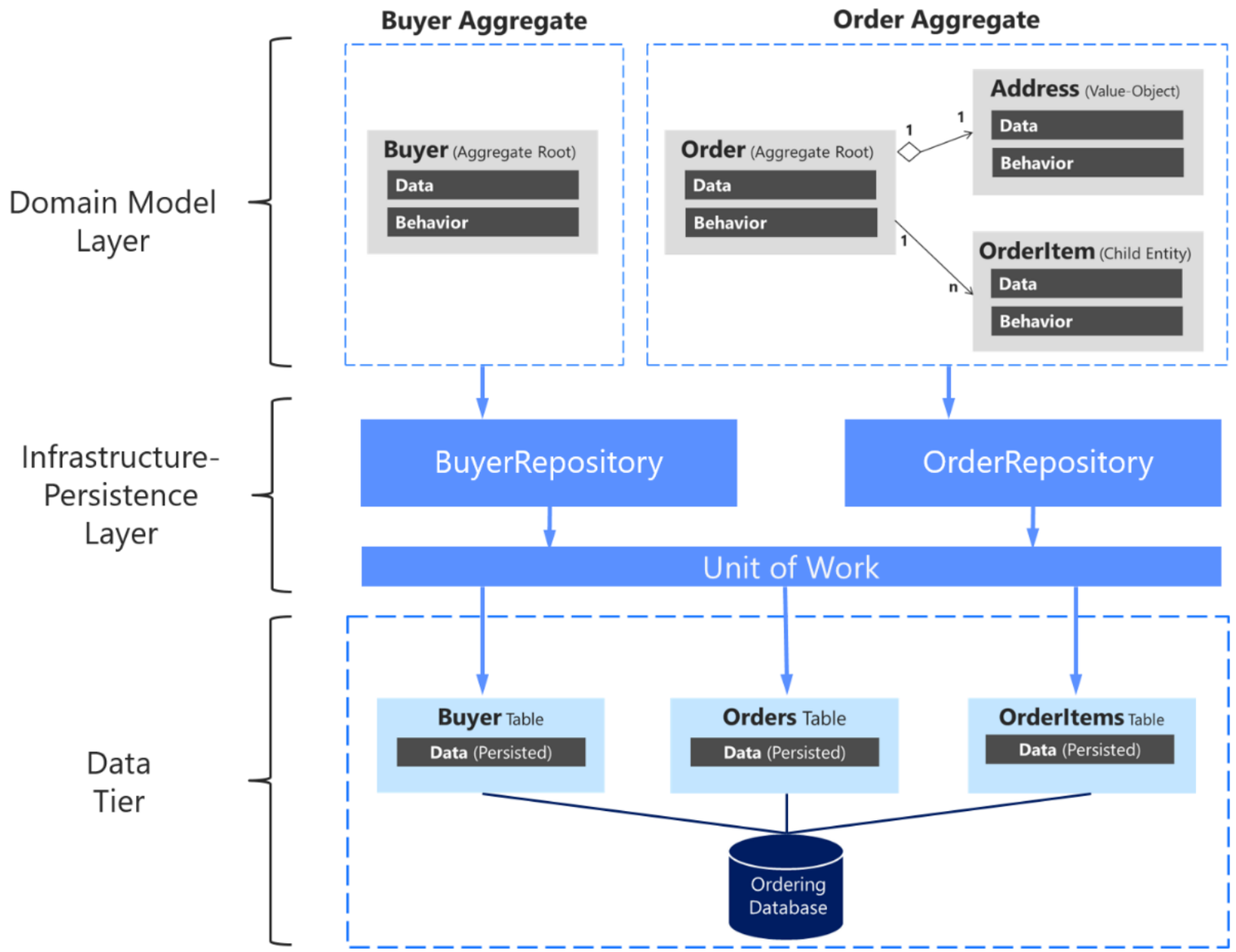
https://www.rabbitmq.com/

Infrastructure persistence layer

Entity Framework DbContext implements both Repository and Unit of Work patterns.

The repository pattern

Define one repository per aggregate.



Enforce one aggregate root per repository

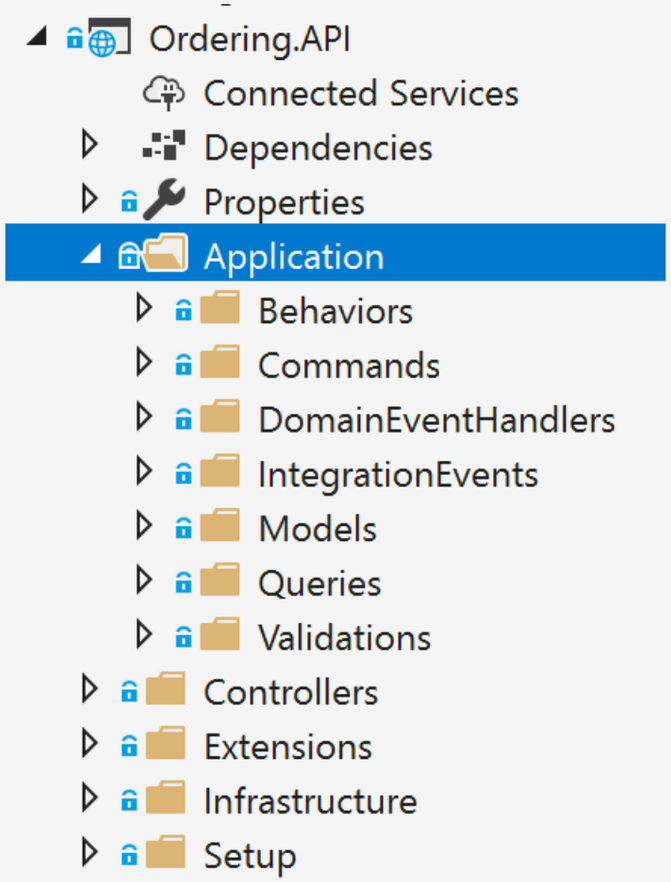
```
public interface IRepository<T> where T : IAggregateRoot
{
    //....
}


public interface IOrderRepository : IRepository<Order>
{
    Order Add(Order order);
    // ...
}


namespace
Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories
{
    public class OrderRepository : IOrderRepository
    {
      // ...
    }
}
```

Implement custom repositories

```
// using directives...
namespace Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories
{
    public class BuyerRepository : IBuyerRepository
    {
        private readonly OrderingContext _context;
        public IUnitOfWork UnitOfWork
        {
            get
            {
                return _context;
            }
        }


        public BuyerRepository(OrderingContext context)
        {
            _context = context ?? throw new ArgumentNullException(nameof(context));
        }


        public Buyer Add(Buyer buyer)
        {
            return _context.Buyers.Add(buyer).Entity;
        }


        public async Task<Buyer> FindAsync(string buyerIdentityGuid)
        {
            var buyer = await _context.Buyers
                .Include(b => b.Payments)
                .Where(b => b.FullName == buyerIdentityGuid)
                .SingleOrDefaultAsync();


            return buyer;
        }
    }
}
```

Application layer

Application layer can be implemented as part of the artifact (assembly) you are building, such as within a Web API project or an MVC web app project



Implement custom repositories

```
public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IIdentityService _identityService;
    private readonly IMediator _mediator;

    // Using DI to inject infrastructure persistence Repositories
    public CreateOrderCommandHandler(IMediator mediator,
                                     IOrderRepository orderRepository,
                                     IIdentityService identityService)
    {
        _orderRepository = orderRepository ?? throw new ArgumentNullException(nameof(orderRepository));
        _identityService = identityService ?? throw new ArgumentNullException(nameof(identityService));
        _mediator = mediator ?? throw new ArgumentNullException(nameof(mediator));
    }

    public async Task<bool> Handle(CreateOrderCommand message)
    {
        // Create the Order AggregateRoot
        // Add child entities and value objects through the Order aggregate root
        // methods and constructor so validations, invariants, and business logic
        // make sure that consistency is preserved across the whole aggregate
        var address = new Address(message.Street, message.City, message.State,
                                  message.Country, message.ZipCode);
        var order = new Order(message.UserId, address, message.CardTypeId,
                              message.CardNumber, message.CardSecurityNumber,
                              message.CardHolderName, message.CardExpiration);

        foreach (var item in message.OrderItems)
        {
            order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice,
                               item.Discount, item.PictureUrl, item.Units);
        }

        _orderRepository.Add(order);

        return await _orderRepository.UnitOfWork
            .SaveEntitiesAsync();
    }
}
```

ASP.NET Core includes a simple built-in IoC container (represented by the IServiceProvider interface) that supports constructor injection by default, and ASP.NET makes certain services available through DI.

Register dependency implementation types and interfaces or abstractions

Use the built-in IoC container provided by ASP.NET Core.

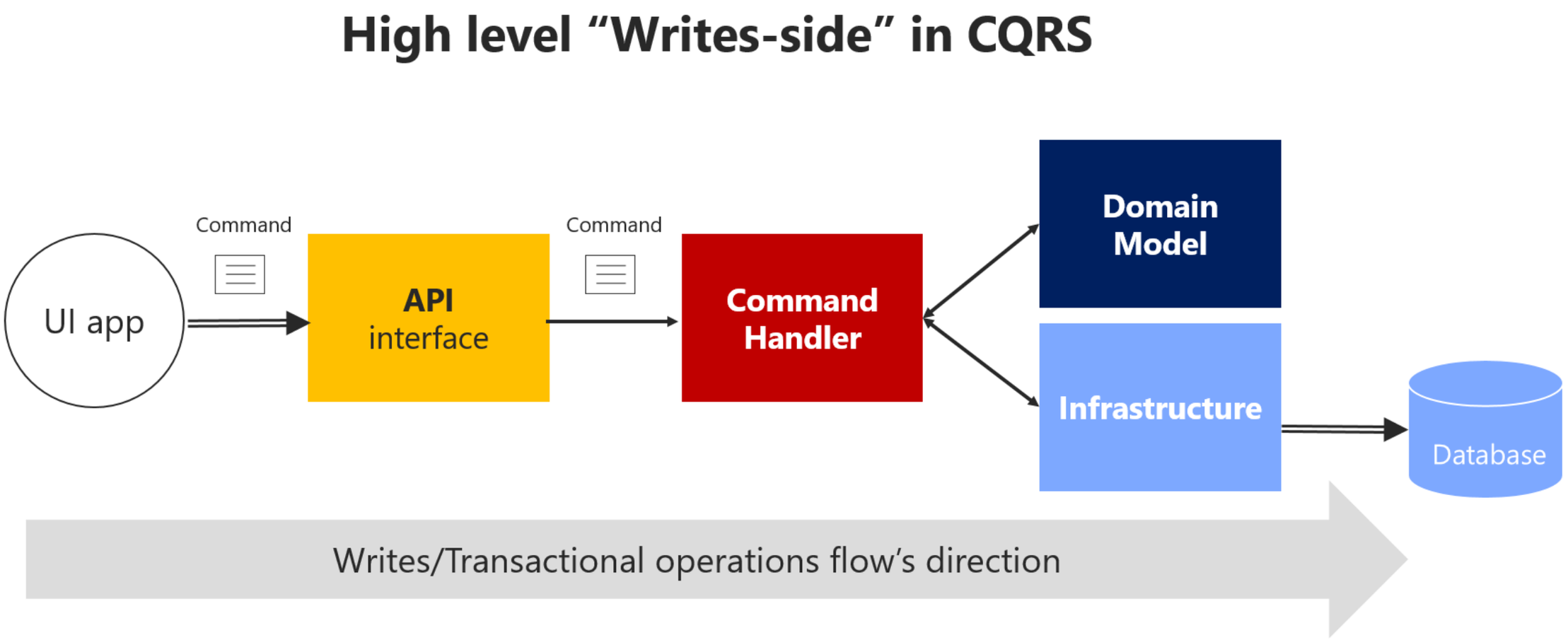
```
// Registration of types into ASP.NET Core built-in container
public void ConfigureServices(IServiceCollection services)
{
    // Register out-of-the-box framework services.
    services.AddDbContext<CatalogContext>(c =>
        c.UseSqlServer(Configuration["ConnectionString"]),
        ServiceLifetime.Scoped);


    services.AddMvc();
    // Register custom application dependencies.
    services.AddScoped<IMyCustomRepository, MyCustomSQLRepository>();
}
```

Use Autofac as an IoC container

When using Autofac (https://autofac.org/) you tipically register the types via modules, which allow you to split the registration types between multiple files depending on where the types are.

```
public class ApplicationModule : Autofac.Module
{
    public string QueriesConnectionString { get; }
    public ApplicationModule(string qconstr)
    {
        QueriesConnectionString = qconstr;
    }


    protected override void Load(ContainerBuilder builder)
    {
        builder.Register(c => new OrderQueries(QueriesConnectionString))
            .As<IOrderQueries>().InstancePerLifetimeScope();

        builder.RegisterType<BuyerRepository>()
           .As<IBuyerRepository>().InstancePerLifetimeScope();

        builder.RegisterType<OrderRepository>()
           .As<IOrderRepository>().InstancePerLifetimeScope();

        builder.RegisterType<RequestManager>()
          .As<IRequestManager>().InstancePerLifetimeScope();
    }
}
```

Implement the Command and Command Handler patterns



The Command pattern is intrinsically related to the CQRS pattern. CQRS has two sides. The first area is queries, using an ORM. The second area is commands, which are the starting point for transactions, and the input channel from outside the service

The command class

A command is a request for the system to perform an action that changes the state of the system. A command is a special kind of Data Transfer Object (DTO), one that is specifically used to request changes or transactions.

```
// DDD and CQRS patterns comment: Note that it is
// recommended to implement immutable Commands
// In this case, its immutability is achieved by having all the setters as
// private plus only being able to update the data just once, when
// creating the object through its constructor.
// References on Immutable Commands:
// http://cqrs.nu/Faq
// https://docs.spine3.org/motivation/immutability.html
// http://blog.gauffin.org/2012/06/griffin-container-introducing-command-support/
// https://docs.microsoft.com/dotnet/csharp/programming-guide/structs/
// classes-and-how-to-implement-a-lightweight-class-with-auto
// -implemented-properties


[DataContract]
public class CreateOrderCommand
    : IRequest<bool>
{
    [DataMember]
    private readonly List<OrderItemDTO> _orderItems;
    [DataMember]
    public string UserId { get; private set; }
    [DataMember]
    public string UserName { get; private set; }
    [DataMember]
    public string City { get; private set; }
    [DataMember]
    public string Street { get; private set; }
    [DataMember]
    public string State { get; private set; }
    [DataMember]
    public string Country { get; private set; }
    [DataMember]
    public string ZipCode { get; private set; }
    [DataMember]
    public string CardNumber { get; private set; }
    [DataMember]
    public string CardHolderName { get; private set; }
    [DataMember]
    public DateTime CardExpiration { get; private set; }
    [DataMember]
    public string CardSecurityNumber { get; private set; }
    [DataMember]
    public int CardTypeId { get; private set; }
    [DataMember]
    public IEnumerable<OrderItemDTO> OrderItems => _orderItems;
    public CreateOrderCommand()
    {
        _orderItems = new List<OrderItemDTO>();
    }


    public CreateOrderCommand(List<BasketItem> basketItems,
        string userId, string userName, string city,
        string street, string state, string country, string zipcode,
        string cardNumber, string cardHolderName, DateTime cardExpiration,
        string cardSecurityNumber, int cardTypeId) : this()
    {
        _orderItems = basketItems.ToOrderItemsDTO().ToList();
        UserId = userId;
        UserName = userName;
        City = city;
        Street = street;
        State = state;
        Country = country;
        ZipCode = zipcode;
        CardNumber = cardNumber;
        CardHolderName = cardHolderName;
        CardExpiration = cardExpiration;
        CardSecurityNumber = cardSecurityNumber;
        CardTypeId = cardTypeId;
        CardExpiration = cardExpiration;
    }

    public class OrderItemDTO
    {
        public int ProductId { get; set; }
        public string ProductName { get; set; }
        public decimal UnitPrice { get; set; }
        public decimal Discount { get; set; }
        public int Units { get; set; }
        public string PictureUrl { get; set; }
    }
}
```

The command class

Many command classes can be simple, requiring only a few fields about some state that needs to be changed.

```
[DataContract]
public class UpdateOrderStatusCommand
    :IRequest<bool>
{
    [DataMember]
    public string Status { get; private set; }

    [DataMember]
    public string OrderId { get; private set; }

    [DataMember]
    public string BuyerIdentityGuid { get; private set; }
}
```

The command handler class

You should implement a specific command handler class for each command. That is how the pattern works, and it's where you'll use the command object, the domain objects, and the infrastructure repository objects. The command handler is in fact the heart of the application layer in terms of CQRS and DDD.

```
public class CreateOrderCommandHandler
        : IRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IIdentityService _identityService;
    private readonly IMediator _mediator;
    private readonly IOrderingIntegrationEventService _orderingIntegrationEventService;
    private readonly ILogger<CreateOrderCommandHandler> _logger;


    // Using DI to inject infrastructure persistence Repositories
    public CreateOrderCommandHandler(IMediator mediator,
        IOrderingIntegrationEventService orderingIntegrationEventService,
        IOrderRepository orderRepository,
        IIdentityService identityService,
        ILogger<CreateOrderCommandHandler> logger)
    {
        _orderRepository = orderRepository ?? throw new ArgumentNullException(nameof(orderRepository));
        _identityService = identityService ?? throw new ArgumentNullException(nameof(identityService));
        _mediator = mediator ?? throw new ArgumentNullException(nameof(mediator));
        _orderingIntegrationEventService = orderingIntegrationEventService ?? throw new ArgumentNullException(nameof(orderingIntegrationEventService));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }


    public async Task<bool> Handle(CreateOrderCommand message, CancellationToken cancellationToken)
    {
        // Add Integration event to clean the basket
        var orderStartedIntegrationEvent = new OrderStartedIntegrationEvent(message.UserId);
        await _orderingIntegrationEventService.AddAndSaveEventAsync(orderStartedIntegrationEvent);

        // Add/Update the Buyer AggregateRoot
        // DDD patterns comment: Add child entities and value-objects through the Order Aggregate-Root
        // methods and constructor so validations, invariants and business logic
        // make sure that consistency is preserved across the whole aggregate
        var address = new Address(message.Street, message.City, message.State, message.Country, message.ZipCode);
        var order = new Order(message.UserId, message.UserName, address, message.CardTypeId, message.CardNumber,
                              message.CardSecurityNumber, message.CardHolderName, message.CardExpiration);

        foreach (var item in message.OrderItems)
        {
            order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice, item.Discount, item.PictureUrl, item.Units);
        }
        _logger.LogInformation("----- Creating Order - Order: {@Order}", order);
        _orderRepository.Add(order);
        return await _orderRepository.UnitOfWork.SaveEntitiesAsync(cancellationToken);
    }
}
```

Implement the command process pipeline with a mediator pattern (MediatR)

In-process pipeline based on Mediator pattern to drive command ingestion and route commands.

```
public class MyMicroserviceController : Controller
{
    public MyMicroserviceController(IMediator mediator,
                                    IMyMicroserviceQueries microserviceQueries)
    {
        // ...
    }
}
```

```
[Route("new")]
[HttpPost]
public async Task<IActionResult> ExecuteBusinessOperation([FromBody]RunOpCommand
                                                          runOperationCommand)
{
    var commandResult = await _mediator.SendAsync(runOperationCommand);


    return commandResult ? (IActionResult)Ok() : (IActionResult)BadRequest();
}
```

AGENDA

.NET CORE & ASP.NET CORE

ASP.NET CORE & DDD PATTERNS

BEYOND ASP.NET CORE

Beyond ASP.NET Core

Some references

MediatR. In-process messaging with no dependencies

https://github.com/jbogard/MediatR

RabbitMQ. Message Broker

https://www.rabbitmq.com/

Autofac. Inversion of Control Container for .NET Core

https://autofac.org/

Awesome .NET Core.

https://github.com/thangchung/awesome-dotnet-core

Microsoft Learn

https://docs.microsoft.com/en-us/learn/

RESUMEN

Recordemos

Domain-Driven Design

ASP.NET Core

REFERENCIAS

Para profundizar

https://docs.microsoft.com/en-us/dotnet/architecture/

PREGRADO

Ingeniería de Software

Escuela de Ingeniería de Sistemas y Computación | Facultad de Ingeniería

UPC
Universidad Peruana
de Ciencias Aplicadas

Prolongación Primavera 2390,
Monterrico, Santiago de Surco
Lima 33 - Perú
T 511 313 3333
https://www.upc.edu.pe

exígete, innova