Documentatie EPDConsole assignment

Architectuur

Onze applicatie volgt de Clean Architecture met vier duidelijke lagen die samen een robuust systeem vormen. De kerngedachte hierbij is dat de businesslogica centraal staat en onafhankelijk blijft van technische implementatie details. De lagen zijn als volgt:

1. Domain
2. Application
3. Infrastructure
4. Presentation

Deze architectuur biedt verschillende voordelen:

- De business logica blijft onafhankelijk van technische implementaties zoals de type data sources
- We gebruiken abstracties en interfaces
  (zoals IPersonRepository en IAppointmentRepository) om een duidelijke scheiding te maken tussen de applicatielogica en infrastructuur
- Door deze structuur kunnen we makkelijk van implementatie wisselen zonder de kernlogica aan te passen
- ASP.NET Core ondersteunt deze architectuur door middel van dependency injection, waardoor het testen van de ApplicationCore (Domain en Application laag) en het uitbreiden van de applicatie eenvoudig blijft

Asynchronische aanpak

Asynchrone programmeren verbetert de responsiviteit van de applicatie door blokkerende oproepen te vermijden. Hierdoor kan de app meerdere verzoeken tegelijk verwerken zonder vertraging.

Voorbeeld:

```
1.          public async Task<T> CreatePerson(T person)
2.          {
3.              await dbSet.AddAsync(person);
4.              await dbContext.SaveChangesAsync();
5.              return person;
6.          }
7.
```

Deze aanpak zorgt voor betere prestaties omdat de applicatie tijdens het wachten op database resultaten andere taken kan uitvoeren.

SOLID Principles

- **Single Responsibility Principle (SRP)**
    - Elke klasse in het project heeft één duidelijke verantwoordelijkheid en verzorgt zijn eigen specifieke taken
    - Voorbeeld: PatientService is exclusief verantwoordelijk voor alle patiëntgerelateerde functionaliteit
    - Dit principe zorgt ervoor dat wijzigingen in één onderdeel geen ongewenste effecten hebben op andere delen binnen de applicatie
- **Open-Closed Principle (OCP)**
    - Alle services en repositories zijn gebaseerd op interfaces
    - Deze aanpak maakt het mogelijk om nieuwe functionaliteit toe te voegen zonder bestaande code aan te passen
    - Nieuwe implementaties kunnen worden gemaakt door interfaces, abstracte methoden en/of abstracte klassen te gebruiken en toe te voegen
    - Voorbeeld: De abstracte klasse Person kan worden uitgebreid met specifieke implementaties zonder de bestaande code aan te tasten
- **Liskov Substitution Principle (LSP)**
    - De klassen Patient en Physician zijn ondergeschikt aan de abstracte klasse Person
    - Objecten van het type Person kunnen vrij worden vervangen door Patient of Physician objecten
    - Dit zorgt voor een flexibel en onderhoudbaar systeem dat gebruik maakt van polymorfisme
- **Interface Segregation Principle (ISP)**
    - De console applicatie biedt de functionaliteit maken en verwijderen voor patiënten en artsen
    - Voor afspraken zijn alleen de specifieke operaties aanmaken en opvragen
    - Dit wordt bereikt door gebruik te maken van gespecialiseerde interfaces:
        - IPersonService voor patiënt- en artsgerelateerde operaties
        - IAppointmentService specifiek voor afspraakfunctionaliteiten
    - Geen onnodige methodes die worden gebruikt
- **Dependency Inversion Principle (DIP)**
    - Alle klassen met afhankelijkheden zijn ontworpen om instantiatie te verwezenlijken via interfaces en constructors
    - Dit zorgt voor een loose coupeling tussen componenten

- Het systeem is flexibel en makkelijker uit te breiden

Repository Pattern

Het Repository Pattern introduceert een extra laag tussen de businesslogica en de database. Deze architecturale aanpak zorgt voor een duidelijke scheiding van verantwoordelijkheden binnen het systeem.

De infrastructuur laag maakt gebruik van abstracties via interfaces, waardoor de businesslogica niet hoeft te weten hoe de data precies wordt opgeslagen. Dit zorgt voor een flexibel systeem waarin verschillende data sources kunnen worden gebruikt zonder wijzigingen aan de kernlogica van de applicatie te moeten maken.

Door deze scheiding wordt de infrastructuur volledig geïsoleerd van de businesslogica, wat resulteert in een systeem dat makkelijker te onderhouden en uit te breiden is. De implementatiedetails van de database worden afgeschermd, waardoor er gefocust wordt op de kernfunctionaliteit zonder zich druk te hoeven te maken over specifieke databases.

Mogelijke uitbreidingen en verbeter punten

Hieronder volgt een opsomming van de mogelijke uitbreidingen en verbeteringen die geïmplementeerd zouden kunnen worden:

- Unit Testing Implementatie
    - Ontwikkeling van uitgebreid unit test framework voor services
    - Implementatie van happy flow en unhappy flow tests per methode
- Verbeterde Exception Handling
    - Implementatie van specifieke exception handling in de Presentation laag
- Performance Optimalisaties
    - Implementatie van FluentAPI voor geavanceerde database queries
    - Optimalisatie van query performance door index querying
- Data Transfer Optimization
    - Meer gebruik maken van DTO's
    - Aanpassing van repository methoden om alleen relevante data op te halen en dit te steken in een gepaste DTO
- Architecturale Verbeteringen
    - Implementatie van CQRS patroon
    - Splitsing van lees- en schrijfoperaties
    - Zorgt voor beterde scalability en performance door gescheiden verwerking