EF ADVANCED

AGENDA

- EfCore Advanced
  - Existing database
  - Transactions
  - Disconnected Entities
  - Development
- Structure EfCore project
  - DDD: Entity + Repository
  - Unit of Work
  - QueryObjects
  - Object Mappers

EXISTING DATABASE

- Reverse engineering a database through EF Core can be done with Scaffold tool
  - Visual Studio:

```
> Scaffold-DbContext 'Data Source=(localdb)\MSSQLLocalDB;Initial
Catalog=Chinook' Microsoft.EntityFrameworkCore.SqlServer
```

Note: Connection string needs to point to the actual database you want to scaffold

Speaker notes

1. Create project
2. Install Microsoft.EntityFrameworkCore.SqlServer and Microsoft.EntityFrameworkCore.Design
3. Run above command with a correct Connection string

.Net Core CLI:
```
$ dotnet ef dbcontext scaffold "Data Source=
(localdb)\MSSQLLocalDB;Initial Catalog=Chinook"
Microsoft.EntityFrameworkCore.SqlServer
```

SCAFFOLD OPTIONS

- Skip tables with `-Tables` / `--tables`
- Preserving names from database `-UseDatabaseNames` / `--use-database-names`
- Fluent API is used by default to change to annotations `-DataAnnotations` / `--data-annotations`
- More configuration to be found in documentations

WHEN SCAFFOLD DON'T WORK

- Columns types not supported with EF Core
- Inheritance
- Tables without primary key

AFTERWARDS

- The model can be changed afterwards (with migrations) -> So inheritance etc. can be created manually afterwards
- Model is created as partial classes - so you can add extra validation

PARTIAL CLASSES

```
[ModelMetadataType(typeof(UserValidation))]
public partial class User {
  public string Email { get; set; }
}

private class UserValidation {
  [EmailAddress]
  public string Email { get; set; }
}
```

TRANSACTIONS

- All changes to be saved with SaveChanges() are applied in a single transaction

```
public class AClass {
 public void AMethod() {
  using (var context = new MyDbContext()) {
   using (var transaction = context.Database.BeginTransaction()) {
     try {
       context.Books.Add(new Book { Title = "First Book" });
       context.SaveChanges();
       // .. Network call which Depends on First Book in DB and Second Book not
       context.Books.Add(new Book { Title = "Second Book" });
       context.SaveChanges();

       var books = context.Books.OrderBy(b => b.Title).ToList();
       // Commit transaction if all commands succeed, transaction will auto-rollback
       // when disposed if either commands fails
       transaction.Commit();
     }
     catch (Exception) { // TODO Handle failure
} } } } }
```

WHEN TO USE TRANSACTIONS

- Business logic gets complex
- Solutions:

1. ONE BIG METHODS WITH ALL THE LOGIC

```
public void OneMethodToRuleThemAll(MyContext context) {
    // Add
    // Update
    // Delete
    context.SaveChanges();
}
```

1. ONE BIG METHODS WITH ALL THE LOGIC

Problem: Obvious

```
public void OneMethodToRuleThemAll(MyContext context) {
    // Add
    // Update
    // Delete
    context.SaveChanges();
}
```

Speaker notes

Breaks SRP

Hard to reuse

2. SMALLER METHODS WHICH ARE CALLED FROM OVERREACHING METHOD

```
public void SaveAll() {
    SaveA();
    SaveB();
    SaveC();
    // Or call context.SaveChanges() instead
    // of each method
}

public void SaveA() {
    context.Add(new A() { ...});
    context.SaveChanges();
}
```

- Problem: If later parts relies on earlier parts being written orm we forget to call

2. SMALLER METHODS WHICH ARE CALLED FROM OVERREACHING METHOD

```
public void SaveAll() {
    SaveA();
    SaveB();
    SaveC();
    // Or call context.SaveChanges() instead
    // of each method
}

public void SaveA() {
    context.Add(new A() { ...});
    context.SaveChanges();
}
```

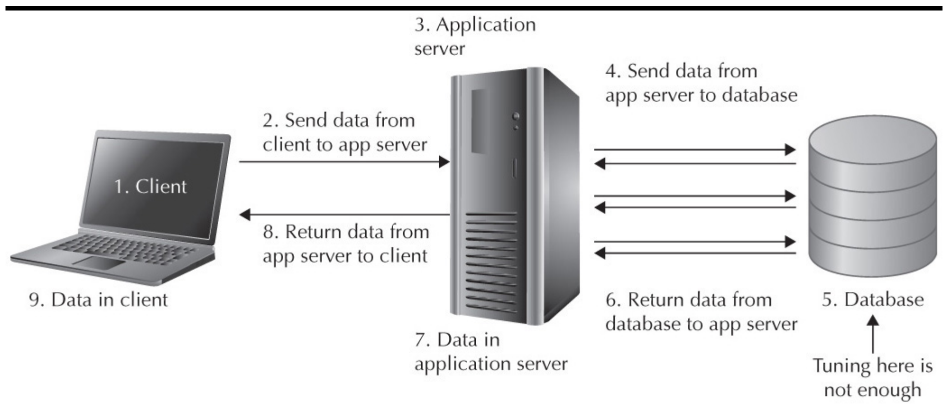
Problem: If later parts relies on earlier parts being written orm we forget to call SaveChanges

3. SMALLER METHODS AND USE TRANSACTION TO RUN THEM AS ONE

- Be aware that transactions can lock tables for writes (or reads) in some cases - so use with care

DISCONNECTED ENTITIES

- Scenario: Changes are made in a different database context instance



3. Application server
4. Send data from app server to database
2. Send data from client to app server
1. Client
8. Return data from app server to client
9. Data in client
6. Return data from database to app server
5. Database
7. Data in application server
Tuning here is not enough

UPDATE DISCONNECTED ENTITIES

- Determine if the entity exists in DB or not.
  - Auto-keys: use `context.Update(entity)` `// In Core 2.0+`
  - else: `context.Find(entity.Id) == null`
    - Use `context.Update(entity)`
    - Or `context.Add(newEntity)`
- Same for Graphs

DELETE DISCONNECTED ENTITIES

- Handling deletes
  - Harder since objects do not exists, so need check which do not exists in incoming
  - Can be handled with 'soft-deletes'

DEVELOPMENT

- From EF Core 2.1 the HasData method is added - part of OnModelCreating
- Migrations is created without connection to DB - have to specify ID manually.
- Data is removed if Primary key is changed

```
public class Context: DbContext {
 public void CreateData(ModelBuilder modelBuilder) {
   modelBuilder.Entity<Book>().HasData(new Book() { Title = "A title", Isbn = 1 });
   modelBuilder.Entity<PriceOffer>().HasData(new PriceOffer{NewPrice = 1.1f, Isbn = 2});
   // Not working !
   // modelBuilder.Entity<Book>().OwnsMany(b => b.Reviews).HasData(
   //      new Review() { Id = 1, BookIsbn = 1, Votername = "V1", NumStars = 1},
   //      new Review() { Id = 2, BookIsbn = 2, Votername = "V2", NumStars = 2}
   // );
   modelBuilder.Entity<Review>().HasData(
       new Review() { Id = 2, BookIsbn = 2, Votername = "V2", NumStars = 2});
   modelBuilder.Entity<Review>().HasData(
       new { Id = 1, BookIsbn = 1, Votername = "V1", NumStars = 1});
} }
```

OTHER SOLUTIONS

- `InsertData()`, `UpdateData()`, `DeleteData()` on `MigraionBuilder`
  - See Custom Migrations operations

LOGGING

- LogLevel.Information gives a list of all SQL commands generated by EF Core
- Setup log factory:

```
public class AClass {
  public void Setup() {
    var logs = new List<String>();
    var loggerFactory = context.GetService<ILoggerFactory>();
    loggerFactory.AddProvider(new MyLoggerProvider(logs, LogLevel.Information));
  }
}
```

PERFORMANCE

- EF Core alerts to possible suboptimal LINQ commands by logging a warning of type QueryClientEvaluationWarning.
  - Sometimes EF Core can not translate LINQ expression to SQL
- Configure EF Core to throw an exception
- Analyse SQL queries
  - E.g. Azure Data studio execution plan

```
Run current Query with Actual Plan
```

KEEP YOUR APPLICATION PERFORMING

1. Use SELECT loading to load only needed data
2. Use paging/filtering to reduce rows loaded into application from SQL Server
3. Lazy loading will affect performance
4. Use `AsNoTracking` when you load read-only data
5. Using async versions when possible
6. Structure code, so database access code is isolated

TESTING

- Create AddDbContext constructor for testing, which allows for all options to come from test methods

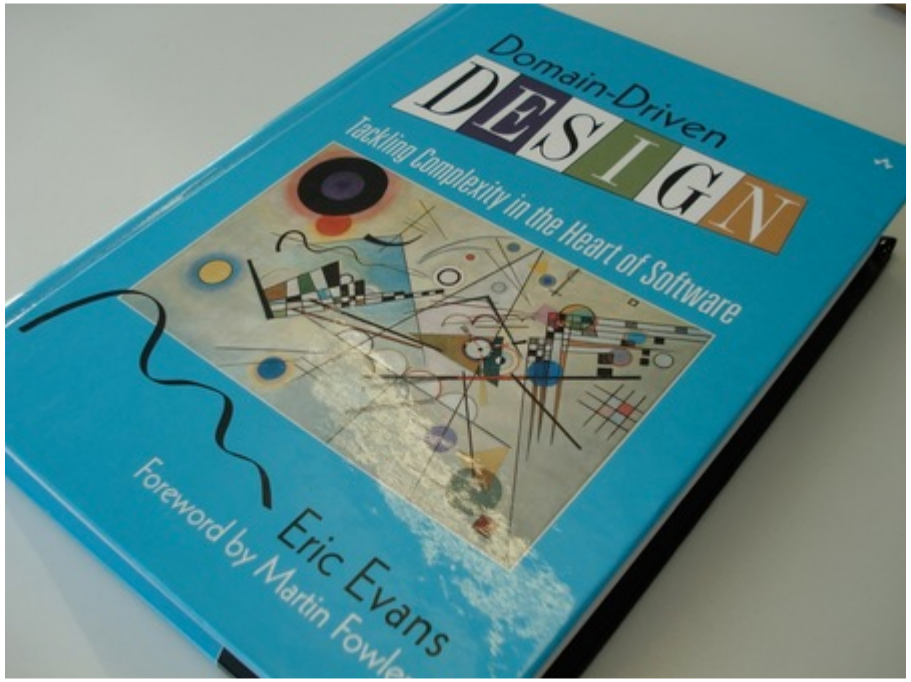
```
public class AppDbContext : DbContext {
  public AppDbContext() { }

  public AppDbContext(DbContextOptions<AppDbContext> options)
      : base(options) { }

  protected override void OnConfiguring(DbContextOptionsBuilde
  {
    if (!options.IsConfigured)
    {
      options.UseSqlServer("Data Source=....");
    }
  }
}
```

STRUCTURING EFCORE APPS

- Build on top of others work
- Faster development
- Less repetitive work
  - -> Fewer bugs
- Keep access to EfCore code in DAL
- Separation of Concerns
  - Each layer is responsible for one thing - mental model is easier
  - Having a isolated DAL is easier to test

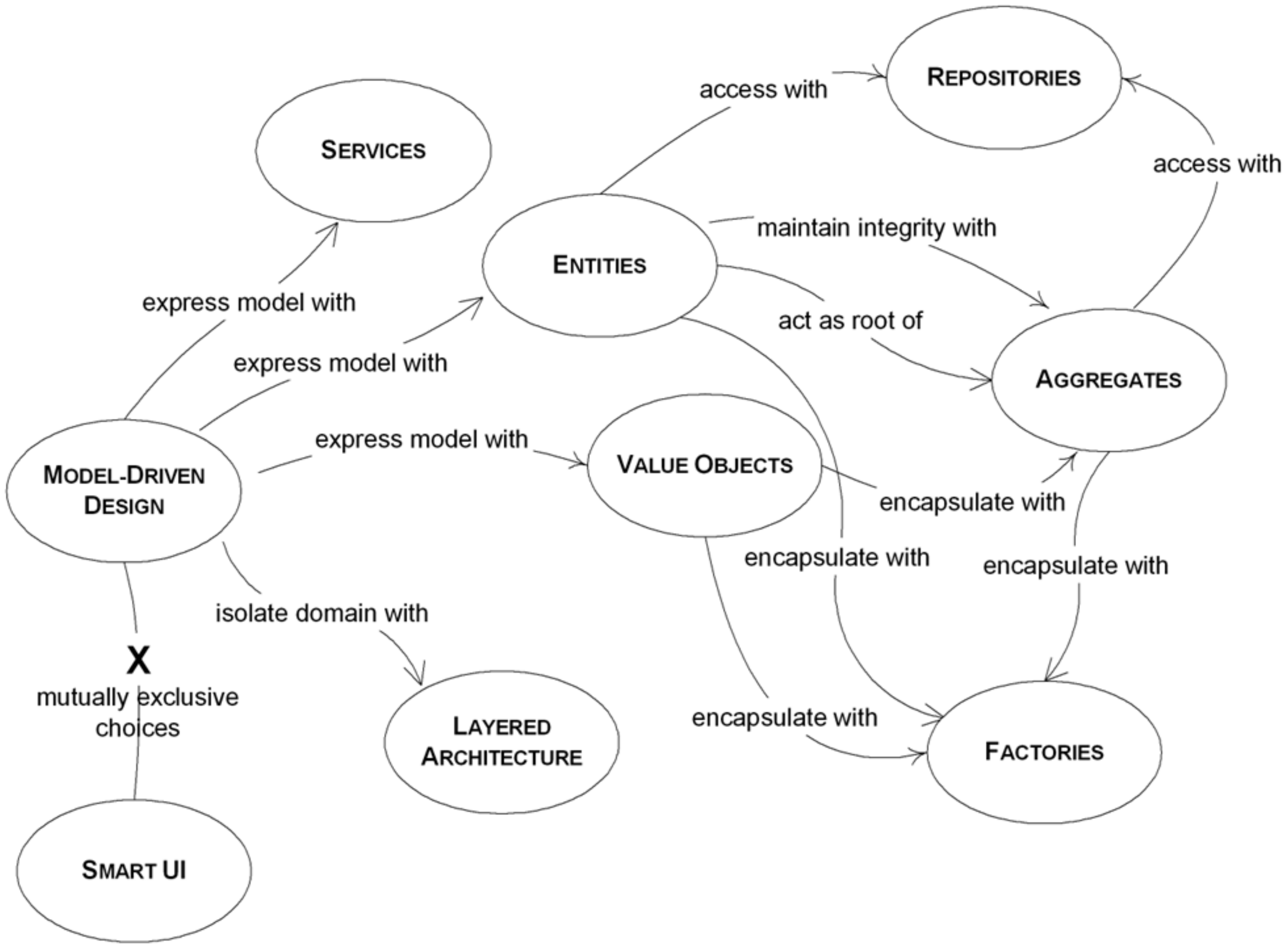
DDD

- From Domain-Driven Design by Eric Evans
  - About putting Business Domain in the center of Software
- DDD in short
  - A project consists of one or more bounded context
  - Within a Bounded context there exists an Ubiquitous Language around an Domain Model
  - Building Blocks
    - **Entity**, Value Object, **Aggregate**, **Repostory**, Domain Event, Service, Factory


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

DDD CONTINUED

- Entity is an object which is not defined by its attributes but by an ID
- Aggregates is a collection of entities
- Root entities is only way to access entities within an Aggregate
- Repository is exposing a set of methods for accessing domain objects (entities)


REPOSITORIES
access with
SERVICES
access with
ENTITIES
maintain integrity with
express model with
act as root of
AGGREGATES
express model with
express model with
VALUE OBJECTS
MODEL-DRIVEN DESIGN
encapsulate with
encapsulate with
encapsulate with
isolate domain with
X
mutually exclusive choices
encapsulate with
LAYERED ARCHITECTURE
FACTORIES
SMART UI

REPOSITORY

- Repository exposes a set of methods that reflects UIL
- Data is changed through methods and not entities - ensure data is updated correctly
- Repository hides away EF core code from application

```
// Example repository methods
public void AddBook(Book book)
public Book FindBook(int Id)
public void DeleteBook(Book book)
public void UpdateBook(Book book)
public List<book> Books(ICriteria criteria)
// Book not has private setters
public void AddReview(Review review)
public void AddAuthor(Author authors)
...
```

Speaker notes

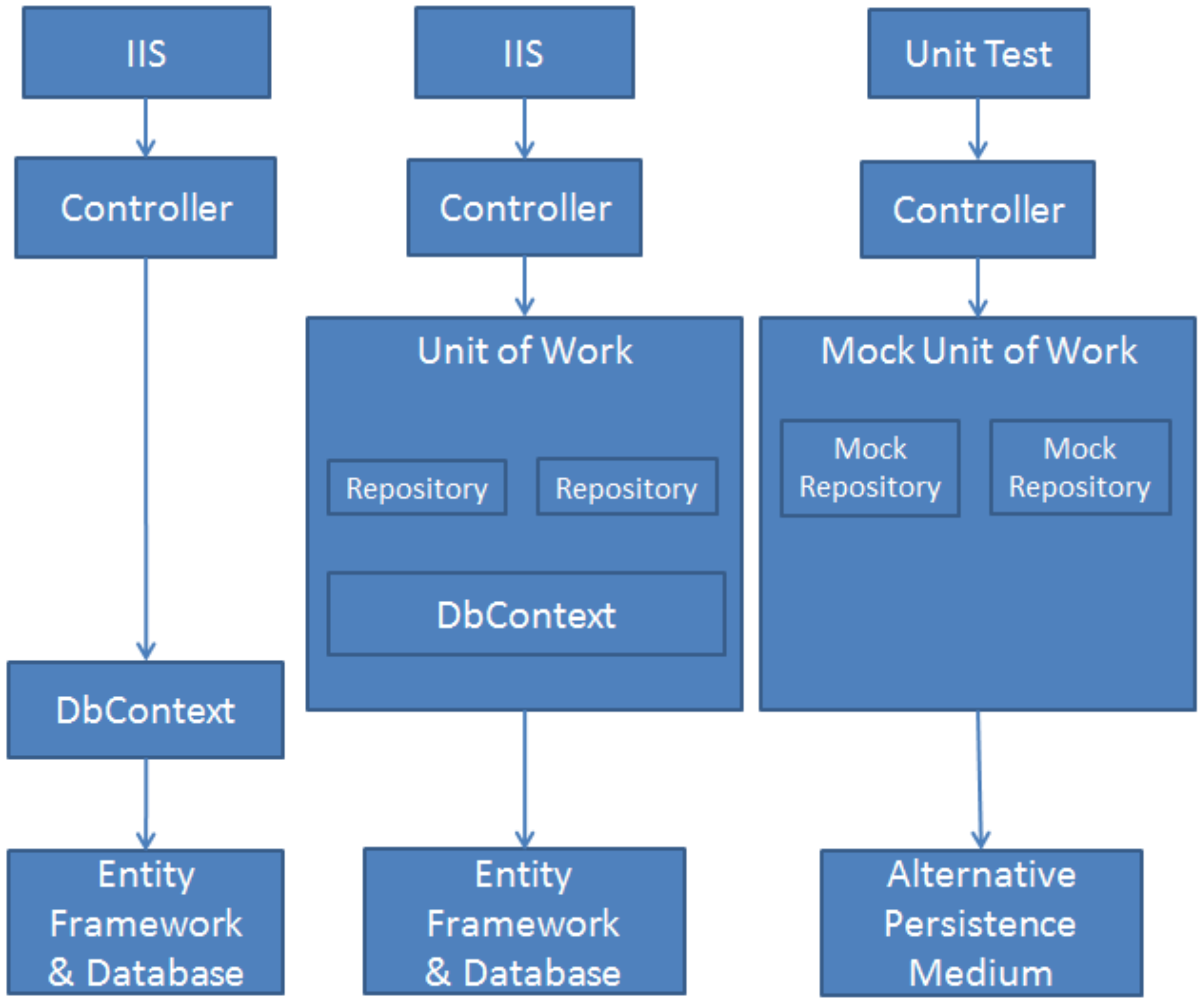
https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-implemenation-entity-framework-core#implement-custom-repositories-with-entity-framework-core

No Repository

Direct access to database context from controller.

With Repository

Abstraction layer between controller and database context. Unit tests can use a custom persistence layer to facilitate testing.



CONSIDERATIONS

- Use Repository to hide DAL from application
  - + Interchangeable DAL
  - % Can't use O/RM as efficient

UNIT OF WORK

"Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems." - Martin Fowler

- The Unit of Work pattern is made to keep track of all changes made in database
  - Avoid changes that are not written

UOW IN EFCORE

- Could see DbContext as Unit of Work
- Microsoft 'recommends' to build a UnitOfWork/Repository pattern around DbContext
  - Create an abstraction between BLL and DAL
  - Easier to maintain and test -> Changes from DAL don't propagate to BLL

```
public class UnitOfWork : IDisposable {
    private DbContext context = new DbContext();
    private GenericRepository<Book> bookRepository;
    public GenericRepository<Department> DepartmentRepository
    // Return Singleton instance
    }
    public void Save() { context.SaveChanges(); }
    // TODO Implement Dispose()
}
```

Speaker notes

https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application

QUERY OBJECT

- "A Query Object is an interpreter [Gang of Four], that is, a structure of objects that can form itself into a SQL query."
- Makes it possible to create queries without knowing SQL and/or database schema.

EXAMPLE

```
public class BookQuery : IBookQuery {
    public bool LoadAuthor { get; set; } = false;
    public int? AuthorId { get; set; } = null;
    public async Task<IEnumerable<Book>> Execute(
            AppDBContext context) {
        if (AuthorId == null) {
        if (LoadAuthor) return await context.Set<Book>()
                    .Include(b=>.Author).ToListAsync();
        else return await context.Set<Book>().ToListAsync();
        } else return await context.Set<Book>()
            .Where(b =>b.AuthorId==(int)).ToListAsync();
} }
```

QUERY OBJECT CONT.

- Another look on Query objects
  - https://www.rahulpnath.com/blog/query-object-pattern-and-entity-framework-making-readable-queries/
- Library for implementing Query Objects and/or repository. E.g.
  - https://github.com/urfnet/URF.NET

OBJECT MAPPERS

- Transform between Entity classes and DTOs
  - Typically one DTO per 'view'
  - Transformation is done in DI service
- Manually way to create LINQ transformation manually -> Time consuming
- 'Automatic' use a library that make use of 'IQueryable'

OM EXAMPLES

- EF Core in action recommends https://github.com/Automapper/Automapper
- AutoMapper work with convention eg. convert PromotionNewPrice to Promotion.NewPrice since there are a navigational property Promotion

```
var config = new MapperConfiguration(cfg => {
    cfg.CreateMap<Book, BookDto>();
    cfg.CreateMap<Review, ReviewDto>();
});
using (var context = new AppDbContext())
{
    var result = context.Books.
        ProjectTo<BookDto>(config)
        .ToList();
}
```

EXERCISES :)

REFERENCES