Anvendelse af CQS i Booking projektet

Vi skal nu i gang med at implementerer "CQS" mønsteret.

"CQS" mønsteret

CQS står for **Command Query Separation**, og det er et designprincip, der bruges til at strukturere metoder i objektorienteret programmering. Grundideen er:

- **Command**: En metode, der **ændrer tilstanden** i et objekt (fx opdaterer data), men **returnerer ikke en værdi**.
- **Query**: En metode, der **returnerer data** (fx læser en værdi), men **ændrer ikke tilstanden**.

Hovedprincip

> *En metode skal enten være en kommando eller en forespørgsel – aldrig begge dele.*

Hvorfor bruge CQS?

- **Forudsigelighed**: Du ved, at en query ikke har bivirkninger.
- **Testbarhed**: Det er lettere at teste, når metoder har ét klart ansvar.
- **Læsbarhed**: Koden bliver mere forståelig, fordi intentionen er tydelig.
- **Understøtter principper som Single Responsibility**:
    - Hver metode har ét klart formål.
    - Det gør refaktorering og vedligeholdelse nemmere.

Nye UserStories

Booking

- Som kunde vil jeg kunne ***se*** alle mine bookings
- Som kunde vil jeg kunne ***se*** en bestemt af mine bookings
- Som kunde vil jeg kunne ***oprette*** en ny booking til mig
- Som kunde vil jeg kunne ***ændre*** en af mine bookings
- Som kunde vil jeg kunne ***slette*** en af mine bookings

Kunde

- Som administrator vil jeg kunne ***se*** alle kunder
- Som administrator vil jeg kunne ***se*** en bestemt kunde
- Som administrator vil jeg kunne ***oprette*** en ny kunde
- Som administrator vil jeg kunne ***ændre*** en kunde

- Som administrator vil jeg kunne ***slette*** en kunde

Det fremgår at ***se*** alle er `Queries`. De andre er `Commands`

"Som kunde vil jeg kunne ***ændre*** en af mine bookings" er en ændring af `UpdateStartTidCommand`

Følgende vil blive implementeret:

- Som kunde vil jeg kunne ***se*** alle mine bookings
- Som kunde vil jeg kunne ***oprette*** en ny booking til mig
- Som kunde vil jeg kunne ***ændre*** en af mine bookings

For at spare tid vil alle UserStories blive implementeret lagvist og alle på engang i hvert lag.

Boooking.Port.Driving

Queries

```
namespace Boooking.Port.Driving;

public interface IBookingQuery
{
    /// <summary>
    ///     Som kunde vil jeg kunne se alle mine bookings
    /// </summary>
    /// <returns></returns>
    List<BookingDto> GetAll();
}

public record BookingDto(int KundeId, int BookingId, DateTime StartTid, DateTime SlutTid);
```

Commands

Navnet på `UpdateStartTid` ændres så det er mere retvisende. Samtidigt ændres `UpdateStartTidCommand` så den bedre afspejler "Som kunde vil jeg kunne ***ændre*** en af mine bookings"

```
namespace Boooking.Port.Driving;

public interface IBookingCommand
{
    /// <summary>
    /// Som kunde vil jeg kunne ændre en af mine bookings
    /// </summary>
    /// <param name="command"></param>
```

```
    void UpdateBooking(UpdateBookingCommand command);

    /// <summary>
    /// Som kunde vil jeg kunne oprette en ny booking til mig
    /// </summary>
    /// <param name="command"></param>
    void CreateBooking(CreateBookingCommand command);
}

public record CreateBookingCommand(int KundeId, DateTime StartTid, DateTime SlutTid);

public record UpdateBookingCommand(int KundeId, int BookingId, DateTime StartTid, DateTime
SlutTid);
```

OBS !!! - Antipattern - null værdier

Man kunne fristes til at bruge den samme DTO til både `Create` og `Update`. Men.... Det er en rigtig dårlig ide. Dels fordi `BookingId` ikke kendes ved `Create`, og dels fordi der ligger et semantisk tjek i at bruge forskelle typer (vi vil få en compiler fejl hvis vi bruger den forkerte).

Application

`BookingCommandHandler` skal tilpasse ændringerne i `IBookingCommand`

```
using Boooking.Port.Driving;

namespace Booking.Application;

public class BookingCommandHandler : IBookingCommand
{
    private readonly IKundeRepository _kundeRepository;
    private readonly IBookingRepository _repo;
    private readonly IServiceProvider _serviceProvider;

    public BookingCommandHandler(IBookingRepository bookingRepository, IKundeRepository
kundeRepository,
        IServiceProvider serviceProvider)
    {
        _repo = bookingRepository;
        _kundeRepository = kundeRepository;
        _serviceProvider = serviceProvider;
    }

    void IBookingCommand.CreateBooking(CreateBookingCommand command)
    {
        // Load
        var kunde = _kundeRepository.Get(command.KundeId);
```

```
        // Do
        var booking = new Domain.Entity.Booking(command.StartTid, command.SlutTid, kunde,
_serviceProvider);

        // Save
        _repo.AddBooking(booking);
    }

    void IBookingCommand.UpdateBooking(UpdateBookingCommand command)
    {
        // Load
        var booking = _repo.GetBooking(command.BookingId);
        if (command.KundeId != booking.Kunde.Id) throw new Exception("Booking tilhører ikke
kunde");

        // Do
        booking.UpdateStartSlut(command.StartTid, command.SlutTid);

        // Save
        _repo.SaveBooking(booking);
    }
}
```

Bemærk at der er oprettet et nyt repository til at håndterer `Kunde`

`IBookingRepository` Har fået en ekstra metode: `AddBooking`

```
namespace Booking.Application;

public interface IBookingRepository
{
    void AddBooking(Domain.Entity.Booking booking);
    Domain.Entity.Booking GetBooking(int id);
    void SaveBooking(Domain.Entity.Booking booking);
}
```

InfraStructure

Repository

I infrastructor skal nyt repository til at håndterer `Kunde` implementeres

```
using Booking.Application;
using Booking.Domain.Entity;
using Booking.Infrastructor.Database;
```

```
namespace Booking.Infrastructor;

public class KundeRepository : IKundeRepository
{
    private readonly BookingContext _db;

    public KundeRepository(BookingContext db)
    {
        _db = db;
    }

    Kunde IKundeRepository.Get(int id)
    {
        return _db.Kunder.Find(id) ?? throw new Exception("Kunde not found");
    }
}
```

`BookingRepository.cs` Skal ændres således Kunde medtages. Og `AddBooking` skal tilføjes.

```
using Booking.Application;
using Booking.Infrastructor.Database;
using Microsoft.EntityFrameworkCore;

namespace Booking.Infrastructor;

public class BookingRepository : IBookingRepository
{
    private readonly BookingContext _db;

    public BookingRepository(BookingContext db)
    {
        _db = db;
    }

    void IBookingRepository.AddBooking(Domain.Entity.Booking booking)
    {
        _db.Bookinger.Add(booking);
        _db.SaveChanges();
    }

    Domain.Entity.Booking IBookingRepository.GetBooking(int id)
    {
        return _db.Bookinger.Include(b => b.Kunde).First(b => b.Id == id) ?? throw new
Exception("Booking not found");
    }

    void IBookingRepository.SaveBooking(Domain.Entity.Booking booking)
    {
        _db.SaveChanges();
    }
```

```
}
```

Queries

Hele Ideen i CQS er at Commands og Queries håndteres forskelligt og effektivt. For at opnår dette implementeres `QueryHandlers` direkte i infrastructur - dvs. at queries går helt udenom application og domain.

`BookingQueryHandler`

```
using Booking.Infrastructor.Database;
using Boooking.Port.Driving;

namespace Booking.Infrastructor;

public class BookingQueryHandler : IBookingQuery
{
    private readonly BookingContext _db;

    public BookingQueryHandler(BookingContext db)
    {
        _db = db;
    }

    List<BookingDto> IBookingQuery.GetAllByKundeId(int kundeId)
    {
        return _db.Bookinger
            .Where(b => b.Kunde.Id == kundeId)
            .Select(b => new BookingDto(b.Kunde.Id, b.Id, b.StartTid, b.SlutTid))
            .ToList();
    }
}
```

CrossCut

IoC skal opdateres: `DependencyInjection`

```
using Booking.Application;
using Booking.Domain.DomainService;
using Booking.Infrastructor;
using Booking.Infrastructor.Database;
using Boooking.Port.Driving;
using Microsoft.Extensions.DependencyInjection;

namespace Booking.CrossCut;

public static class DependencyInjection
{
    public static IServiceCollection AddBookingCore(this IServiceCollection services)
```

```
    {
        services.AddScoped<IBookingCommand, BookingCommandHandler>();
        services.AddScoped<IBookingRepository, BookingRepository>();
        services.AddScoped<IBookingOverlapCheck, BookingOverlapCheck>();
        services.AddScoped<IBookingQuery, BookingQueryHandler>();
        services.AddScoped<IKundeRepository, KundeRepository>();

        // https://stackoverflow.com/questions/70273434/unable-to-resolve-service-for-type-%C2%A8microsoft-entityframeworkcore-dbcontextopti
        services.AddDbContext<BookingContext>();
        return services;
    }
}
```

UI

`Program.cs` Opdateres så API'et kan afprøves. Og... Der er lavet en ny hjælpe klasse `Bonus` - se kode under `Program.cs`

```
// See https://aka.ms/new-console-template for more information

using Booking.ConsoleUI;
using Booking.CrossCut;
using Boooking.Port.Driving;
using Microsoft.Extensions.DependencyInjection;

Console.WriteLine("Hello, World!");

var serviceProvider = IocManager.RegisterService();
Bonus.SeedDatabase(serviceProvider);

var bookingCommand = serviceProvider.GetService<IBookingCommand>();
var bookingQuery = serviceProvider.GetService<IBookingQuery>();

Console.WriteLine("---- Creating booking ---");
var bookings = bookingQuery.GetAllByKundeId(Bonus.KundeId);

foreach (var booking in bookings)
    Console.WriteLine($"Booking ID: {booking.BookingId}, Start: {booking.StartTid}, End: {booking.SlutTid}");

Console.WriteLine("---- Updating booking ---");
var bookingId = Bonus.GetBookingId(serviceProvider);
bookingCommand.UpdateBooking(new UpdateBookingCommand(Bonus.KundeId, bookingId,
    DateTime.Now + new TimeSpan(0, 0, 30, 0), DateTime.Now + new TimeSpan(0, 1, 0, 0)));


Console.WriteLine("---- Creating booking ---");
```

```
bookingCommand.CreateBooking(new CreateBookingCommand(1, DateTime.Now + new TimeSpan(0, 1,
30, 0),
    DateTime.Now + new TimeSpan(0, 2, 0, 0)));

Console.WriteLine("Done");
```

Bonus

Bonus klassen sørger for at udfylde databasen med test data, samt at sikre at nøgler i `Program.cs` eksisterer i databasen.

```
using Booking.Domain.Entity;
using Booking.Infrastructor.Database;
using Microsoft.Extensions.DependencyInjection;

namespace Booking.ConsoleUI;

public class Bonus
{
    // Dette er en bonus-klasse til at illustrere, at du kan tilføje ekstra funktionalitet
her.
    // Du kan f.eks. tilføje metoder til at håndtere specielle scenarier eller
hjælpefunktioner.
    public static int KundeId { get; private set; }

    public static void SeedDatabase(IServiceProvider serviceProvider)
    {
        using var scope = serviceProvider.CreateScope();
        var db = scope.ServiceProvider.GetRequiredService<BookingContext>();
        // Tilføj seed data til databasen her
        db.Database.EnsureCreated();
        // Eksempel på at tilføje en kunde
        if (!db.Kunder.Any())
        {
            db.Kunder.Add(new Kunde { Name = "Kaj" });
            db.SaveChanges();
        }

        var kunde = db.Kunder.First(a => a.Name == "Kaj");
        KundeId = kunde.Id;
        if (!db.Bookinger.Any())
        {
            db.Bookinger.Add(new Domain.Entity.Booking(DateTime.Now.AddHours(1),
DateTime.Now.AddHours(2), kunde,
                serviceProvider));
            db.SaveChanges();
        }
    }

    public static int GetBookingId(IServiceProvider serviceProvider)
```

```
    {
        using var scope = serviceProvider.CreateScope();
        var db = scope.ServiceProvider.GetRequiredService<BookingContext>();
        return db.Bookinger.First(a => a.Kunde.Id == KundeId).Id;
    }
}
```

Overblik