



Fejlesztői Dokumentáció

2025.05.27 Vizsgaremek - Állatorvosi Nyilvántartó Rendszer

Dékány Csaba
Boros Dániel

Tartalomjegyzék

Projekt adatok	2
Dokumentum célközönsége	3
Szoftverkövetelmények	4
Szoftver architektúra	5

Projekt adatok

Projekt neve	Állatorovosi nyilvántartó rendszer	
Téma	Backend és Frontend	A szoftvernek várhatóan nagyobb kitettsége ezen két tantárgy felé
Témavezető	Méhes József	A szoftvernek várhatóan nagyobb kitettsége ezen két tantárgy felé
Fő platform	Web	
Osztály	13T-II	
Csoporttagok	Dékány Csaba	Boros Dániel

Dokumentum célközönsége

Ezen dokumentum célközönsége azok az egyének, akinek a programban hibát kell keresnie, a hibát ki kell javítania, a programot hatékonyabbra kell írnia, át kell vinnie más gépre, át kell írnia más nyelvre, valamint tovább kell fejlesztenie.

Az ő munkájuk megkönnyítésének érdekében jött létre ezen dokumentum.

Szoftverkövetelmények

[TODO]

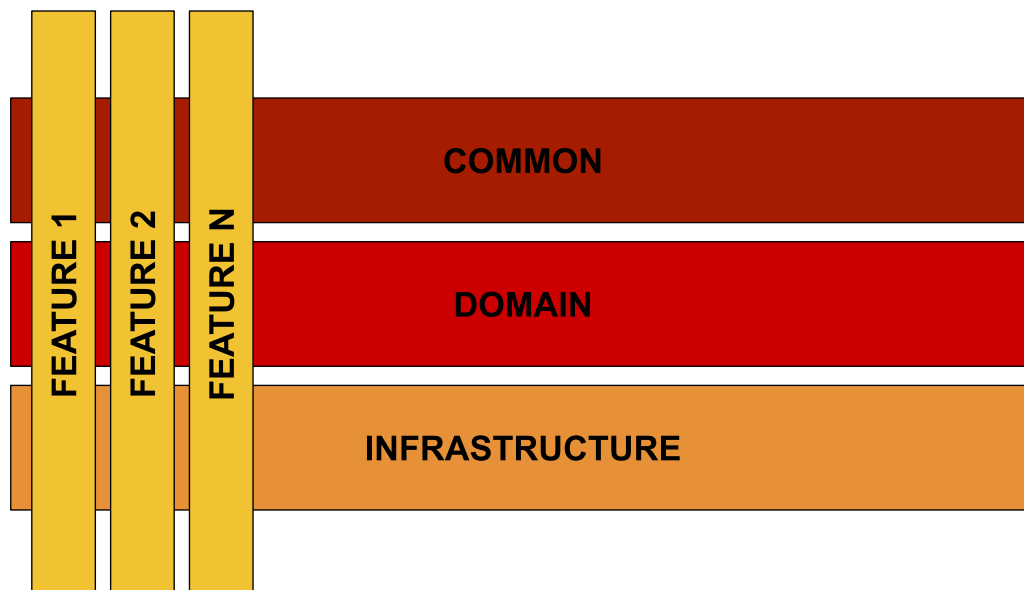
Szoftver architektúra

A **Vertical Slice Architecture** célja, hogy a rendszert funkcionális szeletekre bontsa, amelyek mindegyike önálló egységet képez. Ez az architektúra segíti a kód olvashatóságát, könnyebb tesztelhetőségét, valamint támogatja a gyorsabb és rugalmasabb fejlesztést.


Architektúra Áttekintése

A Vertical Slice Architecture egyedi modulokra bontja az alkalmazást, amelyek mindegyike a következő elemeket tartalmazza:

- Egy adott funkcióhoz tartozó összes logikát.
- A a domain logika, az adatkezelés és a kommunikáció, stb... különálló rétegeit.
- Csak az adott slice-hoz szükséges fájlokat.



(<https://docs.google.com/drawings/d/1-sh4FfvHa7q6Js81EJXVfmPeI5VJw8O5jMYs67iQJIU/edit?usp=sharing>)

 Az alap architektúra felépítés a következő minta elemeit tartalmazza:
<https://github.com/nadirbad/VerticalSliceArchitecture/tree/main>

Projekt Struktúra

Feature

Egy funkcionális szelet (slice) jellemzően a következő elemeket tartalmazza:

```
/Features
  /FeatureName
    /Commands
      - Command.cs
      - CommandHandler.cs
    /Queries
      - Query.cs
      - QueryHandler.cs
    /Models
      - RequestDto.cs
      - ResponseDto.cs
  FeatureNameController.cs
```

Példa:

[TODO: PÉLDA]

Common

A **Common** réteg olyan általános, újrahasználató elemeket tartalmaz, amelyek nem kötődnek szorosan az üzleti logikához, de hasznosak az egész alkalmazásban. Ezek a dolgok általában **technikai infrastruktúrával**, **segédosztályokkal**, vagy **alacsony szintű általános logikával** kapcsolatosak.

Mit írj a Common-ba:

- **Helper osztályok és funkciók:**

- Pl. dátumformázók, string manipulációk, fájlkezelés, stb.

- Példa:

```
class StringHelper {  
    public static function slugify(string $text): string {  
        return strtolower(trim(preg_replace('/[^A-Za-z0-9-]+/', '-',  
$text), '-'));  
    }  
}
```

- **Infrastruktúra elemek:**

- Pl. HTTP kliens, adatbázis csatlakozók, külső API kliensek.
- Ez lehet egy általános "RequestHandler" vagy "HttpClient".

- **Közös validációk vagy szabályok:**

- Példa: e-mail cím validálása.
- Ez általános érvényű, és nem egy adott domainhez kötött.

- **Logger-ek, konfigurációs kezelők:**

- Olyan komponensek, amelyek a rendszer alapműködéséhez szükségesek.

- **Közös interfészek:**

- Pl. RepositoryInterface, EventInterface, stb.
- Ezek segítenek az architektúra alapját lefektetni.

Domain

A **Domain** réteg kizárólag az üzleti logikával és az üzleti szabályokkal foglalkozik. Ez a rendszered "magja", ahol az alkalmazásod valódi értéke megjelenik. Ez **független kell legyen a framework-től** vagy bármilyen külső technológiától.

Mit írj a Domain-be:

- **Entitások:**

- Az üzleti logikát reprezentáló objektumok.
- Példa: User, Order, Product.
- Az entitások gyakran tartalmazznak üzleti szabályokat is (pl. a felhasználó e-mail címének ellenőrzése a példában).

```
class User {  
    private string $email;  
  
    public function __construct(string $email) {  
        if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {  
            throw new InvalidArgumentException('Invalid email  
address');  
        }  
        $this->email = $email;  
    }  
  
    public function getEmail(): string {  
        return $this->email;  
    }  
}
```

- **Value Objects:**

- Azok az objektumok, amelyek egy adott értéket képviselnek és megváltoztathatatlanok.
- Példa: Money, Email, Address.

```
class Money {  
    private float $amount;  
    private string $currency;  
  
    public function __construct(float $amount, string $currency) {  
        $this->amount = $amount;  
        $this->currency = $currency;  
    }  
}
```

```

    }

    public function getFormatted(): string {
        return number_format($this->amount, 2) . ' ' . $this->currency;
    }
}

```

- **Szolgáltatások (Domain Services):**

- Olyan szolgáltatások, amelyek az üzleti logikát valósítják meg, de nem tartoznak szorosan egyetlen entitáshoz.
- Példa: Számlázási logika.

```

class InvoiceService {
    public function calculateTotal(array $items): float {
        return array_sum(array_map(fn($item) => $item->getPrice(),
    $items));
    }
}

```

- **Repository Interfészek:**

- Az adatok tárolásával kapcsolatos interfészek, amelyek az adatkezelést absztrahálják.
- Példa:

```

interface UserRepositoryInterface {
    public function findById(int $id): ?User;
    public function save(User $user): void;
}

```

- **Domain Események:**

- Az eseményalapú architektúra részei, pl. `UserRegistered`, `OrderPlaced`.
- Ezek segítenek más komponensek értesítésében.

Common VS Domain

Kérdés	Common	Domain
Általános technikai funkció?	Igen (pl. fájlkezelés, API-hívások)	Nem
Üzleti logikához kötött?	Nem	Igen
Minden modul használhatja?	Igen	Csak az adott domain-hez kapcsolódik
Alacsony szintű technikai eszköz?	Igen (pl. logger)	Nem
Értékes az üzleti szabályok szempontjából?	Nem	Igen

Fő Elemei

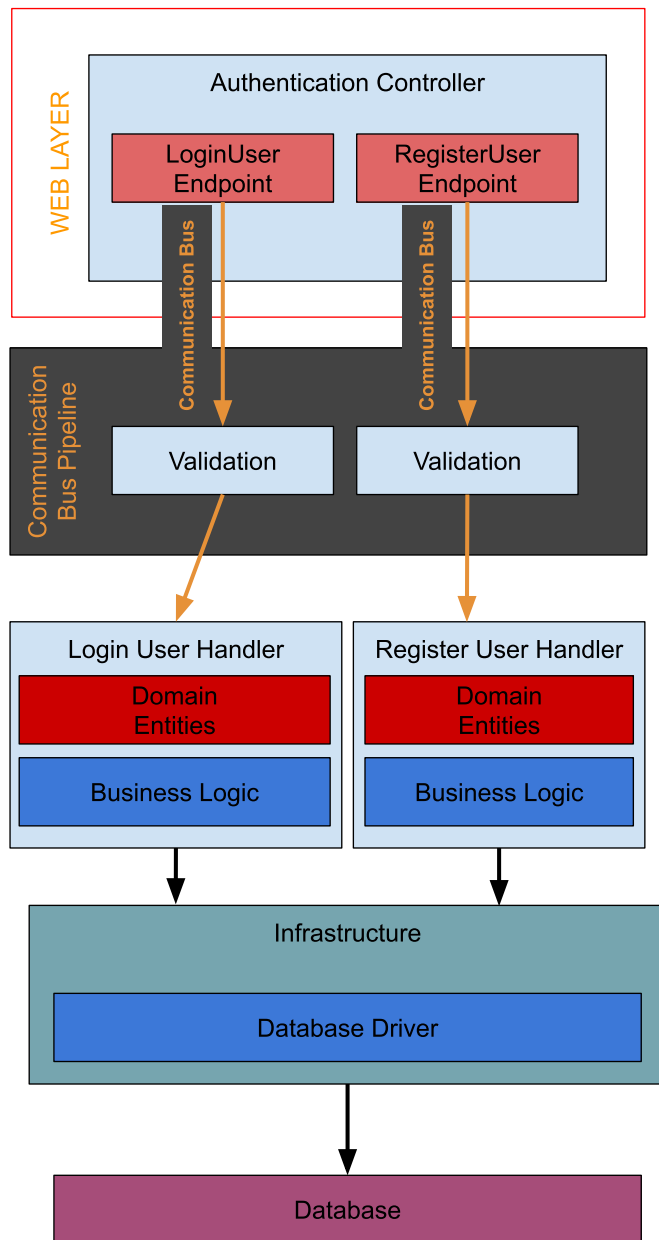
- **Command:** Egy művelet kezdeményezése, például új adat hozzáadása.
- **CommandHandler:** A logika, amely végrehajtja a műveletet.
- **Query:** Adatok lekérése.
- **QueryHandler:** Az adatok visszaadásának logikája.
- **DTO-k (Request/Response):** Az adatokat cserélő modellek, amelyeket az API kliens és a szerver között használnak.
- **Controller:** Az API végpontjai, amelyek a slice-ban meghatározott funkciókhoz tartoznak.

Felelősségkörök szétválasztása (Separation of Concerns)

A Command Bus használata a controller és a feladat végrehajtása között egy tisztább, rugalmasabb és skálázhatóbb architektúrához vezethet. A Command Bus segít elkülöníteni a controller (amely a HTTP kéréseket kezeli) logikáját a tényleges üzleti logikától.

- A controller csak egy Command-ot hoz létre, amely tartalmazza a szükséges adatokat.
- A Command Bus továbbítja ezt egy megfelelő Handler-hez, amely elvégzi az üzleti logikát. Így a controller nem lesz túlterhelt és könnyebben tesztelhető.
- A Command Bus képes úgynevezett "Middleware"-ek használatára (pl. naplózáshoz, validáláshoz, tranzakciókezeléshez), így egységes módon kezelhetőek ezek a funkciók az alkalmazásban.
- Ha a rendszer bonyolultabbá válik, új parancsokat és handler-eket lehet hozzáadni anélkül, hogy meglévő kódot kellene módosítani.
- Mivel a logika a Command Handler-ekben van, könnyen tesztelhetőek izoláltan. A controller tesztelésekor nem kell a komplex üzleti logikával foglalkozni.

Példa a Command Bus demonstrálására



Command Bus Separation of concerns (2)

Technológiai Stack

- ASP.NET Core: Az API megvalósításához.

- **MediatR:** A CQRS-minta (Command and Query Responsibility Segregation) megvalósításához.
- **Entity Framework Core:** Az adatbázis-műveletekhez.
- **FluentValidation:** A bemeneti adatok validálásához.

Előnyök

1. **Olvashatóság:** Egyértelmű, hogy egy funkció mely fájlokat érinti.
2. **Modularitás:** Egy slice teljesen önálló, ezáltal könnyebben bővíthető vagy módosítható.
3. **Tesztelhetőség:** Könnyebben írhatók unit és integrációs tesztek, mivel minden funkció egyetlen egységben helyezkedik el.
4. **Skálázhatóság:** Új funkciók könnyen hozzáadhatók anélkül, hogy más részeket módosítani kellene.

Példa API Végpont

Egy Orders slice példája:

[TODO: PÉLDA]