

# Fejlesztői Dokumentáció

# Table of Contents

Projekt adatok .....	2
Dokumentum célközönsége .....	3
Szoftverkövetelmények .....	4
Szoftver architektúra .....	5
API Konvenciók .....	14
Autentikáció és Autorizáció .....	22
Kérések feldolgozása a kliens oldalon .....	23

# Projekt adatok

<b>Projekt neve</b>	Állatorovosi nyilvántartó rendszer	
<b>Téma</b>	Backend és Frontend	A szoftvernek várhatóan nagyobb kitettsége ezen két tantárgy felé
<b>Témavezető</b>	Méhes József	A szoftvernek várhatóan nagyobb kitettsége ezen két tantárgy felé
<b>Fő platform</b>	Web	
<b>Osztály</b>	13T-II	
<b>Csoporttagok</b>	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]

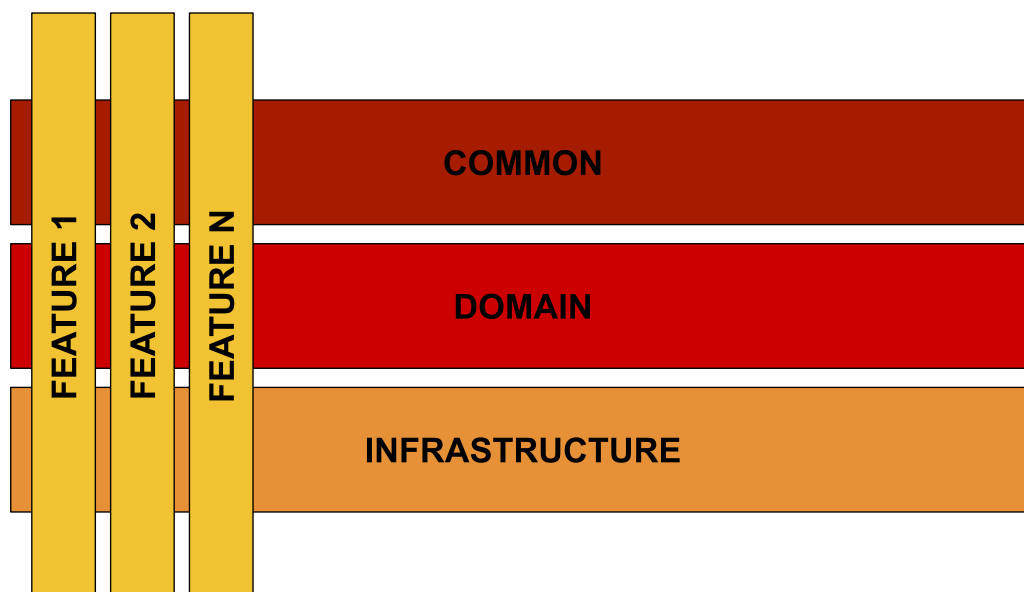
# Szoftver architektúra

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, újrahasználgató 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

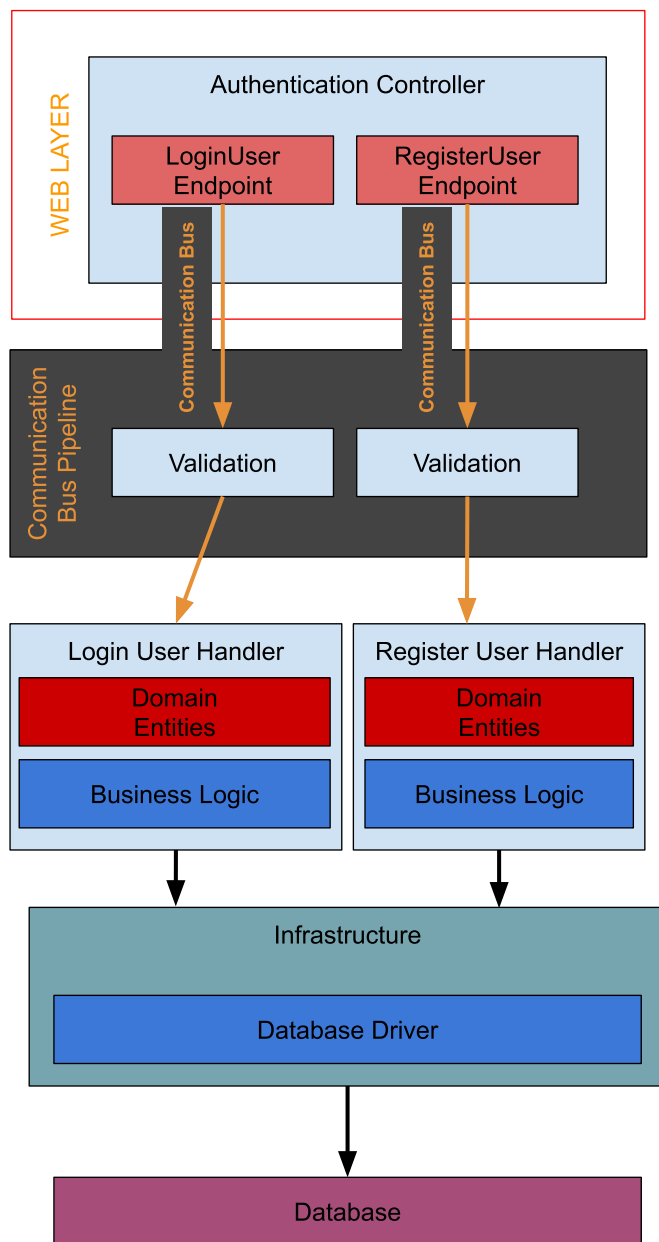
- **Command:** Egy művelet kezdeményezése, például új adat hozzáadása.
- **CommandHandler:** A logika, amely végrehajtja a műveletet.
- **Query:** Adatok lekérése.
- **QueryHandler:** Az adatok visszaadásának logikája.
- **DTO-k (Request/Response):** Az adatokat cserélő modellek, amelyeket az API kliens 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 kezelhetők 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 tesztelhetők izoláltan. A controller tesztelésekor nem kell a komplex üzleti logikával foglalkozni.

## **Communication (command) bus demonstrálása**



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.

# API Konvenciók

Az alábbiakban összefoglaljuk az API-k használatával kapcsolatos legfontosabb szabályokat és elvárásokat, amelyek biztosítják a rendszer megbízhatóságát, biztonságát és teljesítményét.

**i** A végpontok dokumentációját az API Végpont dokumentáció tartalmazza [TODO]

## Kérés-típusok és HTTP metódusok

- **GET:** Csak adatok lekérésére szolgál, nem módosíthatja a szerver állapotát.
- **POST:** Új erőforrások létrehozására vagy műveletek kezdeményezésére használható.
- **PUT/PATCH:** Létező erőforrások módosítására.
  - **PUT:** Teljes erőforrás felülírására.
  - **PATCH:** Részleges frissítésre.
- **DELETE:** Erőforrások törlésére.

## Modellek regisztrálása

Minden a kommunikációhoz szükséges modelt a SharedModels-en keresztül kell regisztrálni, hogy a kliens és a szerver modul által is könnyen hozzáférhető legyen. Ezekre a továbbiakban ApiCommand-ként fogunk hivatkozni.

Az ApiCommand recordot tartalmazó fájlt a `SharedModels/Features/[Feature Group]/[hozzá tartozó feature]` mappában kell létrehozni. A fájlnak tartalmaznia kell a record definíciót és a hozzá tartozó validátort, illetve a választ.

**i** A validátor gondoskodik a kérés validálásáról, mind kliens mind szerveroldalon a MediatR pipeline és a fluent validation segítségével.



A validációra vonatkozó dokumentációt a Fluent Validation komponens dokumentációja tartalmazza

## Példa: LoginUserCommand

Fájl: SharedModels/Features/IAM/LoginUserApiCommand.cs

```
public record LoginUserApiCommand() :
    UnauthenticatedApiCommandBase<LoginUserApiResponse>
{
    public string Email { get; init; }
    public string Password { get; init; }

    public override string GetApiEndpoint()
    {
        return Path.Join(ApiBaseUrl, "/api/v1/iam/login");
    }

    public override HttpMethodEnum GetApiMethod()
    {
        return HttpMethodEnum.Post;
    }
}

public class LoginUserCommandValidator :
    AbstractValidator<LoginUserApiCommand>
{
    public LoginUserCommandValidator()
    {
        RuleFor(x => x.Email).NotEmpty().EmailAddress();
        RuleFor(x => x.Password).NotEmpty();
    }
}

public record LoginUserApiResponse : ICommandResult
{

```



```
public bool Success { get; set; }
public string Message { get; set; }
public string PermissionSet { get; set; }
public string AccessToken { get; set; }
public EntityPermissions GetPermissions()
{
    return new EntityPermissions(PermissionSet);
}
}
```



A példában szereplő `UnauthenticatedApiCommandBase<T>` öröklést az Autentikáció és autorizáció részben fedjük le.

## Kérések feldolgozása a szerver oldalon

### Autentikáció és autorizáció

- Minden hitelesíteni kívánt API-híváshoz **JWT-token** vagy más hitelesítési mechanizmust kell mellékelni.
- A jogosultsági szinteknek megfelelően kell korlátozni a hozzáférést (pl. Admin, Felhasználó, Vendég).

### Adatformátum

- A kérések és válaszok alapértelmezett formátuma **JSON**.
- A válaszok mindig tartalmazzanak a következő metaadatokat:
  - **status**: HTTP státuszkód (pl. 200, 404).
  - **message**: Rövid leírás a válasz állapotáról.
  - **data**: A tényleges adatobjektum (ha van).

## Példa válasz:

```
{
  "status": 200,
  "message": "Sikeres lekérés",
  "data": {
    "userId": 123,
    "username": "johndoe"
  }
}
```

## Sebességkorlátok (Rate limiting)

- Egy adott IP-címről érkező hívások száma percenként legfeljebb **100**.
- A sebességkorlát átlépése esetén a válasz:
  - HTTP státuszkód: **429 Too Many Requests**.
  - Üzenet: "Túl sok kérés. Próbálja újra később."

## Idempotencia

- Az alábbi HTTP metódusok idempotens működést kell biztosítsanak:
  - **GET, PUT, DELETE**: Többszöri meghívásuknak nem szabad különböző eredményt okozni.
  - **POST**: Nem idempotens, de biztosítani kell az ismétlődő kérések megfelelő kezelését (pl. duplikáció elkerülése).

## Hibakezelés

- A szerver által visszaadott hibakódoknak az alábbi kategóriákba kell tartozniuk:
  - **4xx**: Kliensoldali hibák (pl. 400 - Hibás kérés, 401 - Jogosulatlan hozzáférés).

- **5xx**: Szerveroldali hibák (pl. 500 - Szerverhiba).
- A válaszok tartalmazzák a hiba leírását:

```
{  
  "status": 404,  
  "message": "A kért erőforrás nem található."  
}
```

## Titkosítás

- Minden adatforgalmat **HTTPS** felett kell bonyolítani, hogy biztosítsuk az adatok titkosítását.

## API Verziózás

- A stabilitás érdekében az API-kat verziózni kell:
  - Verzió formátuma: `/v1/`, `/v2/` stb.
  - A különböző verziók egymástól függetlenül érhetők el, hogy a meglévő implementációk kompatibilisek maradjanak.

## Végpontok elnevezésére vonatkozó megkötések

A végpontok elnevezésében követendő szabályok a következők:

- **RESTful konvenciók:**
  - A végpontok nevei az erőforrások nevét tükrözzék (többes szám használatával, pl. `/users` a felhasználókhoz).
  - Az URL-ek ne tartalmazzanak igétet, mivel a műveleteket a HTTP metódusok (GET, POST, stb.) határozzák meg.
- **Konzisztens névhasználat:**

- Minden végpont angol nyelven legyen megadva.
- Használjunk **kötőjelet (-)** a több szóból álló nevek elválasztására (pl. `/user-profiles`).
- **Hierarchikus felépítés:**
  - Az erőforrások közötti kapcsolatokat az URL struktúrában tükrözzük:
    - Példa: Egy felhasználó rendelései: `/users/{userId}/orders`.
- **Idempotens és szűkített végpontok:**
  - Az erőforrások részalmazait elérő végpontokat alvégpontként adjuk meg:
    - Példa: Egy specifikus rendelés lekérése: `/orders/{orderId}`.
- **Verziószám az URL-ben:**
  - Az API verzióját az URL első szegmensében tüntessük fel:
    - Példa: `/v1/users`, `/v2/orders`.
- **Speciális műveletek megkülönböztetése:**
  - Ha az alap CRUD (Create, Read, Update, Delete) műveleteken kívüli speciális funkciókat kell támogatni, külön alvégpontot használjunk:
    - Példa: `/users/{userId}/activate` egy felhasználó aktiválására.

**Példák végpontok kialakítására:**

HTTP Metódus	Végpont	Funkció
GET	/v1/users	Minden felhasználó lekérése
POST	/v1/users	Új felhasználó létrehozása
GET	/v1/users/{userId}	Egy konkrét felhasználó adatainak lekérése
PUT	/v1/users/{userId}	Egy felhasználó adatainak frissítése
DELETE	/v1/users/{userId}	Egy felhasználó törlése

## Tesztlefedettségre vonatkozó követelmények

Az API megbízhatóságának, stabilitásának és helyes működésének biztosítása érdekében az alábbi tesztlefedettségi irányelveket kell követni:

### Unit tesztek

- **Cél:** Az API-t támogató üzleti logika és funkciók helyességének ellenőrzése izolált környezetben.
- **Minimum lefedettség:** Az API featureök legalább **90%-os lefedettsége** unit tesztekkel.
- **Tesztelendő komponensek:**
  - Validációs logikák.
  - Egyedi üzleti szabályok.

### Integrációs tesztek

- **Cél:** A komponens funkcionalitásának és az általa használt további komponensekkel való együttműködésének a tesztelése
- **Tesztelendő elemek:**
  - Adatbázis-interakciók (CRUD műveletek).

## End To End (E2E) tesztek

- **Cél:** Az API különböző komponenseinek (adatbázis, külső szolgáltatások, stb.) együttműködésének ellenőrzése.
- **Tesztelendő elemek:**
  - Végpontok teljes adatfolyamatainak ellenőrzése.
  - Külső API-k hívásainak szimulálása és visszatérési értékeik kezelése.
- **Példa:** Egy új rendelés létrehozásának tesztje, beleértve az adatbázisba történő mentést és a külső fizetési szolgáltatóval való interakciót.

## Automatizált tesztelés

- Az összes unit, integrációs és E2E tesztet automatizálni kell, és integrálni kell a CI/CD folyamatba.
- Az API minden módosításakor a teljes tesztsomagnak automatikusan le kell futnia.

## Tesztlefedettségi riport

- A fejlesztési ciklus végén készül

# Autentikáció és Autorizáció

# Kérések feldolgozása a kliens oldalon

Ebben a részben a kliens oldal az API-val való kommunikációjára fogunk kitérni



## Web Interface

- Minden felhasználói interakció a programmal itt kell lefolytatni

## Client Command

- A UI-ből történő kéréseket továbbítja a megfelelő helyre a pipeline-on belül
- A szükséges adatok és utasítások tárolandók bennük

## Client Behaviour

- Client Command futtatásának logikája
- Funkciója:
  - Bemeneti adatok ellenőrzése
  - UI frissítése
  - Hibakezelés, felhasználó tájékoztatása

## Client Command Handler

- Client Command utasításának elvégzését biztosítja
- Az API kommunikációnak az eredményét továbbítja a felhasználónak



## API Command

- API utasítások paramétereit tartalmazza, pl: endpoint, HTTP metódus, beérkezett adatok

## API Command Handler

- API Command lefutását biztosítják

## Generic API Command Handler

- API Commandban meghatározott paraméterek alapján továbbítja az utasítást a Backendnek
- Backend válaszát visszaadja Client Command Handler-nek