

# Working with Repositories

---



**Steve Smith**  
Force Multiplier for  
Dev Teams  
[@ardalis](https://twitter.com/ardalis) [ardalis.com](http://ardalis.com)



**Julie Lerman**  
Software coach,  
DDD Champion  
[@julielerman](https://twitter.com/julielerman) [thedatafarm.com](http://thedatafarm.com)

## Overview



**Define repositories**

**Tips for designing repositories**

**Benefits of repositories**

**Pros and cons of interfaces and generic repos**

**Specification pattern to aid repositories**

**Repository implementations in our app**

# Introducing Repositories

---

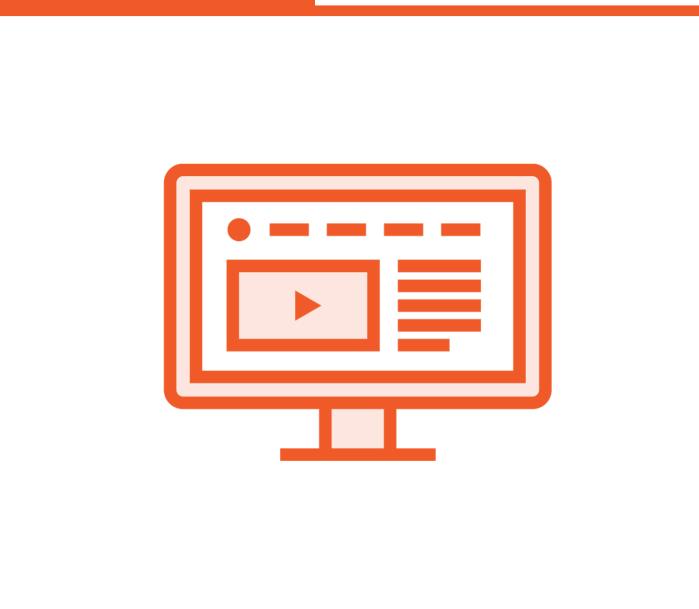


# Repository

Palaeoclimate archives: Core repository of AWI  
Hannes Grobe/AWI, Creative Commons Attribution 3.0

“Considering repositories had a huge impact on how I thought about software design.”

**Julie Lerman**

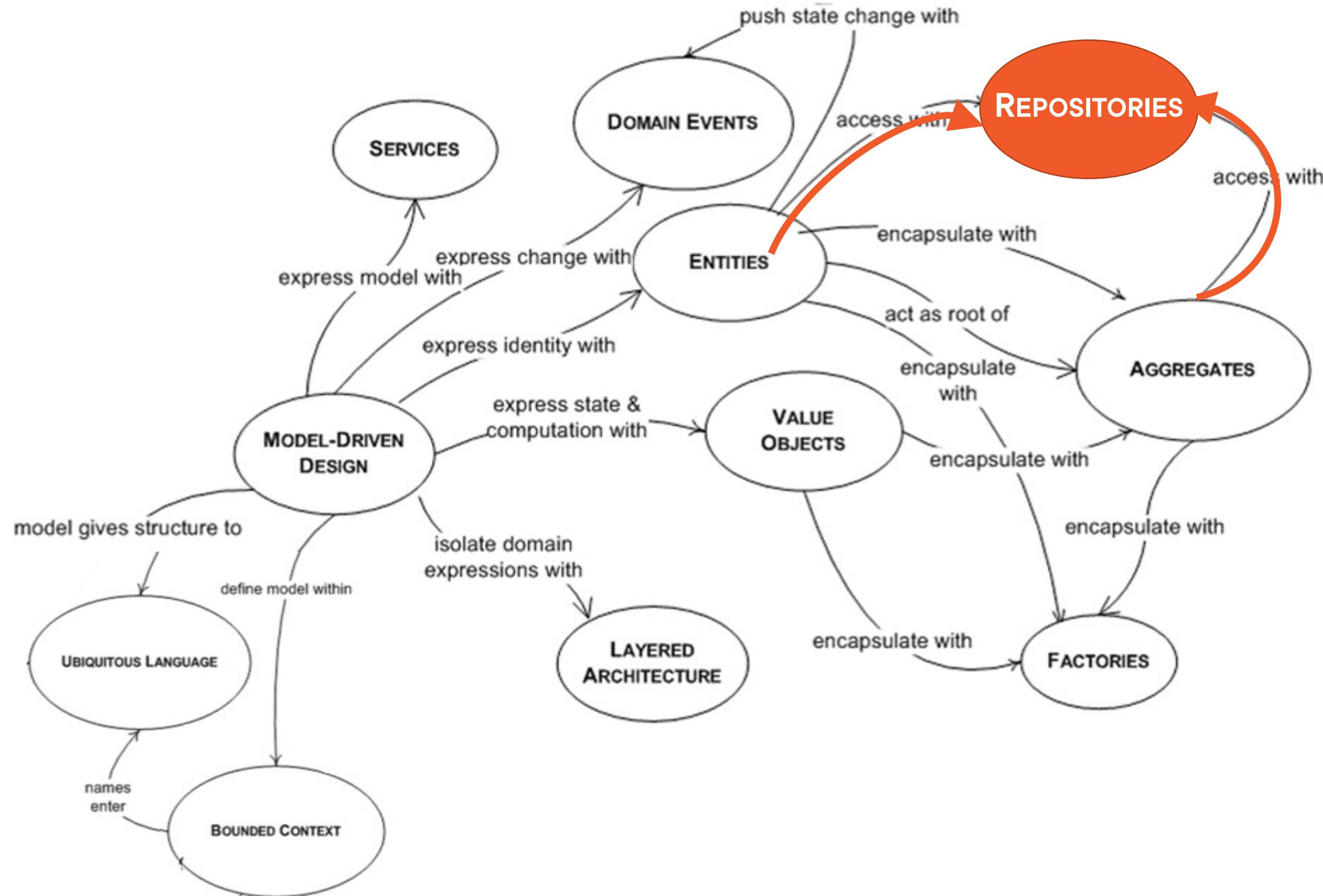


# **Repository Pattern**

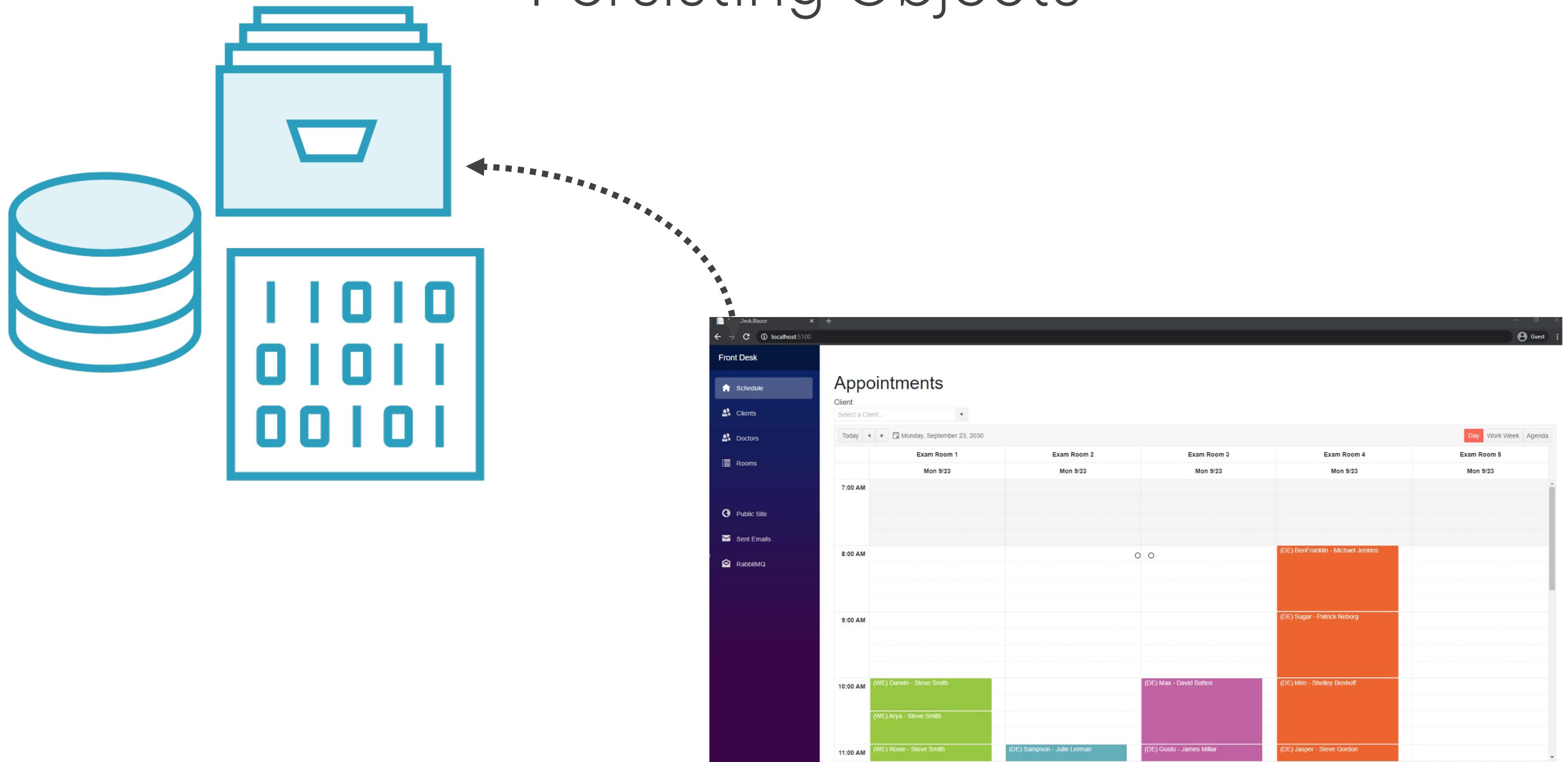
## **C# Design Patterns: Data Access Patterns**

**Filip Ekberg**

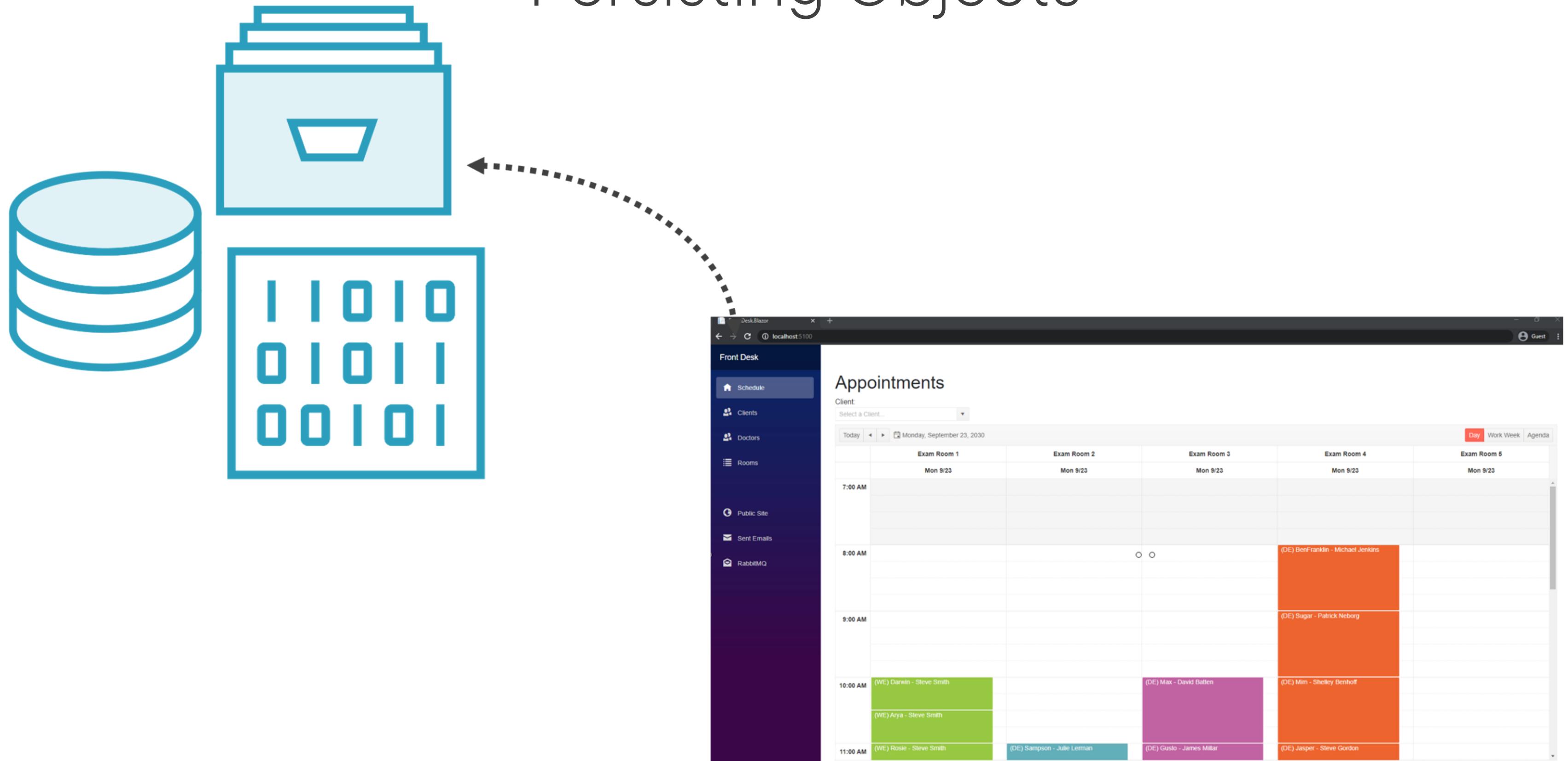
# Repositories in the DDD Mind Map



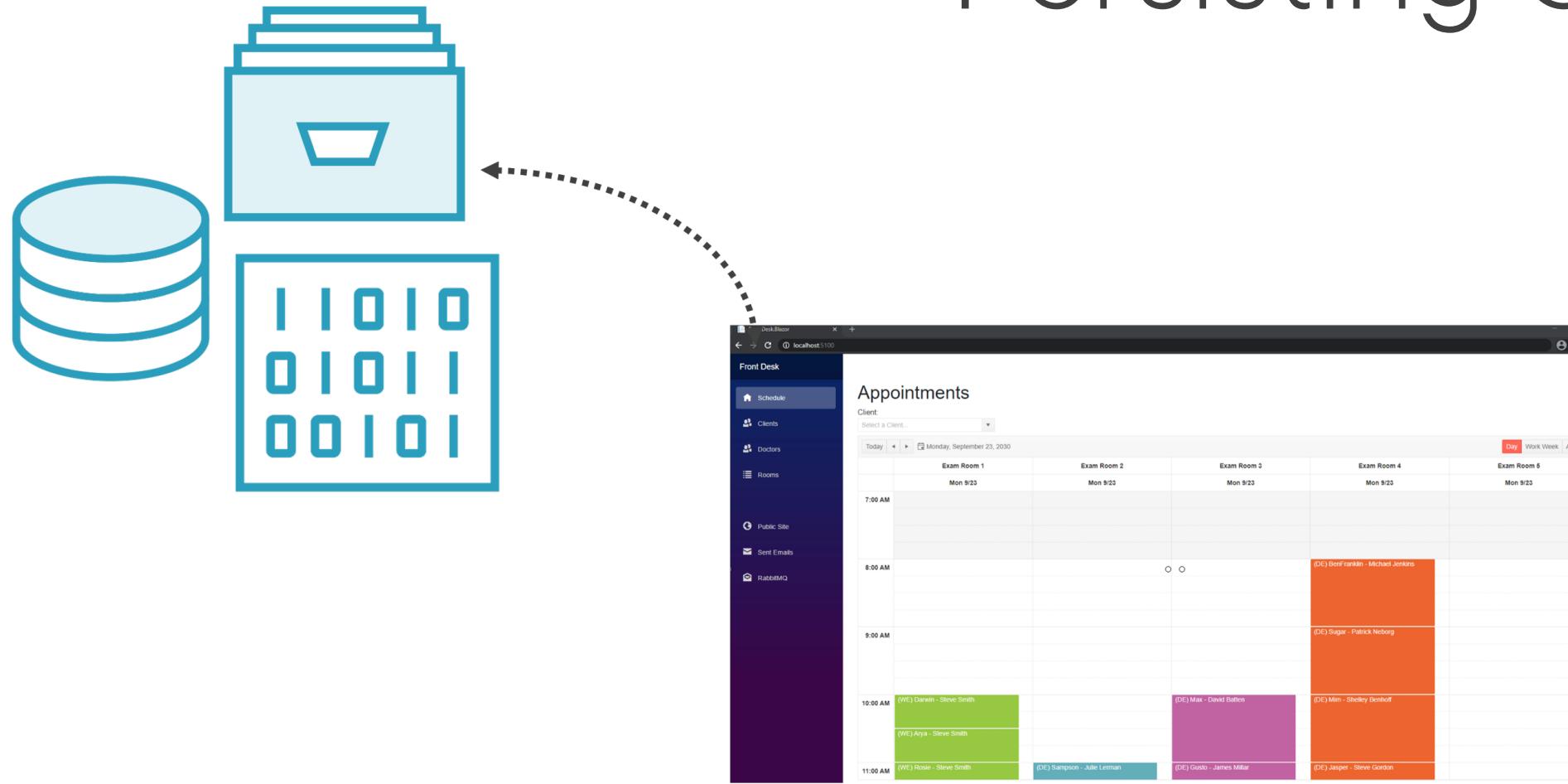
# Persisting Objects



# Persisting Objects

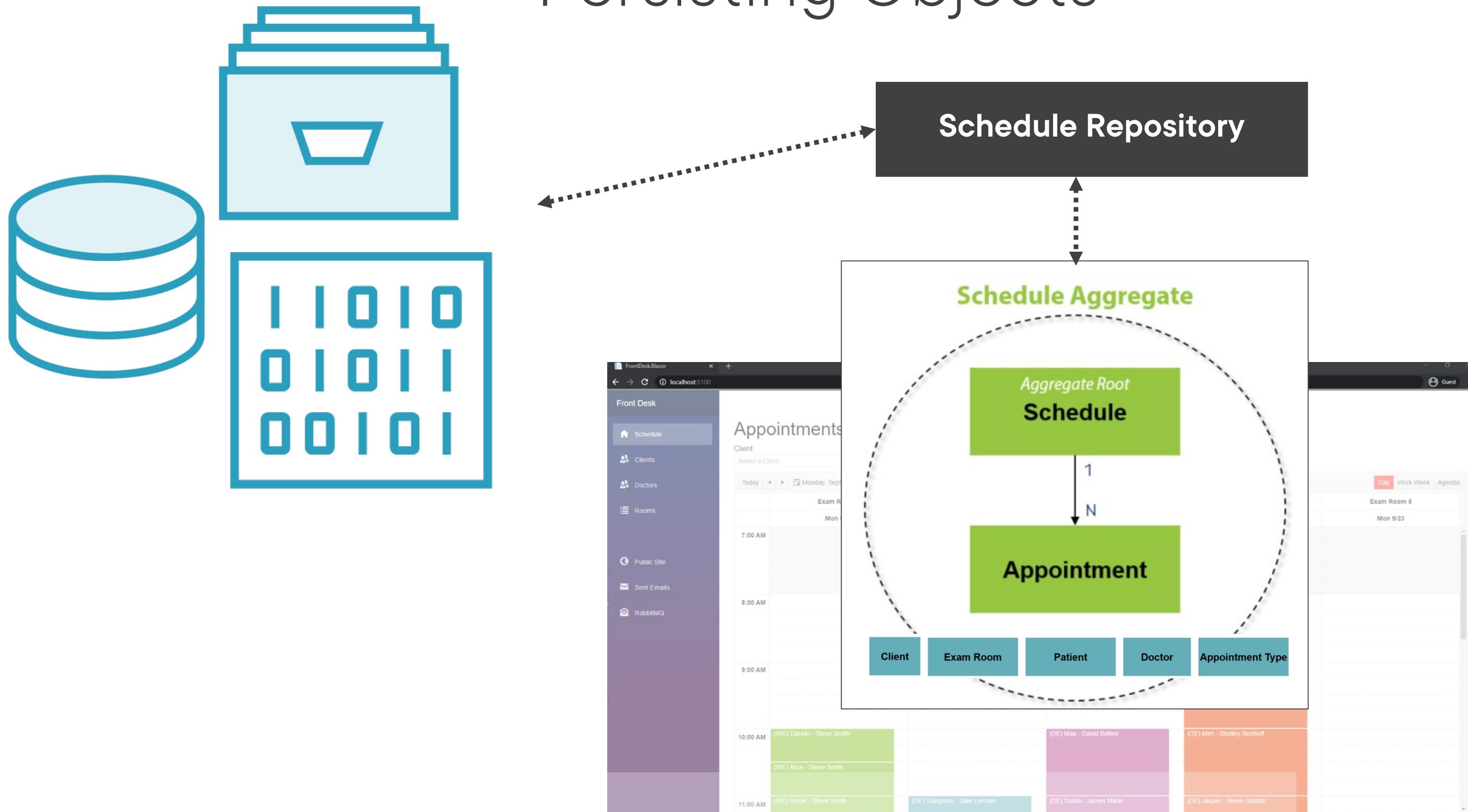


# Persisting Objects



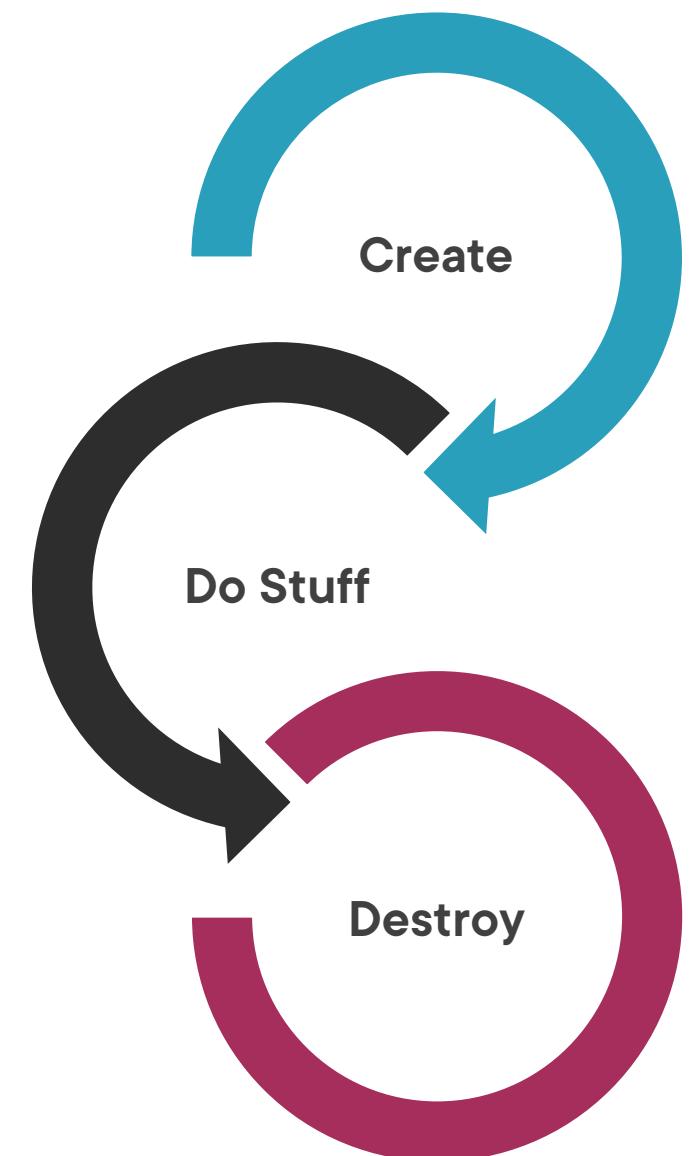
Random data access code  
in your system makes it  
difficult to maintain the  
integrity of your models

# Persisting Objects

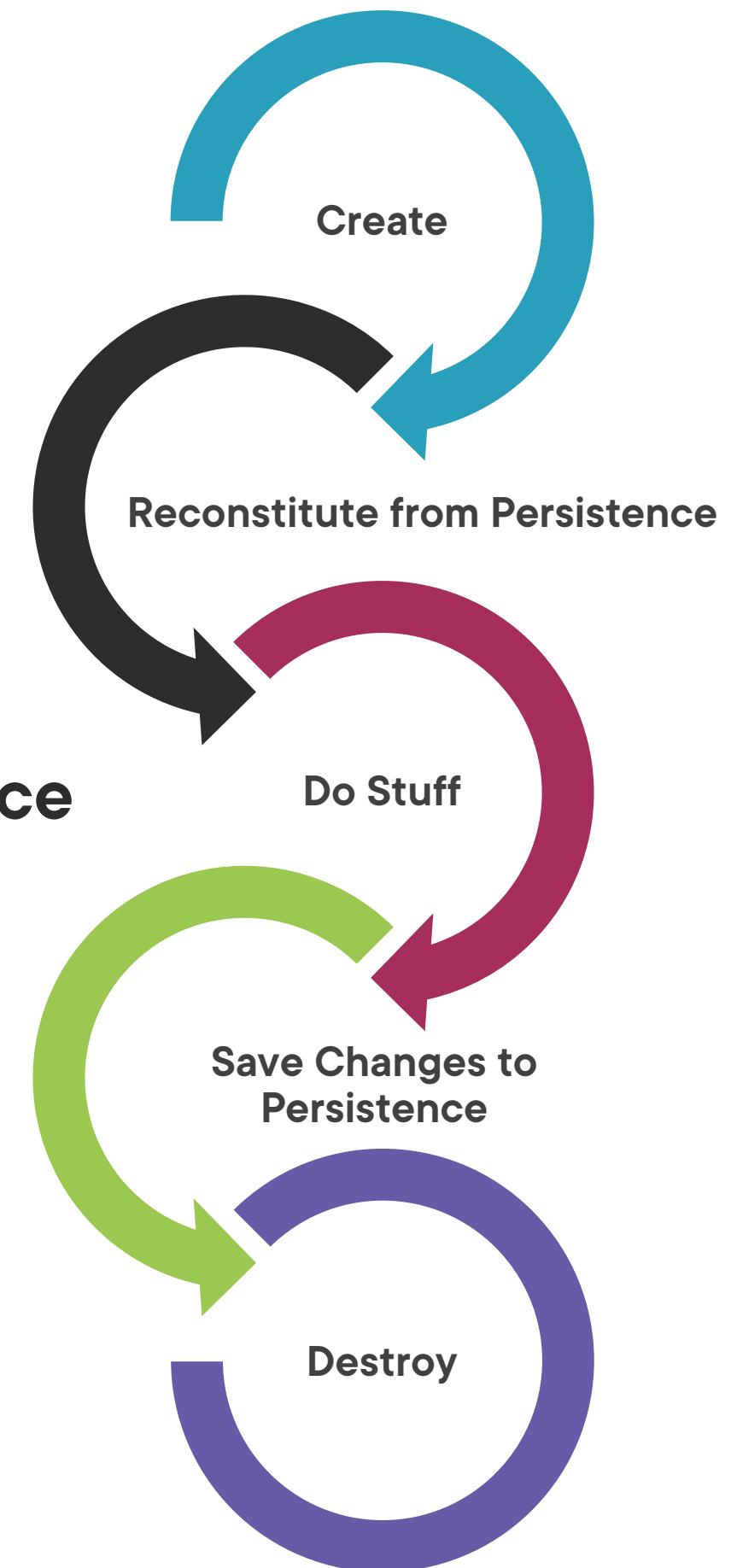


# Object Life Cycles

No Persistence

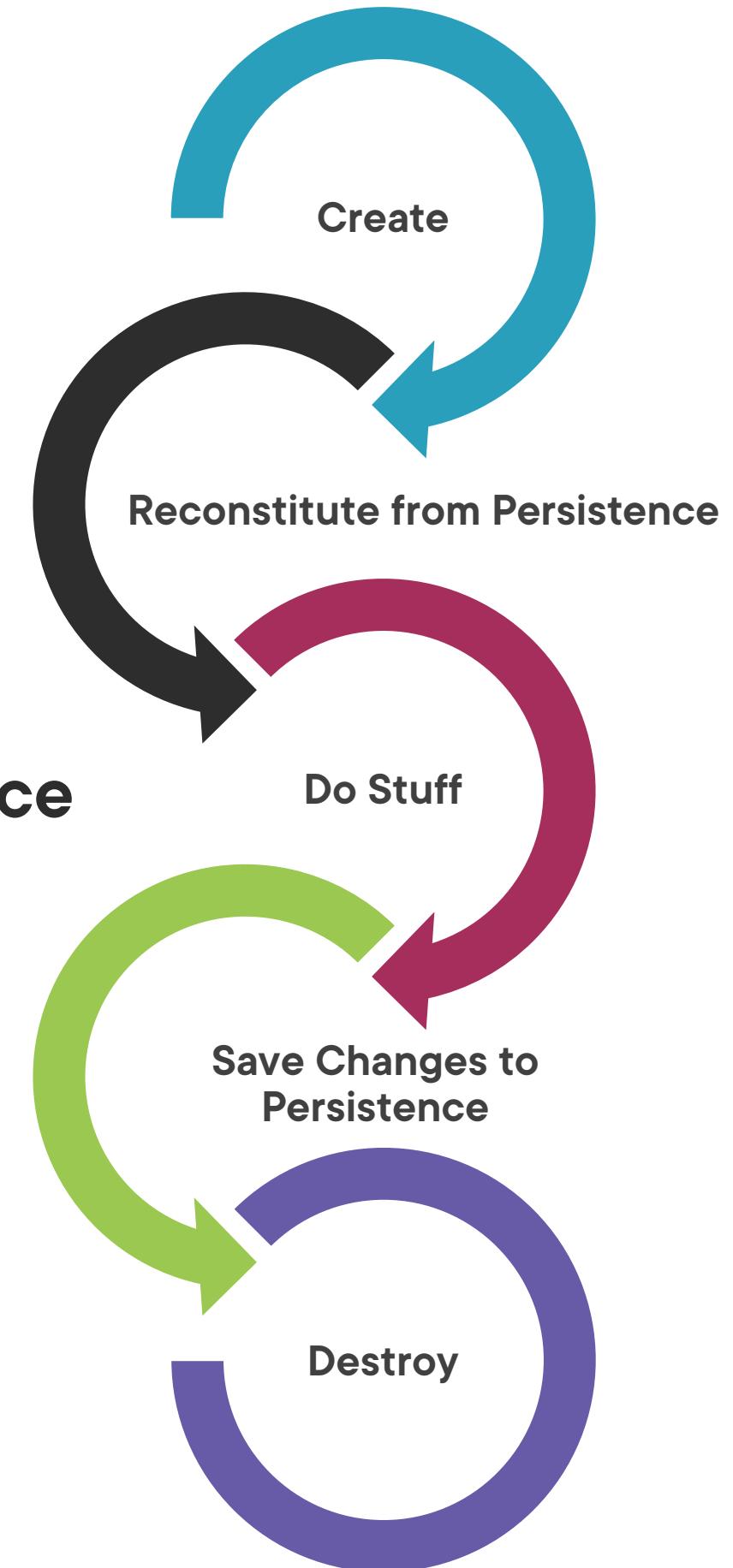


With Persistence



**Use a repository to manage the life cycle of persisted objects.**

## With Persistence



# Persistence Ignorance

**Business objects have no logic related to how data is stored and retrieved**

“A repository represents all objects of a certain type as a conceptual set... like a collection with more elaborate querying capability.”

**Eric Evans**  
Domain-Driven Design

# Repository Benefits

---



- Provides common abstraction for persistence**
- Promotes separation of concerns**
- Communicates design decisions**
- Enables testability**
- Improved maintainability**

# Repository Tips

---

Think of it as an  
in-memory collection



Implement a  
known, common  
access Interface

```
public interface IRepository<T>
{
    T GetById(int id);
    void Add(T entity);
    void Remove(T entity);
    void Update(T entity);
    IEnumerable<T> List();
}
```

Include methods to  
add and remove

```
public void Insert(TEntity entity)
{
    _dbSet.Add(entity);
    _context.SaveChanges();
}

public void Delete(int id)
{
    var entityToDelete=_dbSet.Find(id);
    _dbSet.Remove(entityToDelete);
    _context.SaveChanges();
}
```

## Custom Query Implementation using EF Core

### EfScheduleRepository.cs

```
public Schedule GetScheduleForDateWithAppointments(int clinicId,  
    DateTimeOffset date)  
{  
    var endDate = date.AddDays(1);  
  
    var schedule = _dbContext.Set<Schedule>()  
        .Include(s => s.Appointments.Where( a =>  
            a.TimeRange.Start > date &&  
            a.TimeRange.End < endDate))  
  
        .FirstOrDefault(schedule =>  
            schedule.ClinicId == clinicId);  
  
    return schedule;  
}
```

Get a Client with their  
Patients

## EfClientRepository.cs

```
public Client GetClientByIdWithPatients(int clientId)
{
    var client = _dbContext.Set<Client>()
        .Include(c => c.Patients)
        .FirstOrDefault(client => client.Id == clientId);

    return client;
}
```

# General Repository Tips

**Use repositories for  
aggregate roots only**

**Client focuses on model,  
repository on persistence**

# Avoiding Repository Blunders

---

**Client code can be ignorant of  
repository implementation**

**...but developers cannot**

# Problems Caused by Repository Logic



**N+1  
Query Errors**



**Inappropriate use of  
eager or lazy loading**



**Fetching more data  
than required**

```
var clients=_context.Clients.ToList();

foreach (var client in clients)
{
    _context.Patients.Where(p=>p.ClientId==client.Id)
        .ToList();
}
```

## N+1 Query Errors

```
select Clients.* from Clients
select Patients.* from Patients where ClientId=1
select Patients.* from Patients where ClientId=2
select Patients.* from Patients where ClientId=3
select Patients.* from Patients where ClientId=4
select Patients.* from Patients where ClientId=5
select Patients.* from Patients where ClientId=6
select Patients.* from Patients where ClientId=7
select Patients.* from Patients where ClientId=8
select Patients.* from Patients where ClientId=9
select Patients.* from Patients where ClientId=10
```

# Problems Caused by Repository Logic



**N+1  
Query Errors**



**Inappropriate use of  
eager or lazy loading**



**Fetching more data  
than required**

# Database Profiling Can Surface Many Problems

**Database IDE  
profilers**

**Code-based  
profiling or logging**

**3<sup>rd</sup> Party Profilers**

# Addressing the Debates About Using Repositories

---

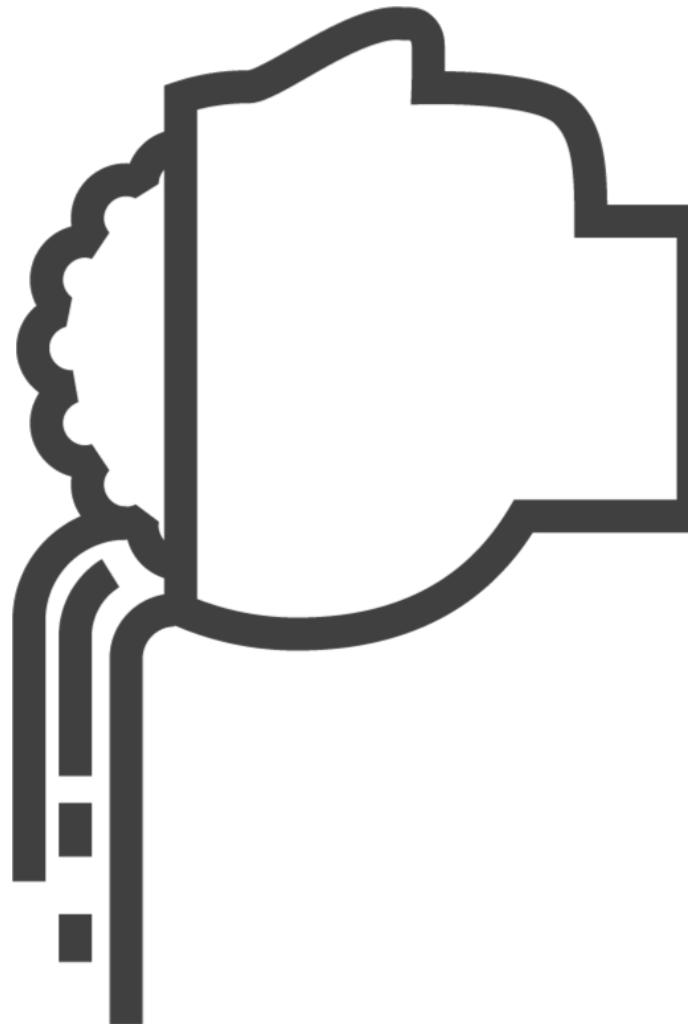
There are two kinds  
of design patterns:  
the ones people complain about  
and the ones nobody uses.

**NEVER**  
use a repository with EF Core!

**ALWAYS**  
use a repository with EF Core!



**Sharing our  
knowledge...**

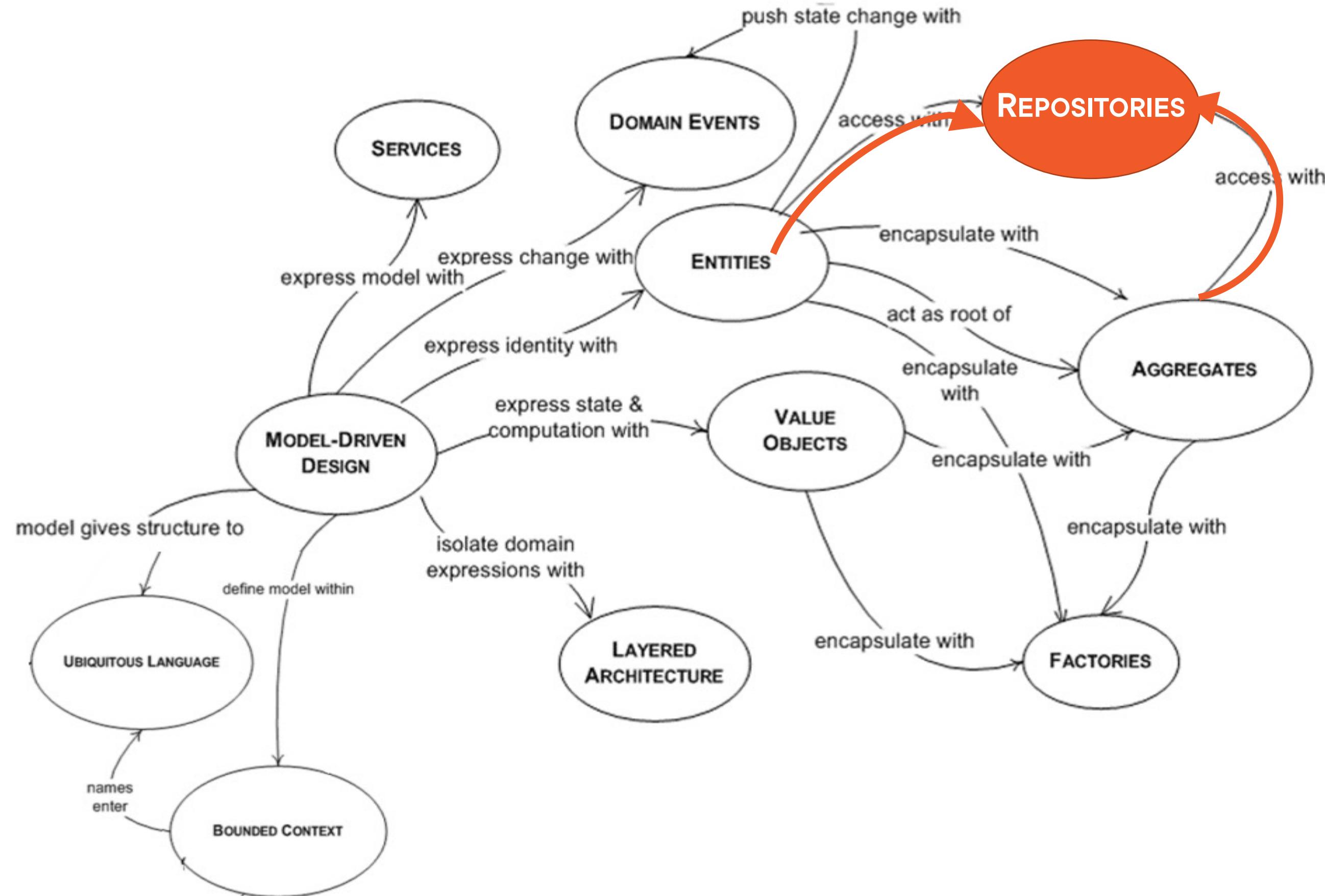


**...so you can make  
educated decisions**

# Repository

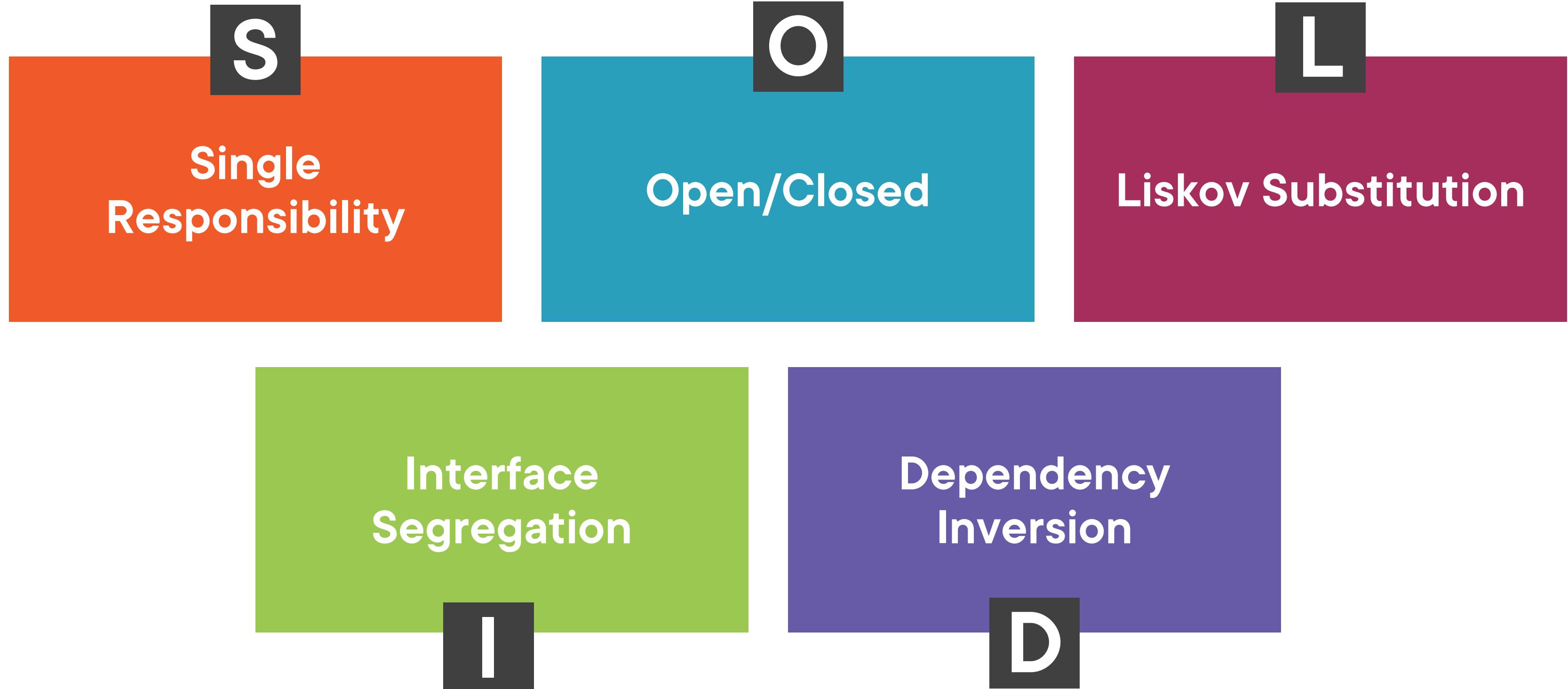
**An abstraction your domain model uses to define what persistence needs it has**

# Repositories in the DDD Mind Map



A domain model should be persistence ignorant as well as ignorant of implementation details.

# SOLID Principles



Source: SOLID Principles for C# Developers (Pluralsight course), Steve Smith

# SOLID and DDD

Dependency  
Inversion

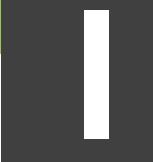


We can define an abstraction in the domain model

Implement that abstraction in another project that depends on the domain model

# SOLID and DDD

## Interface Segregation



**Clients should not be forced to depend on methods they don't use.**

**Prefer small, cohesive interfaces to large, “fat” ones.**

# Façade Pattern

**Using a class to contain a complicated class or API and only expose the methods needed by your program.**

# Abstracting persistence in our domain model

A persistence abstraction (a.k.a. a *Repository*)  
Abstraction defines “what” is needed  
Implementations define “how” it’s done  
EF Core is easily used by implementation  
classes

DDD prevents coupling domain  
problems with persistence  
problems.

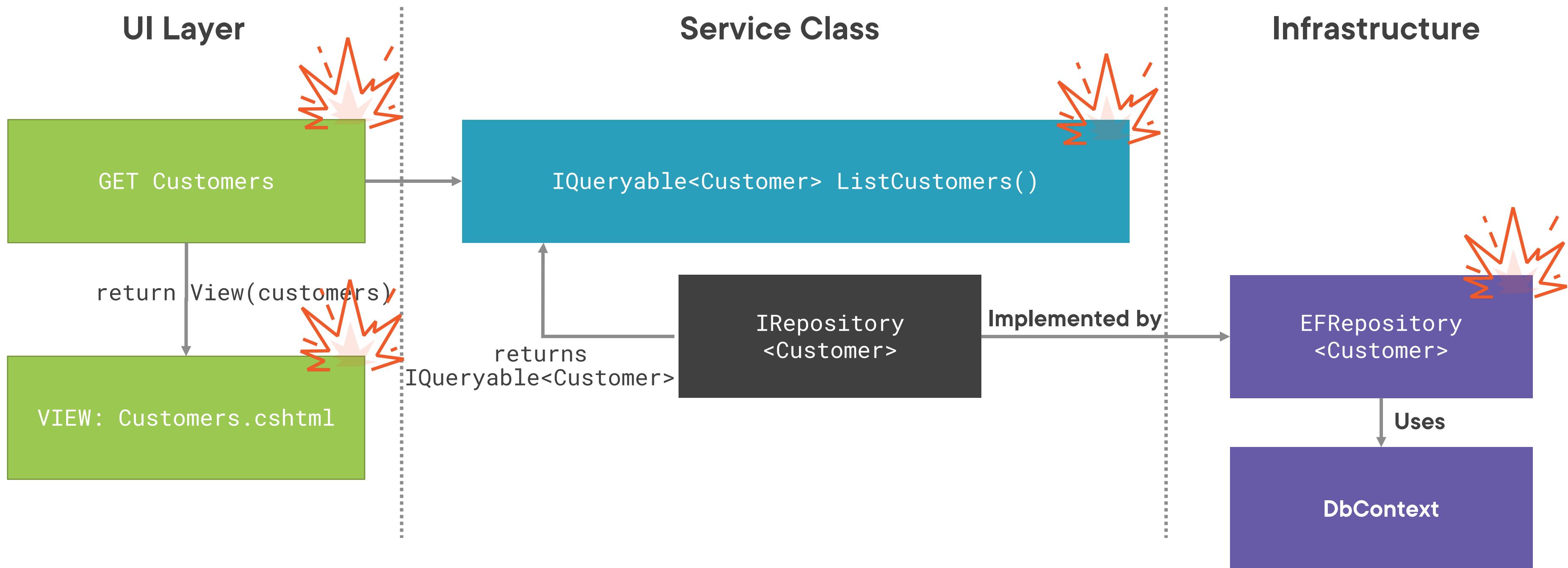
# Returning IQueryable: Pros and Cons

---



Should repositories  
return `IQueryable`?

# Where is Query Logic Defined?



## Using IQueryable Expression

# Returning IQueryable from Repository

## The Good

### Flexibility

**Can build query from multiple locations**

**Minimal Repository code required**

**Restrict data returned to just what is needed**

**Reuse small set of Repository methods**

## The Bad and The Ugly

**Query logic spread out everywhere**

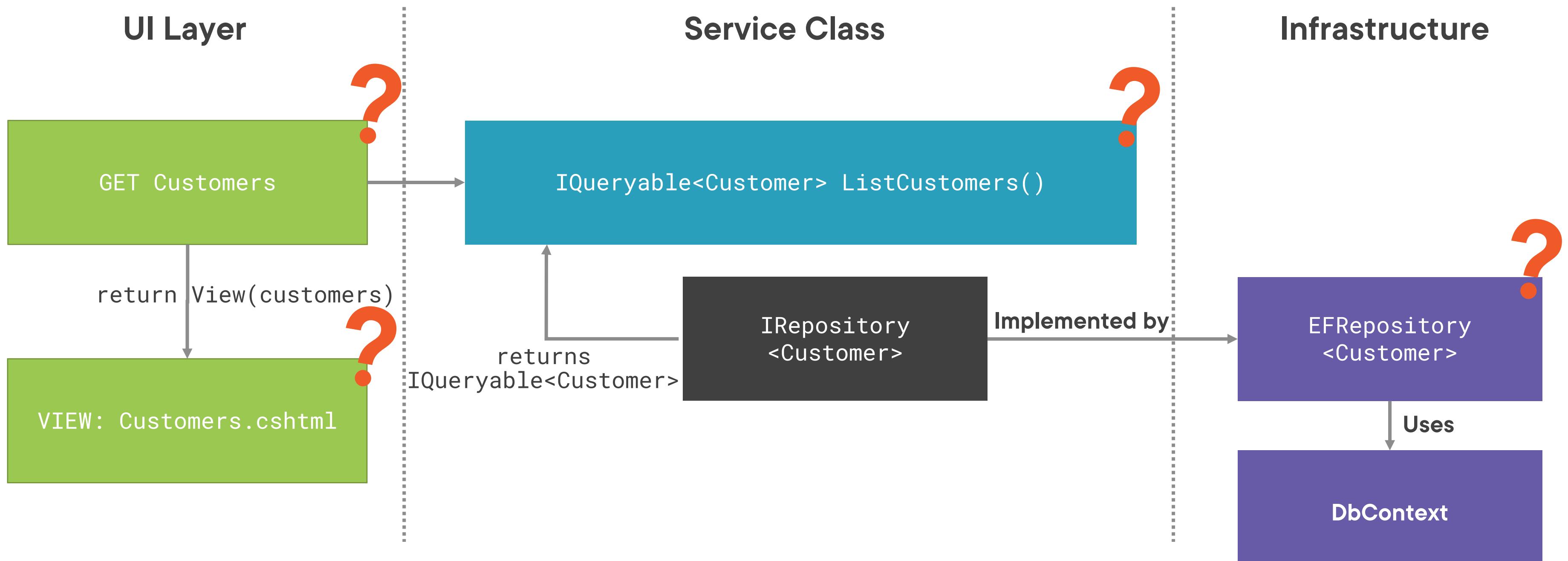
**Violating Single Responsibility Principle**

**Violating Separation of Concerns**

**Confusion about when the query actually executes**

**Code compiles, but blows up when executed**

# When is the Query Executed?



## Using IQueryable Expression

Produces SQL query from  
IQueryable expression and  
executes it

# Returning IQueryable from Repository

## The Good

### Flexibility

Can build query from multiple locations

Minimal Repository code required

Restrict data returned to just what is needed

Reuse small set of Repository methods

## The Bad and The Ugly

Query logic spread out everywhere

Violating Single Responsibility Principle

Violating Separation of Concerns

Confusion about when the query actually executes

Code compiles, but blows up when executed

No encapsulation

# Accept Arbitrary Predicates

(instead of returning IQueryable from Repository List methods)

## ICustomerRepository.cs

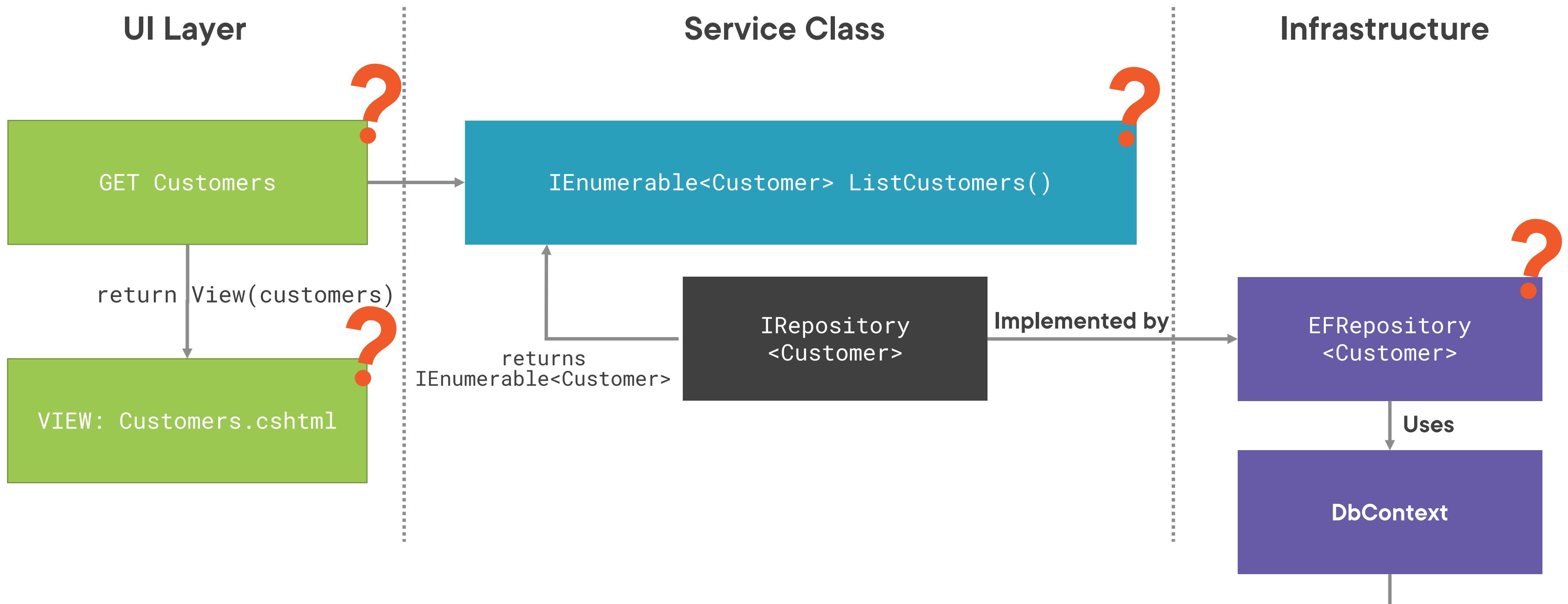
```
public interface ICustomerRepository
{
    IEnumerable<Customer> List(Expression<Func<Customer, bool>> predicate);
}

public IEnumerable<Customer> List(Expression<Func<Customer, bool>> predicate)
{
    return _db.Customers.Where(predicate);
}
```

# Predicate

**Expression used in the search condition of a query's where clause**

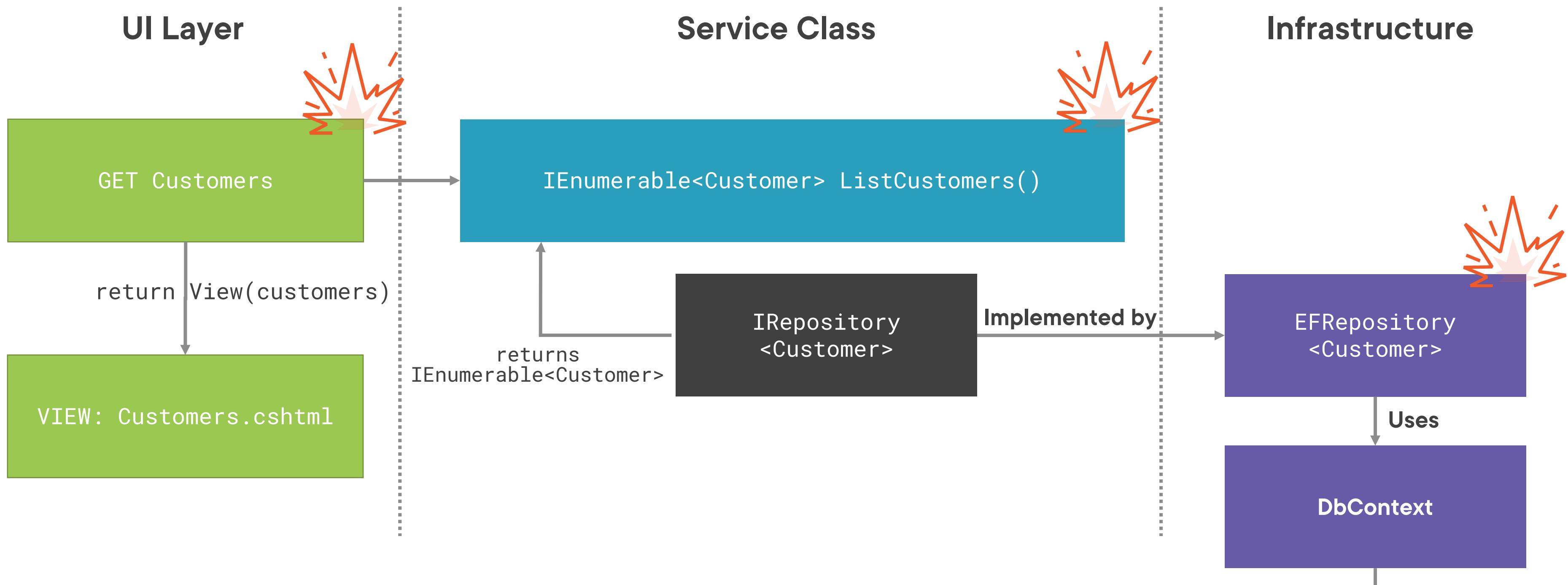
# When is the Query Executed?



## Using Predicate

Produces SQL query  
from a predicate and  
executes it

# Where is the Query Defined?



## Using Predicate

# Passing Predicates to the Repository

## The Good

### Flexibility

~~Can build query from multiple locations~~

Minimal Repository code required

Restrict data returned to just what is needed

Reuse small set of Repository methods

## The Bad and The Ugly

Query logic spread out everywhere

Violating Single Responsibility Principle

Violating Separation of Concerns

~~Confusion about when the query actually executes~~

Code compiles, but blows up when executed

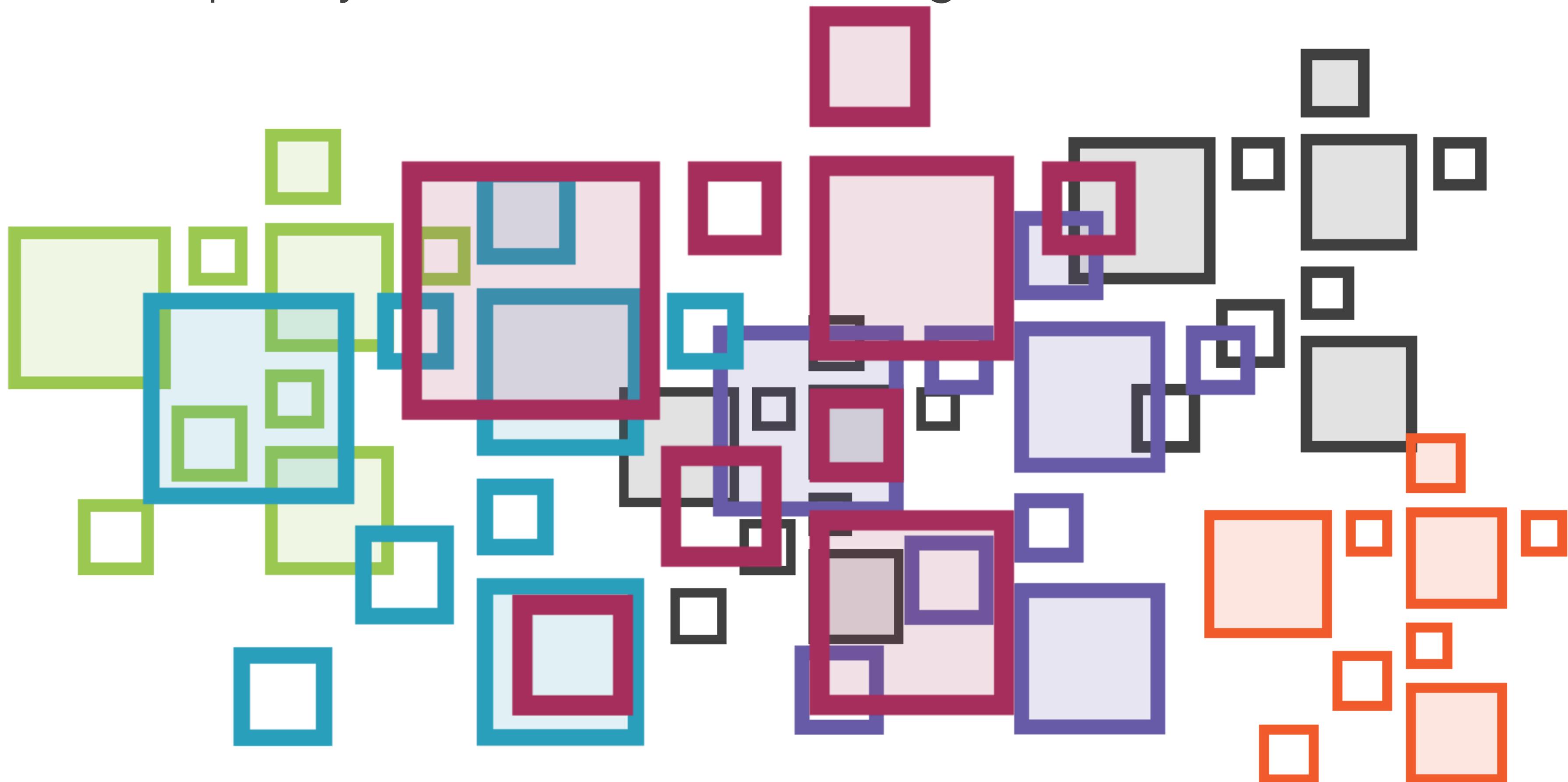
No encapsulation

# One Common Solution: Custom Query Methods!

```
public interface ICustomerReadRepository
{
    Customer GetById(int id);
    List<Customer> List();

    // custom queries
    List<Customer> ListCustomersByState(string state);
    List<Customer> ListCustomersBySales(decimal minSales);
    List<Customer> ListCustomersWithOrders();
    List<Customer> ListCustomersWithAddresses();
    List<Customer> ListCustomersWithOrdersAndAddresses();
    List<Customer> ListCustomersByStateWithOrders(string state);
    List<Customer> ListCustomersByLastName(string lastName);
    List<Customer> ListCustomersByGeo(int latitude, int longitude, int radiusMiles);
    List<Customer> ListCustomersByShoeSize(string size);
    List<Customer> ListCustomersByFavoriteNetflixShow(string title);
    // and more get added all the time
}
```

# Help! My Queries are Getting Out of Control!



# Considering Generic Repositories and Interfaces

---

# Generic Repository Benefits

**Promote code-reuse**

**Generic constraint can  
protect aggregates**

# Generic Repository Trade-Offs

**Consistent  
persistence  
implementation,  
but possible  
unused  
methods**



**Individually  
crafted classes  
with a variety  
of bespoke  
methods**

Trust your judgement and  
choose what makes sense for  
your application

# IRepository May Lead to Unused Methods

## Interface for Any Repository

```
public interface IRepository<T>
{
    T GetById(int id);
    void Add(T entity);
    void Remove(T entity);
    void Update(T entity);
    IEnumerable<T> List();
}
```

## Implementing IRepository

```
class ScheduleRepo : IRepository<Schedule>
{
    public Schedule GetById(int id)
    { ...some logic... }

    public void Add(Schedule entity)
    { ...some logic... }

    public void Remove(Schedule entity)
    { ... Do nothing! ... }

    public void Update(Schedule entity)
    { ...some logic... }

    public void IEnumerable<Schedule> List
    {}

}
```

## A Targeted IScheduleRepository with Relevant Methods

### IScheduleRepository.cs

```
public interface IScheduleRepository
{
    Schedule GetScheduleForDateWithAppointments
        (int clinicId, DateTime date);
    void Update(Schedule schedule);
}
```

# Generic Repositories for Aggregate Roots

```
public class Root: IEntity
{
    public int Id ...
}
```

```
public class RootRepository : IRepository<Root>
{
    public IEnumerable<Root> List()...
    public Root GetById(int id)...
    public void Insert (Root entity) ...
    public void Update (Root entity) ...
    public void Delete (Root etity) ...
}
```

# Generic Repositories for CRUD Work

```
public class Repository< TEntity >
    : IRepository< TEntity >
{
    private readonly CrudContext _context;
    private readonly DbSet< TEntity > _dbSet;

    public Repository(CrudContext context)...
    public IEnumerable< TEntity > List()...
    public Root GetById(int id)...
    public void Insert ( TEntity entity ) ...
    public void Update ( TEntity entity ) ...
    public void Delete ( TEntity entity ) ...
}
```

```
var repo=new Repository< Patient >();
repo.Insert(new Patient());
```

Constraining  
repositories to  
root with  
markers,  
prevents direct  
access to non-  
root entities

```
public class SomeNonRoot : IEntity
{
    public int Id...
    ...
}
```



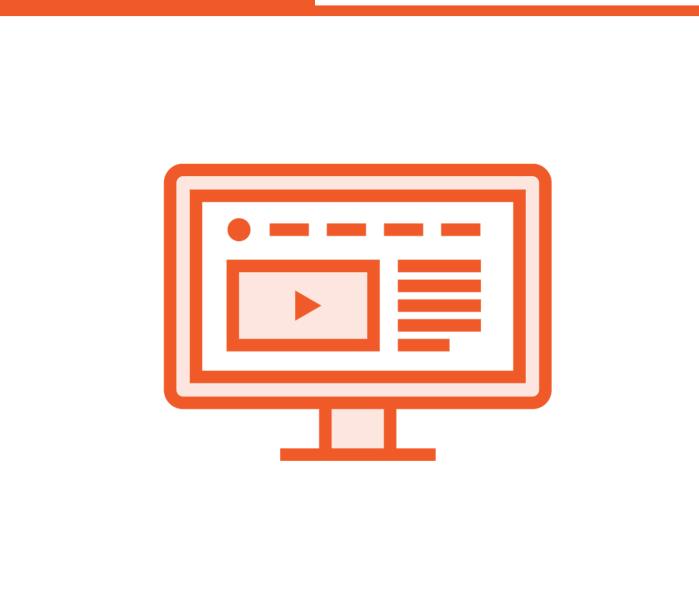
```
var repo=new Repository<SomeNonRoot>();
repo.GetById(1);
```



Marker  
interfaces  
can provide  
protection to  
your aggregates

```
public interface IAggregateRoot : IEntity {}  
  
public class Root : IAggregateRoot  
{  
    public int Id ...  
}  
  
public class Repository<TEntity>  
: IRepository<TEntity>  
where TEntity : class, IAggregateRoot
```

Repository abstractions can  
get large ...  
sometimes too large.



## Learn more about SOLID

**Solid Principles for C# Developers**

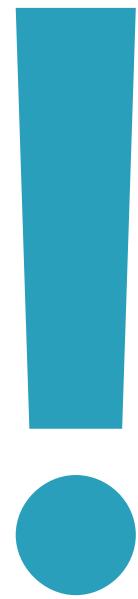
Steve Smith

[bit.ly/solid\\_smith\\_csharp](https://bit.ly/solid_smith_csharp)

# Command Query Responsibility Segregation (CQRS)



**Query Repositories**  
**focus on reading data**



**Command repositories**  
**focus on writing data**

# Some CQRS Benefits with Minimal Effort

**Query-focused repositories  
can benefit from caching**

**Command-focused  
repositories  
can benefit from queues**

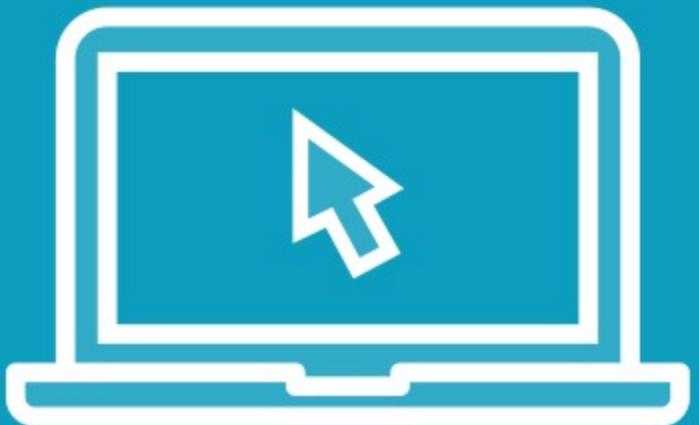
Too many read methods can  
interfere with caching logic.

Specification pattern can help!

# Exploring Repositories in our Application

---

Demo

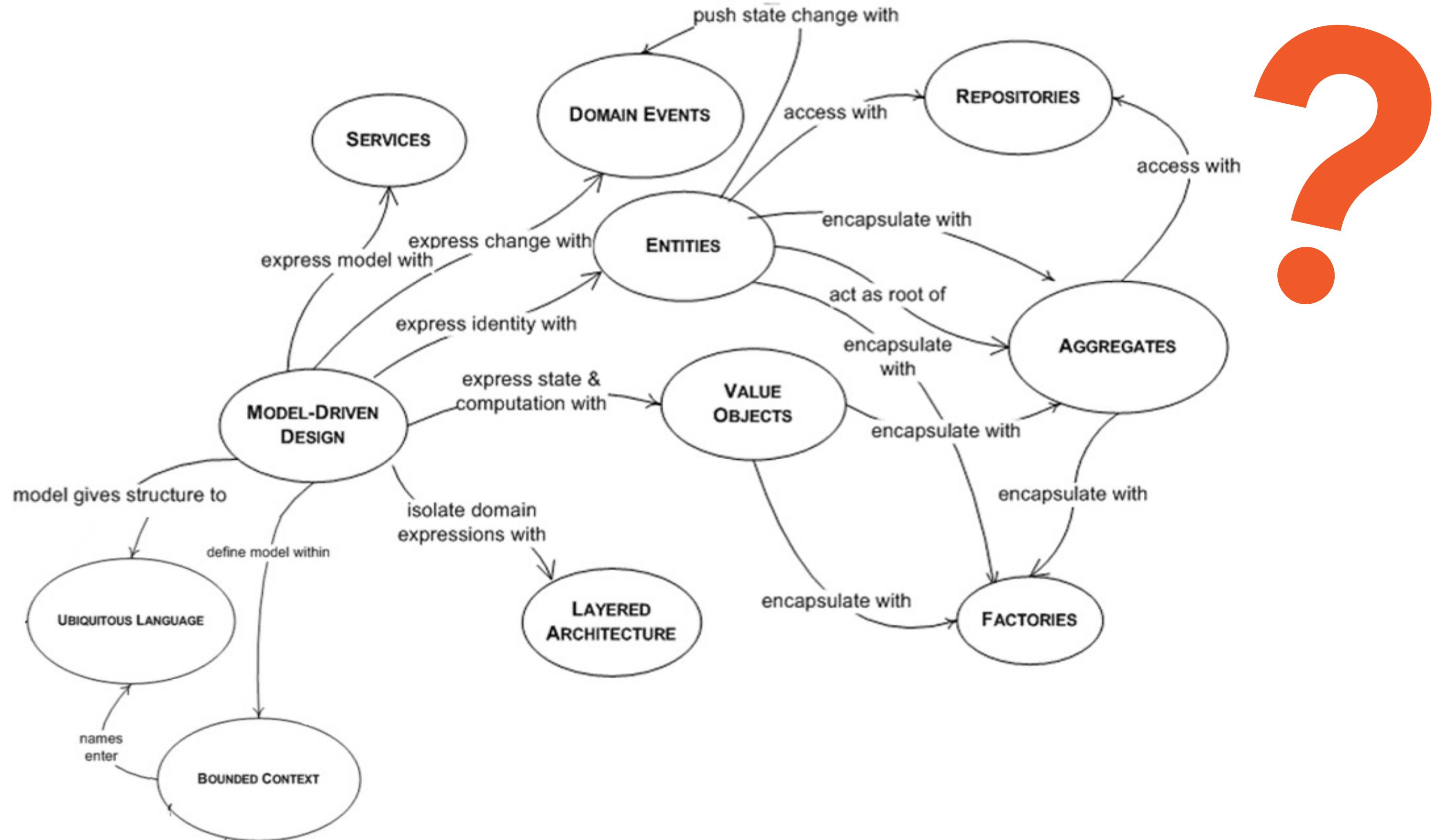


**Repositories in our application**

# Introducing the Specification Pattern

---

# Specifications in the DDD Mind Map



“Specifications mesh smoothly with Repositories, which are the building-block mechanisms for providing query access to domain objects and encapsulating the interface to the database.”

**Eric Evans, *Domain-Driven Design***

# Specifying the State of an Object

**Validation**

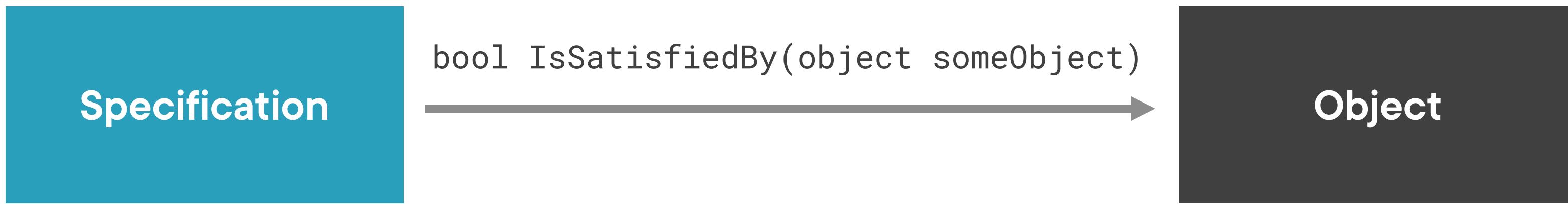
**Selection  
& Querying**

**Creation for a  
specific purpose**

“Create explicit predicate-like Value Objects for specialized purposes. A Specification is a predicate that determines if an object satisfies some criteria.”

**Eric Evans, *Domain-Driven Design***

# A Basic Specification

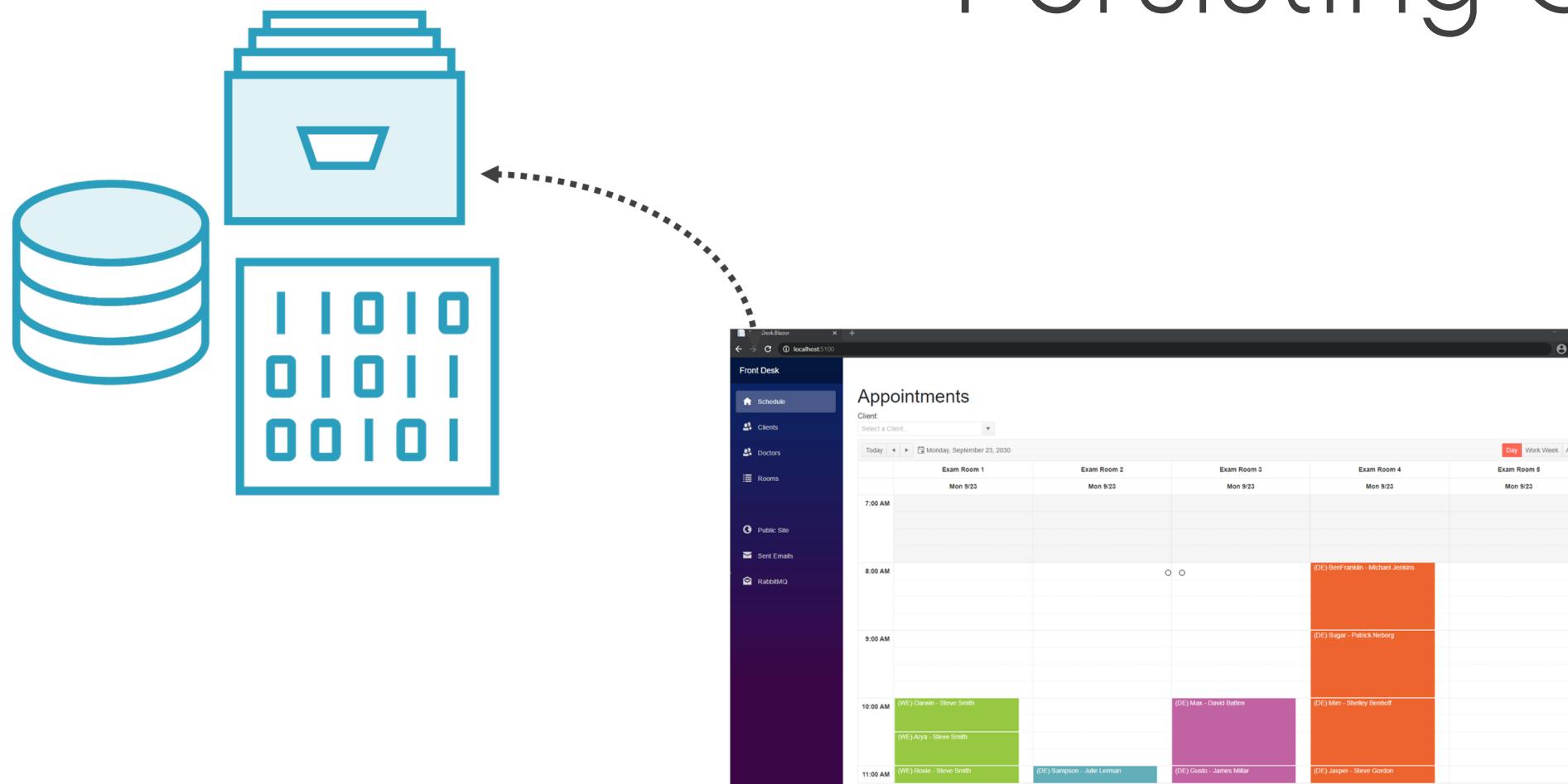


**Note:**  
Criteria evaluated in memory

# Combining Specifications with ORMs

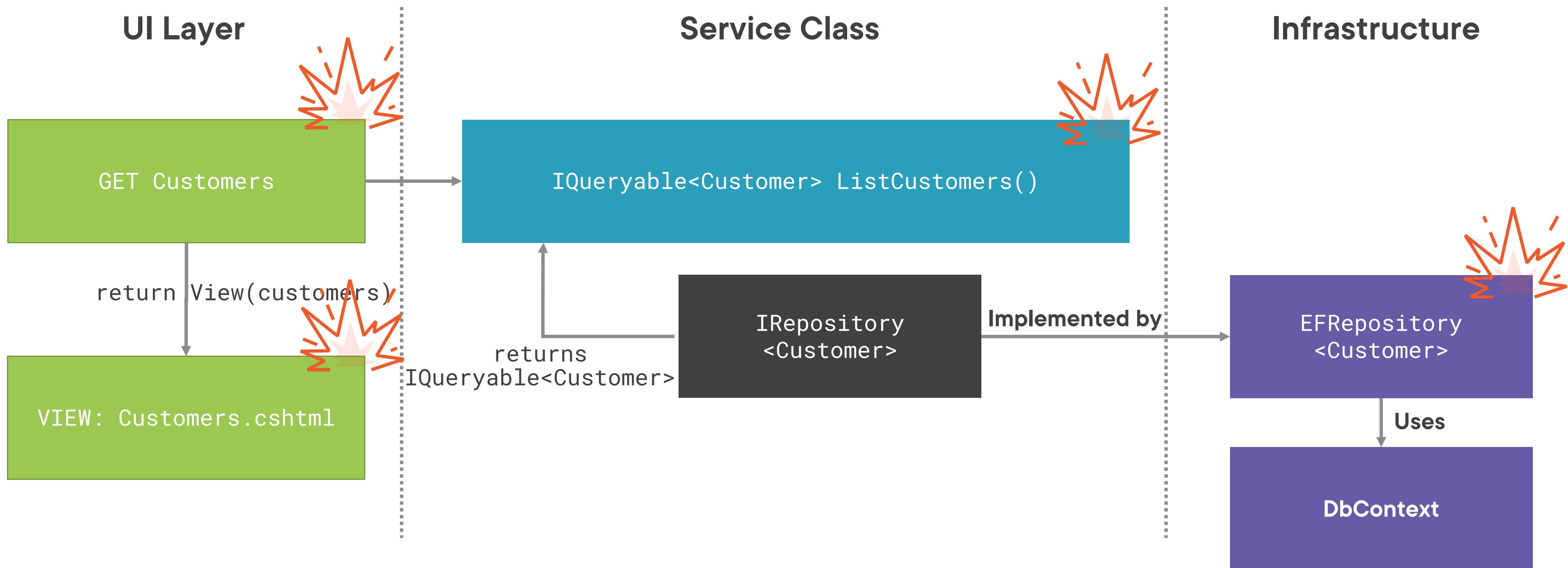


# Persisting Objects



Random data access code  
in your system makes it  
difficult to maintain the  
integrity of your models

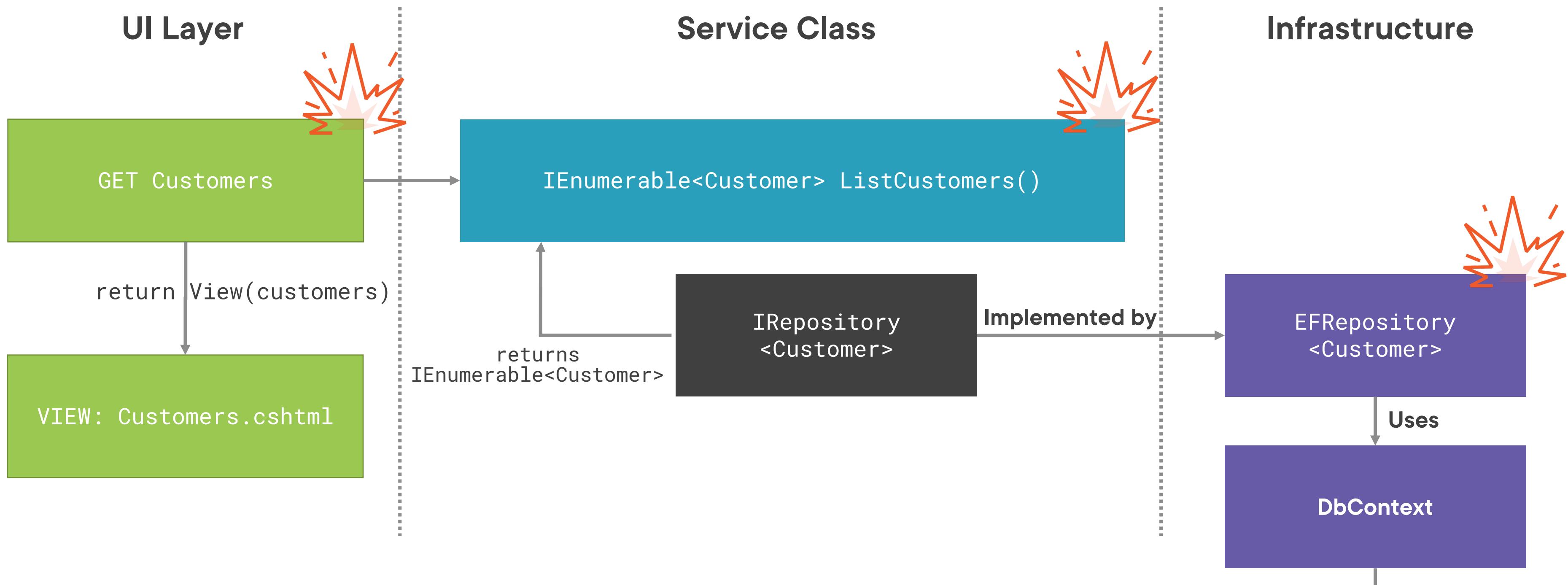
# Where is Query Logic Defined?



## Using IQueryable Expression

Produces SQL query  
from IQueryable  
expression and  
Executes it

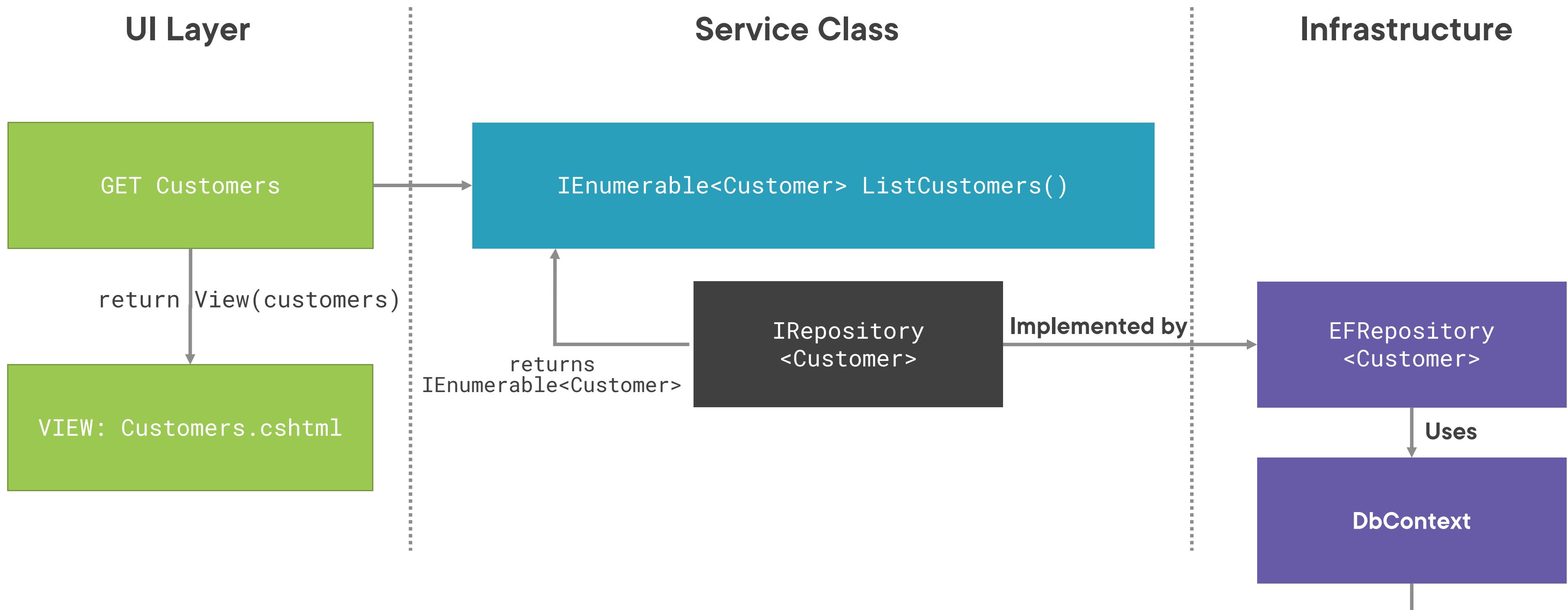
# Where is Query Logic Defined?



## Using Custom Predicate

Produces SQL Query  
from a predicate and  
executes it

# Where is Query Logic Defined?



## Using Specification

# Typed Repository Interfaces Provide Needed Query Methods

```
public interface ICustomerReadRepository
{
    Customer GetById(int id);
    List<Customer> List();

    // custom queries
    List<Customer> ListCustomersByState(string state);
    List<Customer> ListCustomersBySales(decimal minSales);
    List<Customer> ListCustomersWithOrders();
    List<Customer> ListCustomersWithAddresses();
    List<Customer> ListCustomersWithOrdersAndAddresses();
    List<Customer> ListCustomersByStateWithOrders(string state);
    List<Customer> ListCustomersByLastName(string lastName);
    List<Customer> ListCustomersByGeo(int latitude, int longitude, int radiusMiles);
    List<Customer> ListCustomersByShoeSize(string size);
    List<Customer> ListCustomersByFavoriteNetflixShow(string title);
    // and more get added all the time
}
```

# Typed Repository Interfaces Provide Needed Query Methods

```
public interface ICustomerReadRepository
{
    Customer GetById(int id);
    List<Customer> List();

    List<Customer> ListCustomersBySpecification(specification);
}
```

# Some More Specification Benefits

**Named Classes  
*via Ubiquitous Language***

**Reusable**

**Separate Persistence  
from Domain Model  
and UI**

**Keep Business Logic out  
of Persistence Layer  
and Database**

**Help Entities &  
Aggregates follow  
Single Responsibility  
Principle (SRP)**

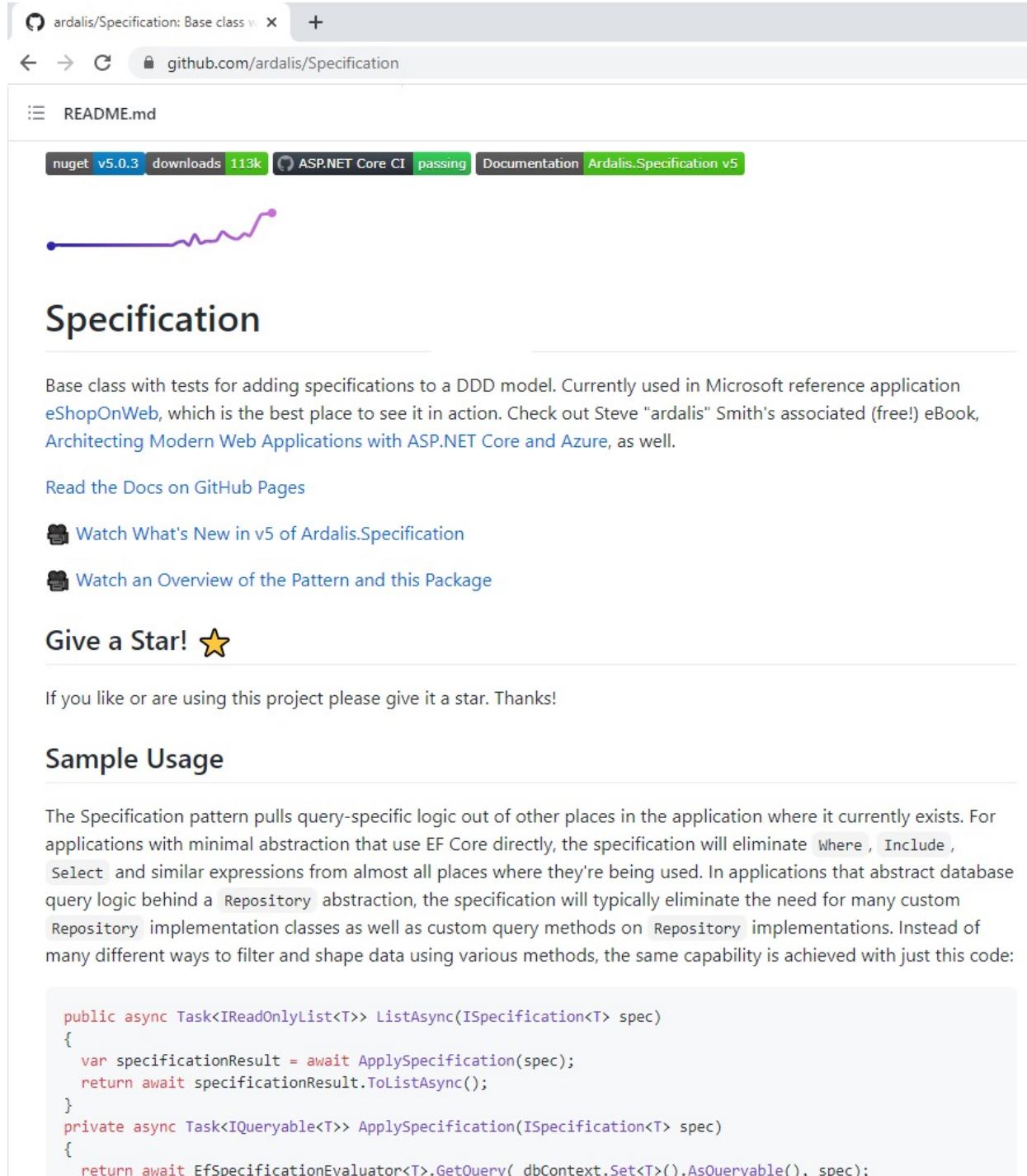
File Edit Selection View Go Run Terminal Help Specification.cs - Specification - Visual Studio Code

C# Specification.cs M X

ArdalisSpecification > src > Ardalism.Specification > C# Specification.cs > {} Ardalism.Specification

```
10 public abstract class Specification<T, TResult> : Specification<T>, ISpecification<T, TResult>
11 {
12     protected new virtual ISpecificationBuilder<T, TResult> Query { get; }
13
14     protected Specification()
15         : this(InMemorySpecificationEvaluator.Default)
16     {
17     }
18
19     protected Specification(IInMemorySpecificationEvaluator inMemorySpecificationEvaluator)
20         : base(inMemorySpecificationEvaluator)
21     {
22         this.Query = new SpecificationBuilder<T, TResult>(this);
23     }
24
25     public new virtual IEnumerable<TResult> Evaluate(IEnumerable<T> entities)
26     {
27         return Evaluator.Evaluate(entities, this);
28     }
29
30     public Expression<Func<T, TResult>>? Selector { get; internal set; }
31
32     public new Func<IEnumerable<TResult>, IEnumerable<TResult>>? PostProcessingAction { get; internal set; }
33 }
```

# Steve's Specification Pattern Base Class



The screenshot shows the GitHub project page for 'ardalis/Specification'. The page includes a navigation bar with links for README.md, nuget v5.0.3, downloads 113k, ASP.NET Core CI passing, Documentation, and Ardalism.Specification v5. Below the navigation, there's a purple line graph icon. The main content area has a section titled 'Specification' with a brief description of the base class and its use in Microsoft's eShopOnWeb application. It also includes links to 'Read the Docs on GitHub Pages', 'Watch What's New in v5 of Ardalism.Specification', and 'Watch an Overview of the Pattern and this Package'. A 'Give a Star!' button with a yellow star icon is present, along with a note about giving it a star if you're using the project. The 'Sample Usage' section explains the pattern and provides a code snippet for applying a specification to a queryable object.

```
public async Task<IReadOnlyList<T>> ListAsync(ISpecification<T> spec)
{
    var specificationResult = await ApplySpecification(spec);
    return await specificationResult.ToListAsync();
}
private async Task<IQueryable<T>> ApplySpecification(ISpecification<T> spec)
{
    return await EfSpecificationEvaluator<T>.GetQuery(_dbContext.Set<T>().AsQueryable(), spec);
}
```

## GitHub Project

[github.com/ardalis/Specification](https://github.com/ardalis/Specification)

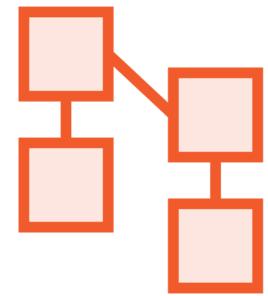
## NuGet package

[nuget.org/packages/Ardalis.Specification/](https://nuget.org/packages/Ardalis.Specification/)

# Implementing Specification Classes



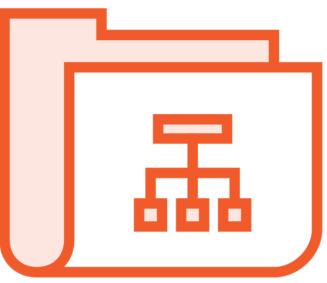
**You will need to write the rules of your specifications**



**The classes belong in your domain model**



**If only a few, organize in root Specifications folder**



**Or, along side your aggregates in their folders**

# Custom Specification Inheriting from Base

```
public class ScheduleIdWithAppointmentsSpec : Specification<Schedule>
{
    public ScheduleByIdWithAppointmentsSpec(Guid scheduleId)
    {
        Query
            .Where(schedule => schedule.Id == scheduleId)
            .Include(schedule => schedule.Appointments);
    }
}
```

# Examples of Applying Specifications in EF Core

```
dbContext.Customers.WithSpecification(specification).ToListAsync();
```

```
dbContext.Customers.WithSpecification(specification).FirstOrDefaultAsync();
```

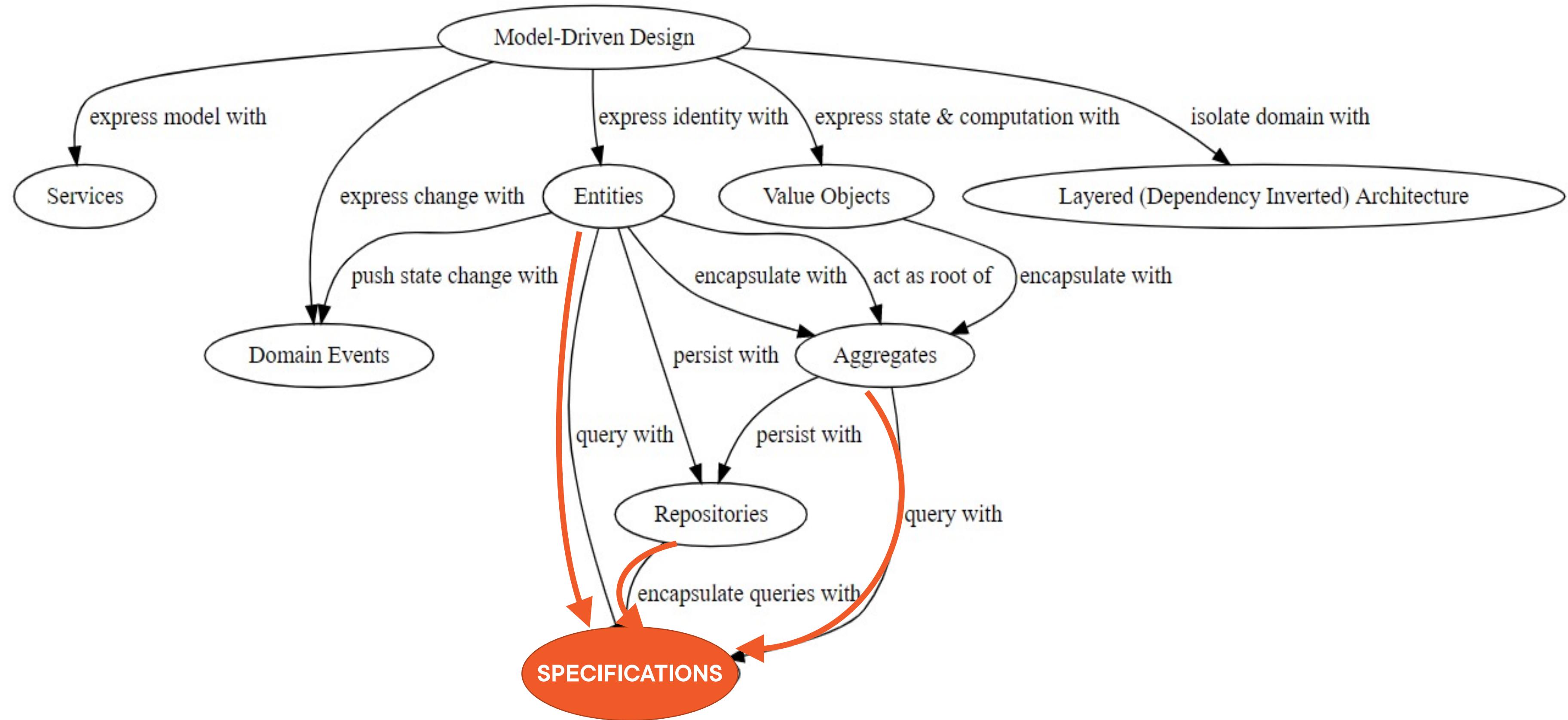
```
dbContext.Customers.WithSpecification(specification)  
    .Select("whatever your expression is").ToListAsync();
```

```
dbContext.Customers.WithSpecification(specification)  
    .UseWhateverExtensionsAvailableForIQueryable;
```

# Using Specifications in Your Code

```
var clientSpec = new ClientByIdIncludePatientsSpecification(appointment.ClientId);  
var client = await _clientRepository.GetBySpecAsync(clientSpec);
```

# Specifications in the DDD Mind Map



# Using Specifications with Repositories in Our App

---

Demo



**Using Specifications with Repositories  
in Our App**

# Module Review and Resources

---

# Key Terms from this Module

## Repository

**A class that encapsulates the data persistence for an aggregate root**

## Specification Pattern

**A method of encapsulating a business rule so that it can be passed to other methods which are responsible for applying it**

## Persistence Ignorance

**Objects are unaware of where their data comes from or goes to**

# Key Terms from this Module

**ACID**

**Atomic, Consistent, Isolated, and Durable**

**SOLID**

**A set of five software design patterns**

## Key Takeaways



**Repository pattern and the DDD mind map**

**Benefits of and tips for building repositories**

**Repository debates:**  
**Use them? Return IQueryables?**

**Specification pattern with repositories**

**Sample code is filled with great examples!**

Up Next:  
Adding in Domain Events  
and Anti-Corruption Layers

---

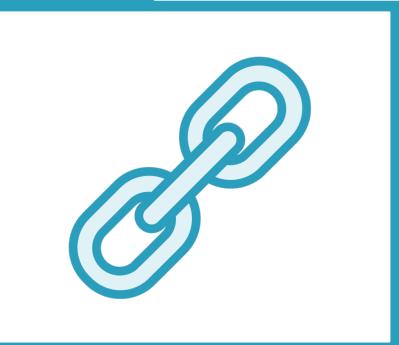
# Resources Referenced in This Module



On Pluralsight: SOLID Principles for C# developers -  
[bit.ly/solid\\_smith\\_csharp](https://bit.ly/solid_smith_csharp)



On Pluralsight: Entity Framework in the Enterprise –  
[bit.ly/PS-EFEnterprise](https://bit.ly/PS-EFEnterprise) (See “The Great Repository Debate” module)

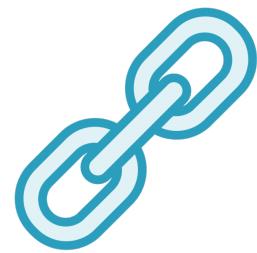


Specification Pattern Base Class [github.com/ardalis/Specification](https://github.com/ardalis/Specification)



On Pluralsight: C# Design Patterns: Façade by David Starr  
[app.pluralsight.com/library/courses/csharp-design-patterns-facade](https://app.pluralsight.com/library/courses/csharp-design-patterns-facade)

# Resources Referenced in This Module



## Avoid In-Memory Databases for Tests

[jimmybogard.com/avoid-in-memory-databases-for-tests/](http://jimmybogard.com/avoid-in-memory-databases-for-tests/)

# Working with Repositories

---



**Steve Smith**  
Force Multiplier for  
Dev Teams  
[@ardalis](https://twitter.com/ardalis) [ardalis.com](http://ardalis.com)



**Julie Lerman**  
Software coach,  
DDD Champion  
[@julielerman](https://twitter.com/julielerman) [thedatafarm.com](http://thedatafarm.com)