

DECLARAȚIE

Subsemnatul Dumitru Alexandru-Răzvan, candidat la examenul de disertație la Facultatea de Matematică și Informatică, domeniul Informatică, programul de master Inginerie software declar pe propria răspundere că lucrarea de față este rezultatul muncii mele, pe baza cercetărilor mele și a informațiilor obținute din surse care au fost citate și indicate conform normelor etice în note și în bibliografie.

Declar că nu am folosit în mod tacit sau ilegal munca altora și că nicio parte din teză nu încalcă drepturile de proprietate intelectuală ale cuiva, persoană fizică sau juridică.

Declar, de asemenea, că lucrarea nu a mai fost prezentată sub această formă vreunei instituții de învățământ superior în vederea obținerii unui grad sau titlu științific ori didactic.

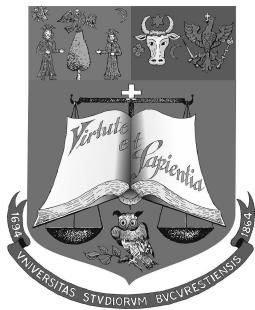
COORDONATOR ȘTIINȚIFIC

CONF. DR. HACIC-CRISTIAN KEVORCHIAN

STUDENT

ALEXANDRU-RĂZVAN DUMITRU

DATA _____



UNIVERSITATEA DIN BUCUREŞTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

Dezvoltarea microserviciilor utilizând
strategia Domain Driven Design

LUCRARE DE DISERTAȚIE

COORDONATOR ȘTIINȚIFIC

CONF. DR. HACIC-CRISTIAN KEVORCHIAN

STUDENT

ALEXANDRU-RĂZVAN DUMITRU

BUCUREŞTI, ROMÂNIA

SEPTEMBRIE 2017

Cuprins

1	Notiuni introductive	4
2	Notiuni teoretice aplicate	8
2.1	CQRS	12
2.1.1	Implementare	20
2.2	Domain Driven Design	23
2.2.1	Entities (Entități)	31
2.2.2	Value Objects (Entități simple)	32
2.2.3	Aggregates (Aggregate)	32
2.2.4	Factories (Fabrici)	34
2.2.5	Repositories (Depozite de Date)	35
2.2.6	Services (Servicii)	35
2.2.7	Domain Events (Evenimente specifice Domeniului)	36
2.2.8	Implementare	37
2.3	Event Sourcing	40
2.3.1	Implementare	47
2.4	Microservicii	50
2.4.1	Implementare	57
3	Descrierea aplicației și dezvoltare ulterioară	59
3.1	Generalități	59
3.1.1	TypeScript	59
3.1.2	Webpack	61
3.1.3	Angular	64
3.1.4	ASP.NET Core	67
3.1.5	RabbitMQ	68
3.1.6	MassTransit	73

3.1.7	EventStore	76
3.1.8	Redis	78
3.2	Arhitectură	80
3.3	Autentificare	82
3.4	Pet Management	94
3.5	Appointment Management	95
3.6	Calendar Management	97
3.7	Pet Timeline	99
3.8	Dezvoltare ulterioară	99
4	Concluzii	101
	Bibliografie	103

Abstract

Cu siguranță complexitatea face parte din câmpul lexical al științei. În domeniul IT complexitatea îmbracă multe forme și este principala grija a arhitectilor, a dezvoltatorilor și a celor care trebuie să finanțeze toate aceste forme de cercetare.

În teza curentă am încercat să aplic principiile de bază ale strategiei *Domain Driven Design* pentru a construi o arhitectură scalabilă, care se folosește de toate posibilitățile curente pentru a menține costurile sub control. În acest sens am adoptat şablonul *Event Sourcing*, evitând astfel stocarea datelor într-o bază de date relațională și am lăsat liberă alegerea mediului în care produsul software este lansat (ASP.NET Core).

Teza include și pașii de dezvoltare a unei suite de servicii care descriu în totalitate funcționalitățile necesare în cadrul unei clinici veterinare. Astfel, am avut posibilitatea de a demonstra modalitatea în care putem structura din punct de vedere arhitectural un produs software care respectă standardele amintite.

Capitolul 1

Noțiuni introductive

Societatea actuală se confruntă cu probleme din ce în ce mai complexe. Astfel, a devenit inevitabilă împărțirea acestora în mai multe probleme de dimensiuni mici. Odată cu această împărțire, responsabilitatea asupra modului în care sunt rezolvate, precum și responsabilitatea asupra resurselor utilizate s-a disipat.

Societățile bazate pe o ierarhie piramidală în vârful căreia se află conducătorul suprem sunt de domeniul trecutului. Companiile mari, indiferent de domeniul de activitate, încă nu pot renunța la întreaga structură ierarhică pe care sunt bazate. IT-ul este unul dintre puținele domeniile în care se poate observa tranziția de la modelul ierarhic de organizare către un model plan în care managementul se face în cadrul fiecărei echipe.

Micromanagementul este văzut ca un stil de management foarte agresiv, în care fiecare decizie a angajaților este supervizată, chiar dacă problema în cauză face parte din aria lor de expertiză. Kamil Lelonek afirma în cadrul articolului *How micromanagement kills creativity and productivity of developers* [1] că un produs software este la fel de lipsit de fermitate ca cerințele tehnice și comerciale pe care acesta trebuie să le îndeplinească. Ader întru totul la această idee și sunt de părere că acest stil de management nu are cum să aducă valoare într-o companie din domeniul IT.

Având în vedere că industria software este întotdeauna într-o continuă schimbare din punct de vedere al tehnologiilor utilizate, precum și a cerințelor comerciale, și în același timp este foarte complexă din punct de vedere al managementului, majoritatea actorilor implicați văd un potențial imens în adoptarea microserviciilor ca organizare arhitecturală a produselor lor. Doar că adoptarea acestor valori presupune modificări din punct de vedere structural a echipelor care dezvoltă aceste produse. Dacă nu

există posibilitatea modificării organigramei curente, probabilitatea adoptării micro-serviciilor ca tehnică de lucru este foarte mică. Melvin Conway afirma, în anul 1968, că organizațiile sunt constrânse să producă sisteme care sunt copii fidele ale relațiilor interumane din cadrul organizației.

În articolul pe care l-a scris pentru DZone.com [2], Arun Gupta, membru fondator al echipei Java EE (Sun Microsystems), definea microserviciile ca un şablon arhitectural care s-a născut odată cu noul val de tehnologii care ajută la minimizarea timpului de dezvoltare, precum și a timpului în care produsul software ajunge în fața clientilor (*continuous integration*). Cultura DevOps, metodologia Agile, și multitudinea de idei generate de acestea sunt rampa de lansare a conceptului de aplicație modulară de mari dimensiuni.

Neal Ford și Rebecca Parsons afirmau în cadrul articolului *Microservices as an Evolutionary Architecture* [3] că "*It is also an example of an evolutionary architecture, which supports incremental non-breaking change as a first principle along multiple dimensions at the structural level of the application.*". În același timp, implementarea unei arhitecturi bazată pe microservicii este un proces anevoios. Dacă alegem această variantă pentru un produs nou, vom avea garanția că modificările ulterioare nu o să aibă un impact prea mare asupra nucleului, acesta rămânând, cel puțin în teorie, ne-schimbăt. Dacă încercăm să rupem bucăți dintr-un produs dezvoltat într-o manieră monolică și să construim microservicii prin care să menținem funcționalitatea curentă, e posibil să ajungem într-un scenariu în care trebuie să întreținem două direcții distincte din punct de vedere arhitectural.

În cadrul cursului *Clean Architecture: Patterns, Practices, and Principles*, Matthew Renze afirma că un arhitect ar trebui să proiecteze un sistem informatic considerând doar ceea ce este benefic pentru cei care interacționează cu acel sistem: utilizatori, testeri, dezvoltatori, persoane care se ocupă de mentenanță. Renze sublinia că gândindu-ne inițial la aceste grupuri de persoane, vom reuși să evităm optimizarea prematură a sistemului, care, după cum spunea Donald Knuth, este rădăcina tuturor problemelor care apar în dezvoltarea de produse software.

Funcția unui arhitect este de a minimiza complexitatea produsului software astfel încât aceasta să nu afecteze termenul limită, dar în același timp să păstreze posibilitatea de dezvoltare ulterioară. Adam Tornhill a comparat depistarea geografică a delicvenților (metodă de investigație care are la bază principiul potrivit căruia domiciliul delicventului se află în aria descrisă de infracțiunile sale) cu depistarea

porțiunilor de cod care afectează produsul software din punct de vedere al complexității și performanței. Cu ajutorul instrumentelor de control al versiunii și al unor utilitare precum CodeCity, Tornhill a reușit să descopere, într-un studiu de caz, că 4% din codul sursă era responsabil pentru 72% din probleme. La acest proiect au lucrat 89 de dezvoltatori, având, în total peste 18.000 de modificări.

În contrapartidă cu ideile prezentate anterior, ar trebui să-l amintim, de exemplu, pe Dan McKinley, care susține alegerea unei arhitecturi cât mai simple, bazată pe tehnologii sigure și plăcute, atât timp cât produsul software care trebuie proiectat nu are o perspectivă clară de dezvoltare. El susține că, de exemplu, dacă trebuie să proiectăm magazine virtuale, PHP este o tehnologie adecvată și unanim acceptată ca fiind cea mai simplă și completă pentru un asemenea produs. Presupunând că ar trebui să integrăm magazinul nostru virtual cu o platformă de distribuție, nu este indicat să rescriem tot produsul în microservicii Java, ci am putea foarte simplu să luăm doar partea de verificare și finalizare a coșului și să dezvoltăm un microserviciu în PHP care să comunice atât cu magazinul virtual cât și cu API-ul expus de platforma de distribuție.

Parafrâzându-l pe Simon Brown, autor al seriei *Software Architecture for Developers*, putem să ne punem următoarea întrebare: Dacă nu am reușit să construim un produs software într-o manieră monolică, care sunt premisele după care ne ghidăm atunci când dorim să construim același produs bazat pe microservicii? Majoritatea problemelor cu produsele care sunt construite într-o manieră monolică sunt generate de abuzurile dezvoltatorilor asupra arhitecturii stratificate. În cazul arhitecturii bazate pe microservicii, o parte dintre aceste abuzuri sunt contracarate *by design*, dar complexitatea se mută în zona de comunicare (mai ales dacă microserviciile sunt complet deconectate, astfel, fiind utilizată mesageria asincronă).

Strategia *Domain Driven Design*, denumită în continuare DDD, este utilizată pentru rezolvarea problemelor de complexitate mare. Aceasta presupune, în linii mari, următoarele:

- O comunicare continuă între cei care dezvoltă produsul software și cei care cunosc procesele afacerii pe care o vizează acest produs.
- Împărțirea domeniului în subdomenii independente. De exemplu, dacă suntem nevoiți să implementăm un produs software pentru optimizarea proceselor dintr-un șantier naval, vom descoperi că această afacere se împarte, de fapt, în procura-

rea materiilor prime, producția propriu-zisă și managementul angajaților (resurse umane). Pentru a rezolva problemele care apar în aceste subdomenii complexe, avem nevoie de experți specializați pentru fiecare zonă de interes.

- Utilizarea principiului *Separation of Concerns* pentru identificarea subdomeniilor.

În același timp, putem să ne bazăm pe diferite abstractizări ale infrastructurii (baza de date utilizată, coada de mesaje) și astfel, avem posibilitatea de a ne concentra pe definirea limitelor subdomeniului la care lucrăm și pe conturarea funcționalităților pe care acesta le va suporta, bazându-ne pe specificațiile primite de la clienți. Eric Evans afirma că DDD nu este o strategie potrivită pentru rezolvarea problemelor care presupun o complexitate tehnică substanțială, dar au o complexitate mică a domeniului de afaceri. Astfel, nu este recomandabil să încercăm aplicarea strategiei DDD produselor software în care complexitatea tehnologiilor utilizate nu este secundată de complexitatea domeniului.

Atât strategia DDD, cât și utilizarea microserviciilor ca abordare arhitecturală, ne ajută să gestionăm complexitatea domeniului prin dezvoltarea unor componente software de dimensiune redusă, flexibile, care rezolvă problemele reprezentative pentru un anumit subdomeniu și care sunt ușor de controlat din punct de vedere managerial.

Scopul lucrării de față este prezentarea generală, din punct de vedere teoretic, a principiilor care stau la baza strategiei DDD și cum pot fi acestea utilizate pentru dezvoltarea unui produs software care utilizează microserviciile ca abordare arhitecturală.

În următorul capitol, am prezentat șablonanele CQRS și Event Sourcing, utilizate în proiectarea și implementarea produselor software și cum sunt acestea folosite pentru a rezolva complexitatea tehnologică a unui produs software care se bazează pe microservicii. De asemenea, am readus în discuție principiile de bază ale strategiei DDD, de această dată exemplificându-le, și am comparat arhitectura monolit cu cea bazată pe microservicii.

În al treilea capitol am adus în discuție tehnologiile folosite, am explicat arhitectura aleasă, atât la nivel macro, cât și în detaliu, iar mai apoi am prezentat modalitatea de autentificare și modulele produsului software dezvoltat.

În ultimul capitol am expus concluziile utilizării acestei suite de principii și tehnologii.

Capitolul 2

Noțiuni teoretice aplicate

În cadrul acestui capitol am prezentat drumul pe care l-am parcurs plecând de la o arhitectură standard, des întâlnită în proiectarea produselor software care au ca principală componentă datele și prelucrarea acestora (CRUD), și ajungând, într-un final la o arhitectură complexă bazată pe microservicii, *event sourcing*, DDD și CQRS.

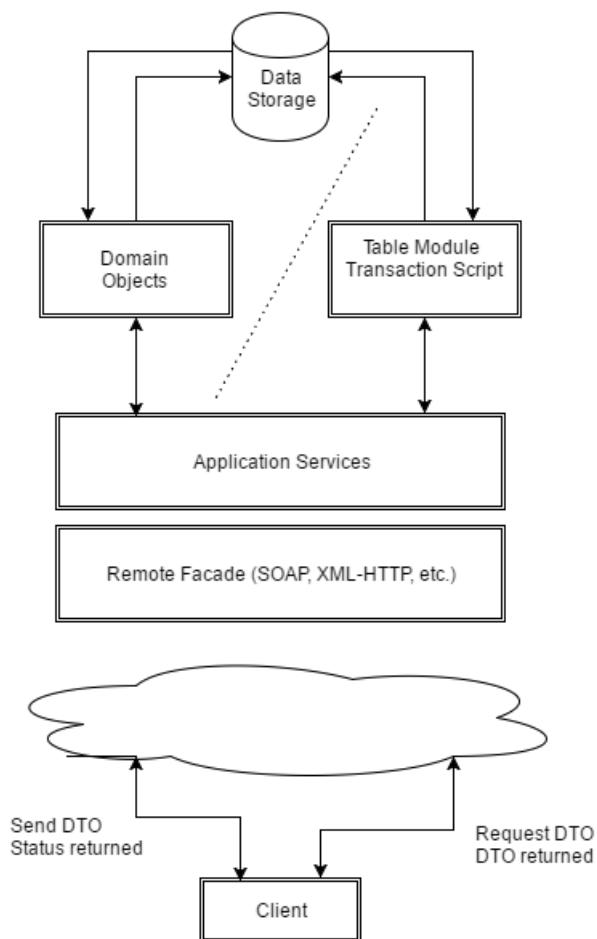


Figura 2.1: Arhitectură standard - produse software CRUD

Note:

- Clientul va interacționa cu produsul software pe baza unor DTO-uri (*Data Transfer Object*).
- Modelele din domeniu pot fi înlocuite cu ajutorul unor şabloane precum *Table Module* sau *Transaction Script*. Dacă nu utilizăm aceste şabloane, suntem forțați să ajungem într-un scenariu în care acestea devin anemice, adică nu descriu în totalitate funcționalitățile pe care le suportă, de cele mai multe ori, având doar proprietăți. În anul 2003, Martin Fowler afirma următoarele: *The catch comes when you look at the behavior, and you realize that there is hardly any behavior on these objects, making them little more than bags of getters and setters.*
- Serviciile utilizează aceste modele anemice doar pentru date. De obicei sunt folosite diferite instrumente pentru a facilita maparea modelelor din domeniu către DTO-uri și invers. Unul dintre acestea este AutoMapper.
- În fiecare strat al produsului software regăsim faptul că a fost construit pentru a reflecta în totalitate datele modelate.

În cadrul arhitecturii prezentate anterior [4], fluxul se poate reduce la următoarele:

1. Clientul cere către *Remote Facade* datele reprezentative ale obiectului X având identificatorul Y (în cazul în care se dorește modificarea unui obiect existent) sau un model de date implicit (în cazul în care se dorește crearea unui nou obiect).
2. *Remote Facade* contactează straturile inferioare ale produsului pentru încărcarea datelor reprezentative sau a celor implicate.
3. Clientul primește aceste date, de obicei folosind un alt model pentru prezentarea acestora utilizatorului final.
4. Utilizatorul final interacționează cu acest model (îl modifică sau completează toate datele necesare creării) și îl trimită înapoi. De obicei este utilizat un DTO identic cu cel care a fost cerut inițial.
5. DTO-ul modificat / nou creat este primit și sunt modificate toate modelele din domeniu în cadrul unei tranzacții.

- În cazul în care acțiunea se finalizează cu succes, clientul primește îmștiințarea că modificările au fost efectuate sau că obiectul a fost creat, altfel primește o listă de erori de validare sau de erori care sunt legate de persistență.

În cadrul tezei *Design patterns for ASP.NET applications* [5], am adaptat arhitectura prezentată anterior pentru a putea fi utilizată în cadrul unui produs software ASP.NET MVC. Componenta de tip *controller* reacționează la acțiunile întreprinse de utilizator, controlând datele și generând un răspuns sub forma unui *view* care este în concordanță cu cerințele sau modificările trimise. În constructorul acestei componente sunt injectate (*Dependency Injection*) toate serviciile pe care le utilizează.

De exemplu, pentru adăugarea unui nou utilizator, am mapat modelul primit la trei modele din domeniu (*UserProfileModel*, *UserModel*, *AddressModel*). Acestea sunt folosite pentru apelarea metodei *Create* din serviciul cu ajutorul căruia realizăm managementul utilizatorilor.

```

1 public class UsersController : BaseController
2 {
3     private readonly IUserService _userService;
4     private readonly IAddressService _addressService;
5     private readonly IUserProfileService _userProfileService;
6
7     public UsersController(IUserService userService, IAddressService addressService,
8                           IUserProfileService userProfileService)
9     {
10         _userService = userService;
11         _addressService = addressService;
12         _userProfileService = userProfileService;
13     }
14     // Metodă POST pentru crearea unui nou utilizator
15     public BsJsonResult New(BsToolbarModel<UsersearchModel, UserNewModel> model)
16     {
17         var userDomainModel = new UserModel();
18         var userProfileDomainModel = new UserProfileModel();
19         var addressDomainModel = new AddressModel();
20         var newUserModel = model.New;
21         // Maparea din modelul expus clientului către modelele de domeniu
22         newUserModel.ToUserProfileDomainModel(userProfileDomainModel);
23         newUserModel.ToAddressDomainModel(addressDomainModel);
24         newUserModel.ToUserDomainModel(userDomainModel);
25         // Apelarea metodei de creare utilizând modelele de domeniu
26         userId = _userService.Create(userDomainModel, userProfileDomainModel,
27                                     addressDomainModel);
28         /* */
29     }
30 }
```

Extras 1: Componentă de tip *controller* - servicii pentru manipularea datelor

În continuare, serviciul deschide o tranzacție și salvează pe rând entitățile în ordinea pe care acestea o necesită. Avem nevoie de o anumită ordine deoarece avem definite multiple chei externe.

```

1 public class UserService : BaseService, IUserService
2 {
3     public UserService(IUserRepository userRepository,
4         IAddressRepository addressRepository, IUserProfileRepository userProfileRepository,
5         IUnitOfWork unitOfWork) : base(unitOfWork)
6     {
7         _addressRepository = addressRepository;
8         _userProfileRepository = userProfileRepository;
9         _userRepository = userRepository;
10    }
11
12    public IEnumerable<UserModel> GetAll()
13    {
14        return _userRepository.GetAll();
15    }
16
17    public long Create(UserModel userModel, UserProfileModel userProfileModel,
18                      AddressModel userAddressModel)
19    {
20
21        long? userId = null;
22        // Deschiderea tranzacției curente
23        unitOfWork.BeginTransaction();
24        // Salvarea adresei și ajustarea profilului utilizatorului
25        // pentru salvarea cheii externe
26        var addressId = _addressRepository.InsertOrUpdate(userAddressModel);
27        userProfileModel.Id_Address = addressId;
28        // Salvarea utilizatorului
29        userId = _userRepository.InsertOrUpdate(userModel);
30        // Salvarea profilului acestuia, după completarea cheii externe
31        userProfileModel.Id_User = userId.Value;
32        _userRepository.InsertOrUpdate(userProfileModel);
33        // Operarea tranzacției curente
34        unitOfWork.Commit();
35        return userId.Value;
36    }
37 }
```

Extras 2: Serviciul cu ajutorul căruia realizăm managementul utilizatorilor

Modelele de domeniu folosite pentru crearea unui nou utilizator sunt folosite și în pagina de profil a acestuia. Astfel, ne aflăm într-un scenariu în care suntem nevoiți să încărcăm toate informațiile unui utilizator pentru afișarea profilului, care, în teorie, este alcătuit doar dintr-o fractiune din aceste informații. Această problemă este generată de faptul că utilizăm aceleași modele de domeniu și pentru scriere și pentru citire (de exemplu: modelele pe care le suportă ca parametru metodele definite în clasele *UserService* și *UserRepository*, *UserProfileRepository*, *AddressRepository*).

2.1 CQRS

În cadrul acestui subcapitol am prezentat şablonul *Command and Query Responsibility Segregation*, denumit în continuare CQRS, care presupune că separarea rolurilor poate îmbunătăti substanțial arhitectura și performanțele produsului software.

Dino Esposito, CTO Crionet și unul dintre cei mai mari autori de literatură de specialitate în zona arhitecturii produselor software *enterprise*, afirma că utilizarea unui singur model pentru a acoperi domeniul și toate aspectele funcționale și non-funcționale ale acestuia este o utopie și, uneori, este aproape de a fi o iluzie optică.

Într-unul din cursurile sale, *Modern Software Architecture: Domain Models, CQRS, and Event Sourcing* [6], acesta prezintă pe larg două perspective diferite de modelare a domeniului unui produs software utilizat pentru monitorizarea scorului unui meci:

1. Utilizând o clasă pentru descrierea comportamentului tabelei, care suportă în același timp și procesarea diferitelor tipuri de evenimente. În acest scenariu pierdem posibilitatea de a răspunde la interogări simple precum: *Care este scorul ultimului meci?*

```
1 public class Match
2 {
3     public Match(...){ /* */ }
4     public Score Score { get; internal set; }
5     public int Period { get; internal set; }
6     public Match Start() { /* */ }
7     public Match Finish() { /* */ }
8     public Match NewPeriod() { /* */ }
9     public Match Goal (...) { /* */ }
10 }
```

Extras 3: Clasa *Match* - descrie comportamentul, precum și starea curentă

2. Utilizând o clasă care descrie doar datele folosite pentru calcularea și afișarea stării curente a produsului software. În acest scenariu pierdem modalitatea de a descrie comportamentul tabelei și mărim riscul de a ajunge în stări nepotrivite.

```
1 public class Match
2 {
3     public Score Score { get; set; }
4     public int Period { get; set; }
5     public int Goals1 { get; set; }
6     public int Goals2 { get; set; }
7 }
```

Extras 4: Clasa *Match* - descrie datele

Dacă produsul software utilizează un singur model pentru a descrie domeniul, atunci niciuna dintre abordările prezentate nu este suficientă. Astfel, pare o idee mai bună utilizarea ambelor modele, primul dintre ele fiind folosit pentru scriere deoarece modifică starea (comandă/*command*), iar cel de-al doilea fiind folosit pentru citire deoarece nu modifică starea (interrogare/*query*).

Sablonul CQRS are la bază principiul *Command-Query Separation* (*Asking a question should not change the answer*) conceput de Bertrand Meyer ca parte a limbajului de programare Eiffel [7]. Cele două idei care compun acest principiu sunt următoarele:

- O comandă (procedură) modifică starea produsului software dar nu întoarce un rezultat.
- O interrogare (funcție sau atribut) întoarce un rezultat fără a modifica starea produsului software.

Martin Fowler afirma că Bertrand Meyer a ales să definească principiul *Command-Query Separation* într-un mod absolut și că în practică există și excepții de la regulile de bază. Un exemplu concluziv este ștergerea unui element din vârful unei structuri de tip stivă (metoda *pop*). Această metodă modifică starea curentă deoarece, după executare, vârful stivei este diferit, dar în același timp returnează valoarea din vârful stivei după ștergerea acesteia, încălcând astfel principiul prezentat.

În cele ce urmează, presupunând că dezvoltăm un produs software pentru managementul unui sănțier naval, descriem două posibilități de definire a serviciului care se ocupă de crearea, editarea și anularea comenziilor pentru materie primă:

1. În cadrul unei arhitecturi standard putem crea un singur serviciu

```

1 public class OrderService
2 {
3     Order GetOrder(int orderId) { /* */ }
4     // interrogarea comenziilor pe baza unui filtru predefinit
5     // OrderSet este o colecție de comenzi
6     OrderSet GetOrders(OrderFilter filter) { /* */ }
7     // interrogarea comenziilor recurente
8     OrderSet GetRecurringOrders() { /* */ }
9     // marcarea unei comenzi ca fiind procesată atunci când găsim distribuitor
10    void MarkOrderAsProcessed(int orderId) { /* */ }
11    void CancelOrder(int orderId) { /* */ }
12    // modificarea unei comenzi existente care nu e deja procesată
13    void UpdateOrder(int orderId, OrderDto order) { /* */ }
14    void CreateOrder(OrderDto order) { /* */ }
15 }
```

Extras 5: Clasa *OrderService*

2. În cadrul unei arhitecturi bazate pe şablonul CQRS, putem crea două servicii distincte, unul pentru citirea datelor și celălalt pentru scrierea acestora

```

1 public class OrderReadService
2 {
3     Order GetOrder(int orderId) { /* */ }
4     OrderSet GetOrders(OrderFilter filter) { /* */ }
5     OrderSet GetRecurringOrders() { /* */ }
6 }
7
8 public class OrderWriteService
9 {
10    void MarkOrderAsProcessed(int orderId) { /* */ }
11    void CancelOrder(int orderId) { /* */ }
12    void UpdateOrder(int orderId, OrderDto order) { /* */ }
13    void CreateOrder(OrderDto order) { /* */ }
14 }
```

Extras 6: Clasele *OrderReadService* & *OrderWriteService*

Observăm că aplicarea şablonului CQRS este facilă și presupune schimbări la nivel arhitectural, fără a adăuga complexitate produsului software.

Serviciul *OrderReadService*:

- Conține doar metodele care întorc date care, mai apoi, sunt utilizate pentru a construi interfața produsului software.
- Poate utiliza o bază de date NoSQL sau o bază de date relațională care se poate afla în prima formă normală pentru a spori performanța produsului software.

Serviciul *OrderWriteService*:

- Conține doar metodele care modifică starea produsului software.
- Ar trebui să utilizeze o bază de date relațională aflată în a treia formă normală pentru a garanta integritatea datelor, modificarea stării produsului software întâmplându-se doar în cadrul unei tranzacții.

Note generale:

- Având în vedere că produsul software a fost împărțit în partea care se ocupă de procesarea comenziilor și cea care se ocupă de procesarea interogărilor, scalabilitatea acestuia a crescut semnificativ. De obicei, în cadrul produselor software web partea care se ocupă de procesarea interogărilor este cea care suferă la capitolul performanței și de cele mai multe ori este cea care trebuie îmbunătățită.

- Avem posibilitatea de a optimiza cele două partii ale produsului software independent.
- Avem posibilitatea de a dezvolta cele două partii ale produsului software utilizând tehnologii diferite.
- O mare parte a produselor software pot fi eventual consistente. Astfel, avem posibilitatea de a spori performanța asigurându-ne de integritatea și consistența datelor doar în partea care se ocupă de procesarea comenzilor.

În cadrul cursului *Modern Software Architecture: Domain Models, CQRS, and Event Sourcing* [6], Dino Esposito prezintă trei modalități de a aplica şablonul CQRS, diferențiate de complexitate și de timpul de dezvoltare:

- Regular CQRS

Presupune folosirea unei singure baze de date, păstrarea modelelor de domeniu, precum și utilizarea acestora pentru definirea regulilor comerciale ale produsului software. Pentru citire, ne putem baza pe cel mai accesibil utilitar, în funcție de performanță/complexitatea dorită: *LINQ to SQL*, *Object-Relational Mapping*, *Dapper* pentru maparea seturilor de date primite de la procedurile stocate. De asemenea, în cazul în care utilizăm un *ORM*, de exemplu *Entity Framework Core*, avem posibilitatea de a expune datele doar pentru a fi citite:

```

1 public class OrdersReadOnlyContext : DbContext
2 {
3     public OrdersReadOnlyContext()
4     {
5         ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;
6     }
7     public DbSet<Order> Orders {get;set;}
8 }
```

Extras 7: Readonly Database - Entity Framework Core

Conform primei idei din cadrul principiului *Command-Query Separation*, o comandă nu ar trebui să întoarcă un rezultat, ci doar ar trebui să modifice starea produsului software. Astfel, este indicat ca imediat după ce o comandă a fost executată cu succes, utilizatorul să fie redirecționat către metoda care se ocupă de citirea entității curente pentru a evita multiplă trimitere a aceluiași formular.

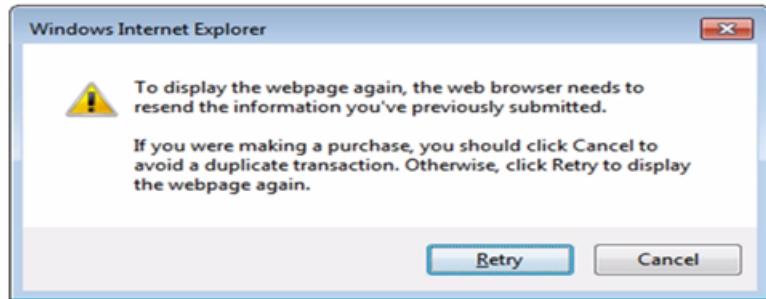


Figura 2.2: Mesaj de confirmare a trimiterii aceluiasi formular - Internet Explorer

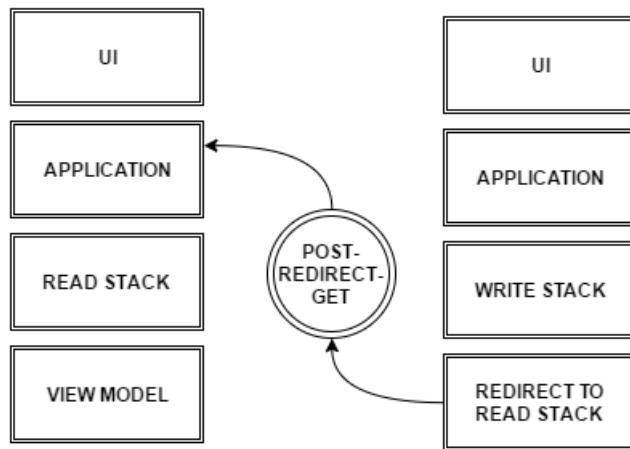


Figura 2.3: řablonul *Post-Redirect-Get*

- Premium CQRS

Are la bază faptul că o comandă declanșează un flux de lucru în cadrul produsului software. Astfel, presupune folosirea a două baze de date, una pentru păstrarea acțiunilor declanșate de comenzi și una pentru stocarea datelor într-o formă convenabilă pentru citire. Starea produsului software este în continuare menținută într-o manieră tranzacțională. Inconvenientul apare în cadrul sincronizării datelor. În funcție de cerințele comerciale, avem următoarele posibilități de sincronizare:

- Procesarea cu succes a unei comenzi declanșează actualizarea sincronă a datelor.
- Procesarea cu succes a unei comenzi declanșează actualizarea asincronă a datelor.

- Existența unei activități planificate la nivel de baze de date pentru replicarea acestora la un moment predefinit sau la un interval predefinit.
- Existența unei activități la nivel de baze de date pentru replicarea acestora care poate fi declanșată manual, la nevoie.

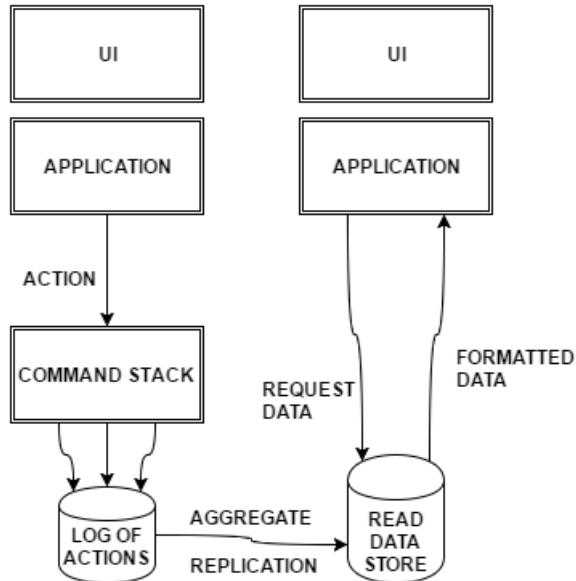


Figura 2.4: Arhitectură CQRS Premium

Arhitectura prezentată în figura anterioară presupune definirea conceptului de sarcină în cadrul produsului software. Fiecare acțiune pe care o primim de la utilizator presupune, de fapt, primirea unei sarcini care definește un flux de lucru. O sarcină este un set de comenzi, împreună cu evenimentele pe care acestea le generează în cadrul produsului software de-a lungul procesării. Evenimentele sunt utilizate în cadrul produsului software pentru a notifica entitățile interesate de întâmplarea unui fapt. Toate acestea sunt salvate în baza de date *LOG OF ACTIONS* (indiferent de natura acesteia: relațională, NoSQL), aggregate și utilizate pentru replicarea stării produsului software în cadrul bazei de date folosite pentru citire (*READ DATA STORE*).

- Deluxe CQRS

Presupune ajustări ample din punct de vedere arhitectural: introducerea unui canal de comunicare pe care sunt scrise evenimentele și comenziile din cadrul produsului software, precum și definirea unor elemente numite *Saga* care sunt utilizate pentru a stoca starea sistemului și pentru definirea fluxurilor de lucru.

```

1 public class OrderSaga : Saga<OrderData>,
2     IStartWith<CreateOrderCommand>,
3     ICanHandle<CancelOrderCommand>,
4     ICanHandle<UpdateOrderCommand>,
5     ICanHandle<MarkOrderAsProcessedCommand>,
6     ICanHandle<FoundDistributorEvent>,
7     ICanHandle<PaymentCompletedEvent>,
8     ICanHandle<PaymentFailedEvent>,
9     ICanHandle<OrderDeliveryStartedEvent>,
10    ICanHandle<OrderDeliveredEvent>
11 {
12     public void Handle(CreateOrderCommand message)
13     {
14         var request = OrderRequest.Factory.Create(
15             Mapper.FromCreateOrderCommandToOrderRequest(message));
16         var response = _orderRepository.CreateOrderFromRequest(request);
17         if(!response.Success)
18         {
19             var failureEvent = new OrderCreationFailedEvent(request.Id,
20                     response.ErrorMessage);
21             Bus.RaiseEvent(failureEvent);
22             return;
23         }
24
25         var createdEvent = new OrderCreatedEvent(request.Id, response.AggregateId);
26         Bus.RaiseEvent(createdEvent);
27     }
28 /* metode Handle pentru fiecare dintre comenzi și evenimente */
29 }
```

Extras 8: Element *Saga* - *OrderSaga*

Cu ajutorul interfeței *ICanHandle* am specificat faptul că elementul de tip *Saga* este interesat de un anumit eveniment care se petrece în cadrul produsului software și, de asemenea, că știe cum să proceseze o anumită comandă. Elementele de tip *Saga* trebuie să poată fi identificate în mod unic, ori printr-un GUID, ori printr-un identificator al obiectului căruia îi mențin starea în cadrul fluxului de lucru (în cazul nostru prin numărul comenzi).

Canalul de comunicare (*Bus*) cunoaște o colecție de obiecte care așteaptă să fie înștiințate de apariția unei comenzi sau a unui eveniment, astfel, mesajele de intrare sunt redirecționate cu succes. Pe lângă această colecție, avem referințe interne pentru elementele de tip *Saga*, precum și o colecție de elemente de acest tip care sunt în desfășurare.

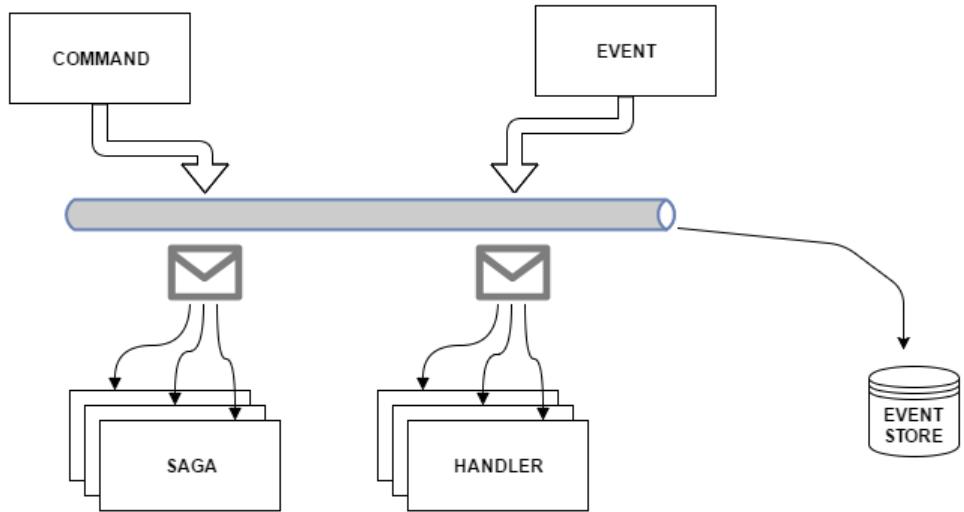


Figura 2.5: Arhitectură CQRS Deluxe

Complexitatea arhitecturală crește indiferent dacă alegem dezvoltarea unui canal de comunicare propriu sau dacă utilizăm unul dintre produsele deja existente: NServiceBus, Rebus, MassTransit. În schimb, produsul software devine mai flexibil, noile funcționalități rezumându-se la implementarea unor elemente de tip *Saga* sau la dezvoltarea posibilității de a procesa evenimentele pe care nu le suportă până în acest moment.

În cadrul extrasului anterior am prezentat un element de tip *Saga* prin care am definit fluxul de lucru pentru o comandă de materie primă. Ca urmare a procesării comenzi *CreateOrderCommand* am scris pe canalul de comunicare evenimentul *OrderCreatedEvent* sau evenimentul *OrderCreationFailedEvent* în funcție de modificarea sau nu cu succes a stării produsului software.

```

1 public class EmailHandler : Handler,
2     IHandleMessage<OrderCreationFailedEvent>,
3     IHandleMessage<OrderCreatedEvent>
4 {
5     public void Handle(OrderCreationFailedEvent message)
6     {
7         var emailBody = string.Format("Your request {0} could not be processed." +
8             "We encountered the following error {1}", message.RequestId,
9             message.ErrorMessage);
10        EmailService.Send("josh@mcgregor.com", body);
11    }
12    /* metode Handle pentru celelalte evenimente */
13 }
```

Extras 9: Arhitectură CQRS Deluxe - procesare evenimente

2.1.1 Implementare

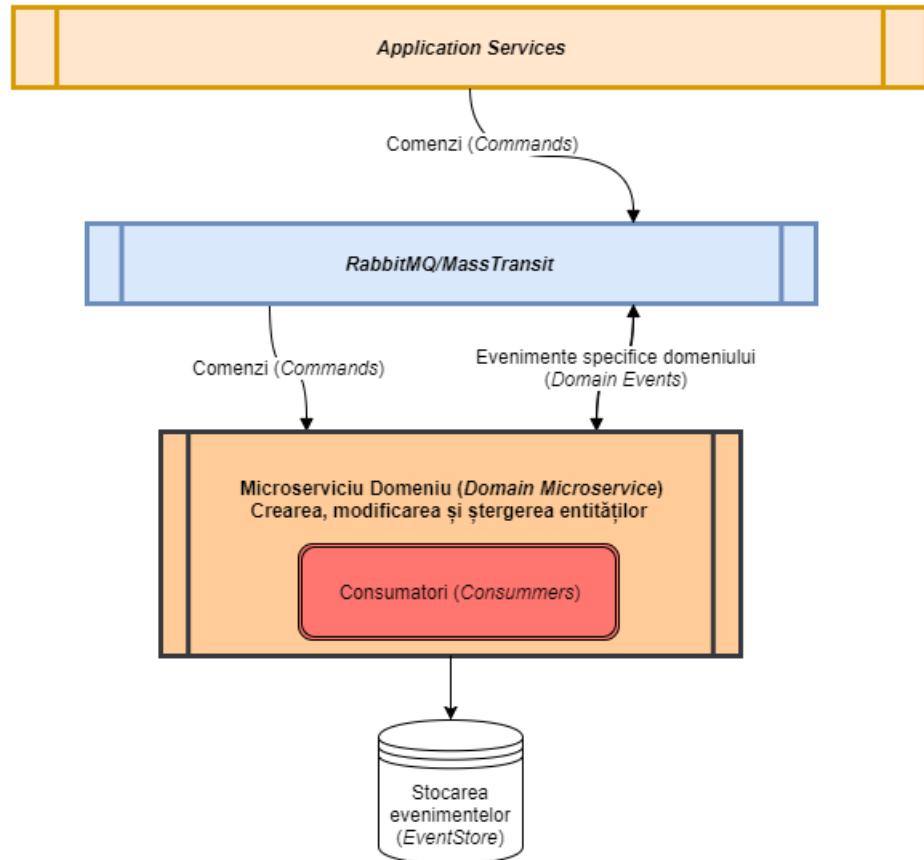


Figura 2.6: Arhitectura aplicației WanVet - Comenzi

În cadrul produsului software pe care l-am dezvoltat avem următoarele niveluri: UI (*Presentation*), Application, Domeniu (*Domain*) și Infrastructură (*Infrastructure*).

Procesarea unei comenzi presupune următoarele:

1. Utilizatorul trimite o cerere sau face o anumită acțiune care se traduce într-o comandă.
2. Această comandă ajunge într-un *controller API* și de acolo este transmisă către nivelul *Application*.
3. În nivelul *Application* un serviciu preia comanda și o trimită către agentul de intermediere a mesajelor (RabbitMQ) utilizând MassTransit.
4. Microserviciile care sunt interesate de această comandă (de obicei unul singur) o preiau și o procesează în cadrul unui consumator (*consumer*).
5. Sunt salvate evenimentele noi pe care le identificăm în cadrul agregatelor.

6. Procesarea se poate finaliza cu emiterea unuia sau mai multor evenimente. Acestea ajung în cadrul agentului de intermediere a mesajelor și sunt preluate pentru procesare de microserviciile interesate.

```

1 public class CreatePetCommandConsumer : IConsumer<ICreatePetCommand>
2 {
3     private readonly IRepository _repository;
4
5     public CreatePetCommandConsumer(IRepository repository)
6     {
7         _repository = repository;
8     }
9
10    public async Task Consume(ConsumeContext<ICreatePetCommand> context)
11    {
12        var command = context.Message;
13        var pet = new Pet(command.OwnerId, command.Name, command.Breed, command.Sex,
14                           command.Species, command.ColorHex, command.BirthDate,
15                           command.ProfileImageUrl);
16        await _repository.SaveAsync(pet);
17        await context.Publish<IPetCreatedEvent>(new { Version = pet.Version,
18                           Timestamp = DateTimeOffset.UtcNow,
19                           MessageType = typeof(IPetCreatedEvent).Name, AggregateId = pet.Id,
20                           OwnerId = pet.OwnerId, OwnerEmail = command.OwnerEmail,
21                           BirthDate = command.BirthDate, Name = pet.Name, Breed = pet.Breed,
22                           Sex = pet.Sex, ColorHex = pet.ColorHex, Species = pet.Species,
23                           ProfileImageUrl = pet.ProfileImageUrl });
24    }
25 }
```

Extras 10: Procesarea comenzii CreatePetCommand

```

1 // Configurarea rutelor Nancy
2 Get("/{id}", args => Response.AsJson(_bus.Execute<GetPetQuery, PetReadModel>
3     (new GetPetQuery(args.id))));
4 public class GetPetQueryHandler : IQueryHandler<GetPetQuery, PetReadModel>
5 {
6     private readonly IRedisService _redisService;
7     public GetPetQueryHandler(IRedisService redisService)
8     {
9         _redisService = redisService;
10    }
11    public PetReadModel Handle(GetPetQuery query)
12    {
13        var pet = new PetReadModel();
14        pet = _redisService.HashGet<PetReadModel>(${pet.RedisKey}, ${query.Id},
15            CommandFlags.PreferMaster);
16        return pet;
17    }
18 }
```

Extras 11: Procesarea interogării GetPetQuery

Procesarea unei interogări presupune următoarele:

1. Nevoia interogării intervine odată cu accesarea unei pagini/secțiuni.
2. Interogarea ajunge într-un *controller API* și de acolo este transmisă către nivelul *Application*.
3. În nivelul *Application* aceasta este transmisă mai departe (*GET/POST*) către microserviciul de interogări la care avem acces direct (*host & port*).

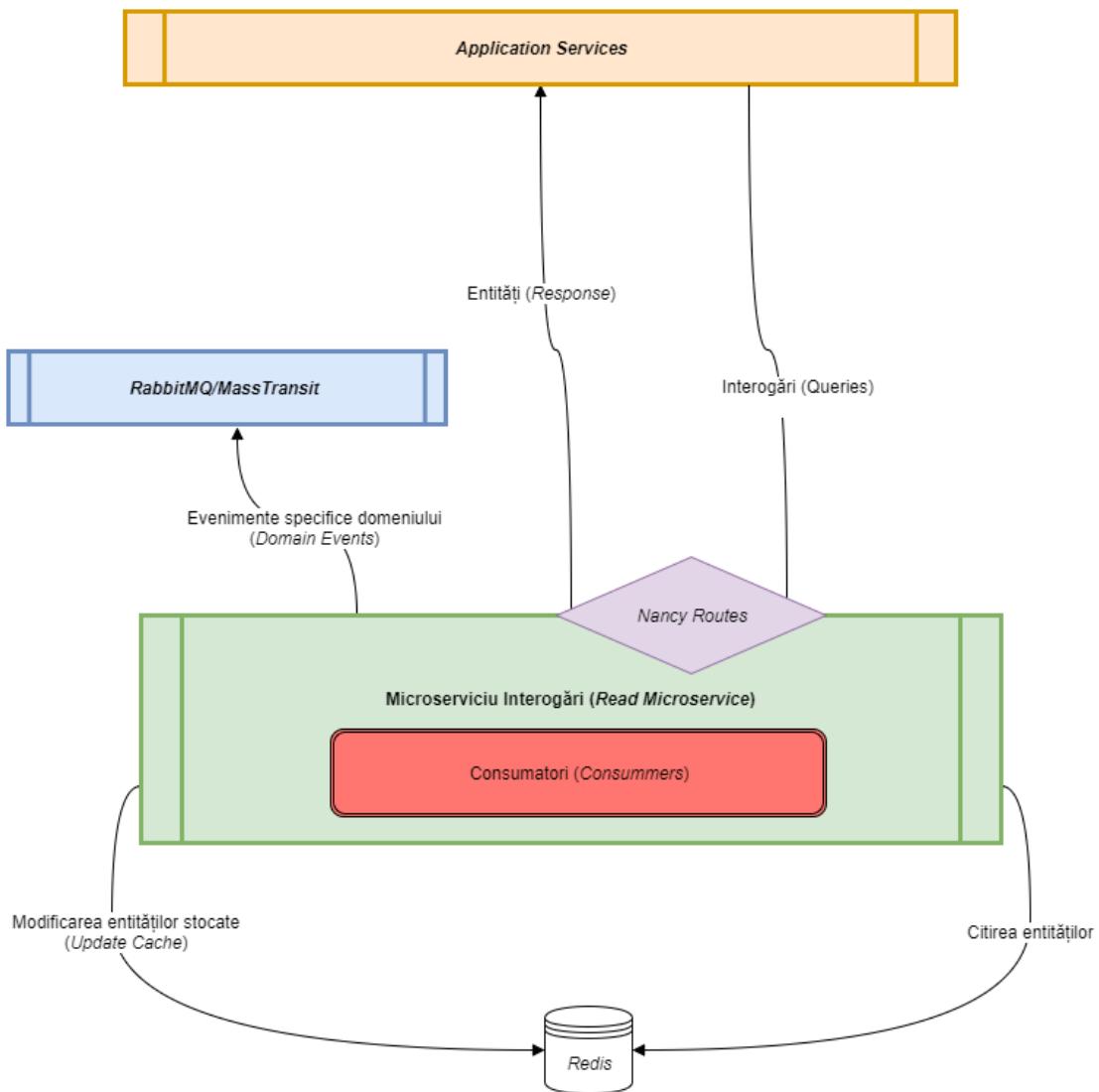


Figura 2.7: Arhitectura aplicației *WanVet* - Interogări

4. Microserviciul de interogări are definite mai multe rute (utilizând pachetul NuGet Nancy), fiecare dintre acestea fiind specifică unei singure interogări.
5. Sunt citite entitățile din Redis și returnate către nivelul *Application*.

6. Din nivelul *Application* și până când acestea ajung în *UI/Presentation* mai pot suferi modificări (grupări, ordonări, etc.)

2.2 Domain Driven Design

În cadrul acestui subcapitol am prezentat strategia *Domain Driven Design*, denumită în continuare DDD, care presupune rezolvarea unor cerințe complexe prin conturarea unui model evolutiv care să descrie în totalitate problemele specifice domeniului.

În cartea sa, *Domain-Driven Design Tackling Complexity in the Heart of Software* [8], Eric Evans afirmă că orice model reprezintă un aspect al realității și că orice produs software este reprezentativ pentru o anumită activitate sau pentru un set de interese care, de fapt, se concretizează a fi domeniul acelui produs software.

Utilizarea strategiei DDD presupune ca, pe tot parcursul dezvoltării și menținării, să existe unele persoane care să cunoască până la cel mai mic detaliu problemele și chestiunile cărora se adresează produsul software. Aceste persoane se numesc *subject-matter expert* (SME) sau experți în domeniu. De exemplu, dacă dorim să dezvoltăm un produs software pentru documentare legislativă utilizând strategia DDD, cu siguranță avem nevoie de un avocat care reprezintă, de fapt, expertul în domeniu.

Rolul unui expert în domeniu este foarte important atât din punct de vedere al chestiunilor administrative, cât și din punct de vedere al dezvoltării produsului software. Inițial, expertul are o anumită idee despre cum ar arăta un model abstract care să descrie domeniul pe care îl reprezintă. Echipa de dezvoltare are, de asemenea, o anumită idee despre cum ar arăta un astfel de model abstract, dar de cele mai multe ori, acesta nu reflectă în totalitate realitatea. De asemenea, utilizatorii finali au anumite așteptări de la produsul software. Crearea unui singur model de la bun început este imposibilă. Astfel, modelele propuse inițial evoluează odată cu produsul, definitivarea unui model abstract comun al tuturor actorilor implicați întâmplându-se în urma mai multor iterări. Aceste iterări presupun următoarele:

- Discuții cu experții în domeniu pentru o înțelegere mai bună a proceselor.
- Discuții cu utilizatorii finali pentru o înțelegere mai bună a nevoilor acestora.
- Refactorizarea codului curent pentru a reflecta ultimele modificări aduse modelului.

Tendința de a contura un singur model în urma iterățiilor este una naturală deoarece ne demonstrează că toți actorii implicați șilefuiesc modelele inițiale pe care și le-au imaginat (fiecare în parte) și converg către o construcție comună și asumată.

Parafrâzându-l pe Eric Evans [8], putem afirma că orice persoană care este implicată în procesul de dezvoltare ar trebui să aibă cunoștințele necesare pentru a descrie acest model în cod sursă. Atingerea acestui nivel de înțelegere presupune o participare activă în discuțiile cu experții în domeniu.

Model Driven Architecture (MDA) și DDD au în comun existența unui model central. În MDA, acest model central trebuie să fie independent de platformă (PIM - *Platform-Independent Model*) și descris cu ajutorul unor diagrame *Unified Modeling Language* (UML). De asemenea, în MDA, există unul sau mai multe modele dependente de platformă (PSM - *Platform-Specific Models*) și un set de interfețe care descriu cum este implementat modelul de bază (PMI) pe o anumită platformă.

În DDD nu există un limbaj impus care să fie utilizat de toți actorii implicați pentru descrierea modelului central/domeniului. Astfel a apărut termenul de limbaj omniprezent (*Ubiquitous Language*) care definește, de fapt, multitudinea termenilor pe care îi utilizează și îi înțeleg pe deplin atât experții în domeniu, cât și echipa de dezvoltare. Acest limbaj omniprezent se reflectă în totalitate în denumirea claselor, a proprietăților și a metodelor acestora.

De-a lungul timpului arhitectii s-au concentrat pe următoarele trei caracteristici: fermitate, utilitate și frumusețe. Brian Foote și Joseph Yoder au identificat un stil de arhitectură software în care primează utilitatea, fiind lăsate în urmă caracteristici precum durabilitatea sau estetica din punct de vedere structural. Brian Marick a denumit acest tip de arhitectură o mare bilă de noroi (*A Big Ball Of Mud*), o definiție suficientă fiind următoarea [9]: *O junglă a codului scris într-un mod delăsător, fără responsabilitate și căreia îi lipsește și cea mai mică formă de organizare. Produsele care se bazează pe această arhitectură prezintă semne grave de dezorganizare, precum și reparații expediente. Datele sunt deținute de mai multe componente ale produsului și sunt plimbate între aceste componente într-o manieră haotică, astfel încât tendința generală este de a globaliza accesul acestor componente la date. Este posibil ca structura acestor produse să nu fi fost definită vreodată. Iar dacă a fost definită, cu siguranță s-a erodat până în punctul în care nu mai poate fi recunoscută.* Principala cauză a evoluției unui produs software către o arhitectură de acest tip este apariția unor cerinte nedокументate și care ar fi trebuit deja să fie implementate. În acel moment apare compromisul. În ge-

neral acesta se traduce în produsul final ca o porțiune de cod sursă care implementează acea cerință și asupra căreia este necesar să se revină (*Throwaway Code*). Având în vedere că produsele software sunt de obicei subestimate și că întotdeauna apar cerințe noi care ar fi trebuit deja să fie implementate, nu mai există timpul necesar revenirii.

Produsele software au tendința să acumuleze caracteristici noi într-un timp foarte scurt. De exemplu, să presupunem că am reușit să dezvoltăm o versiune inițială a unei aplicații Android utilizată pentru a minimiza interacțiunea clientilor cu recepționerii, ospătarii, etc. Această aplicație a fost dezvoltată cu ajutorul unor experți Horeca, din punct de vedere tehnic fiind utilizată strategia DDD. Având în vedere că nu eram siguri dacă această aplicație are posibilitatea să ajungă să fie utilizată, domeniul inițial este unul superficial. De cele mai multe ori, odată cu semnarea primelor contracte, avem și primele caracteristici noi pe care trebuie să le dezvoltăm. Dacă reușim să extindem ca aplicația curentă să fie utilizată și pentru plata diferitelor servicii sau pentru interacțiunea cu alte persoane implicate, domeniul se complică din ce în ce mai mult. Având în vedere că sustenabilitatea aplicației depinde de numărul de contracte semnate, putem presupune că domeniul crește direct proporțional cu numărul de clienți.

Creșterea domeniului presupune ori creșterea numărului de actori implicați (experți, analiști, dezvoltatori, etc), ori creșterea volumului de muncă. Pe termen lung, a doua variantă este nerealistă și improbabilă. Odată cu mărirea echipei intervine și specializarea acestora pe anumite părți din domeniu. Strategia DDD încurajează utilizarea şablonului *Bounded Context*, denumit în continuare BC. Aceasta presupune împărțirea modelului central în mai multe contexte în momentul în care acesta devine prea mare pentru a fi controlat de o singură echipă, interacțiunea dintre aceste contexte fiind explicită și definită a priori. De asemenea, este foarte probabil ca o echipă să dezvolte funcționalități specifice unui singur context.

Din punct de vedere al limbajului utilizat, acesta se specializează la rândul său odată cu definitivarea echipei și a zonelor pe care aceste echipe le au în grija. Odată cu creșterea și împărțirea domeniului în mai multe contexte, limbajul omniprezent inițial devine neactual, fiind relevant doar dacă privim produsul software în ansamblu.

Dacă privim în amănunt putem observa că limbajul omniprezent inițial este utilizat ca ghid în dezvoltarea limbajelor omniprezente specifice fiecărui context în parte. Deși denumirea de contexte delimitate (*Bounded Contexts*) ne trimite cu gândul la izolare, aceste contexte pot avea și concepte comune. Aceste concepte pot fi modelate diferit în funcție de necesitățile pe care trebuie să le acopere în cadrul fiecărui context în parte.

Exact această modelare diferită a acelorași concepte ne poate ajuta în delimitarea și definirea contextelor. De exemplu, în cadrul aplicației pentru Horeca de care aminteam anterior, putem presupune că avem două contexte diferenți *Operations Context* (acoperă drumul produselor de la bucătărie până la client) și *Sales Context* (acoperă integrarea cu operatorii de plăți, precum și procesarea numerarului). Conceptul de comandă și cel de client sunt modelate diferit. În *Operations Context*, comanda este constituită din totalitatea preparatelor, ajustări ale acestora (picant, etc.), precum și când trebuie să ajungă la client (ordinea de servire). În *Sales Context*, comanda este constituită dintr-o listă de înregistrări de tipul preparat-preț, un total de plată, precum și din reguli asupra remunerației pentru serviciul oferit.

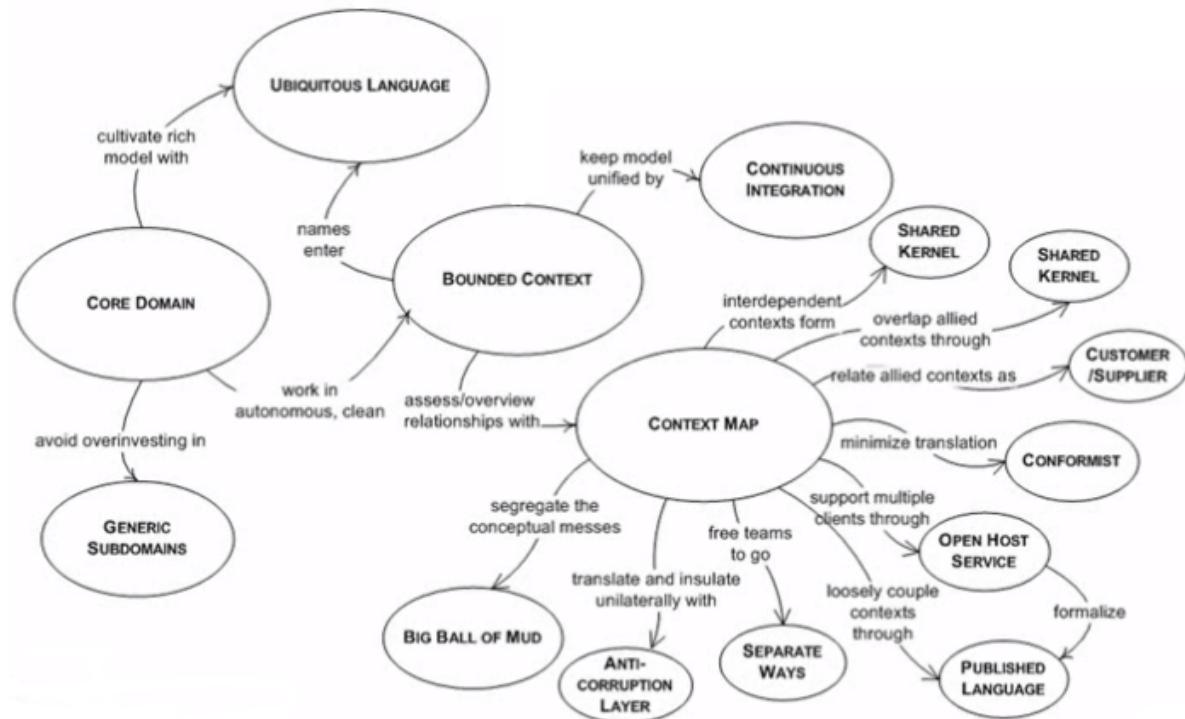


Figura 2.8: Păstrarea integrității modelului inițial [8]

Dacă privim ce reprezintă conceptul de comandă în interiorul *Operations Context*, cu siguranță avem întrebări precum următoarele: Cine achită nota de plată?; Trebuie să generăm factură sau nu?; etc. Este clar faptul că, luând în calcul un singur context, nu putem să descriem totalitatea proceselor prin care trece o comandă în interiorul produsului software. Această problemă este rezolvată de şablonul *Context Map* care presupune identificarea conceptelor, includerea acestora în contextele delimitate care

le înglobează și utilizarea acestor denumiri de contexte ca idei centrale ale limbajului omniprezent general. Pe lângă acestea, recomandă definirea interacțiunilor dintre contexte, precum și modalitatea de deplasare a datelor dintr-un context în altul.

Şablonul *Continous Integration* presupune ca, după ce am reușit să definim un anumit context, să încercăm integrarea acestuia în imaginea de ansamblu a produsului software prin mijloace tehnice precum testare automată. În cazul în care un anumit concept nu este în totalitate reprezentativ pentru un singur context sau dacă nu avem resursele necesare pentru a dezvolta teste care să ne demonstreze că interacțiunea dintre contexte este stabilă și bine definită, avem posibilitatea de a defini un nucleu comun (*Shared Kernel*) la care să participe activ toți cei care sunt interesați de conceptul în cauză.

Răspunzând la următoarele întrebări (formulate inițial de Eric Evans): De ce să implementăm funcționalități specifice pentru această zonă?; De ce nu cumpărăm o soluție care să rezolve această problemă?; De ce nu externalizăm implementarea acestor funcționalități?; avem posibilitatea de a identifica domeniul central (*Core Domain*).

Pe lângă acesta, co-există:

- Subdomeniile generice (*Generic Subdomains*) precum procesarea platilor, managementul angajaților sau a problemelor de marketing care nu fac parte din domeniul central și pentru care există soluții dedicate.
- Subdomeniile care susțin domeniul central (*Supporting Subdomain*). Dezvoltarea acestora poate să cadă în sarcina programatorilor începători, și, de asemenea, pot suferi la capitolul arhitectură atât timp cât nu afectează în vreun fel domeniul central.

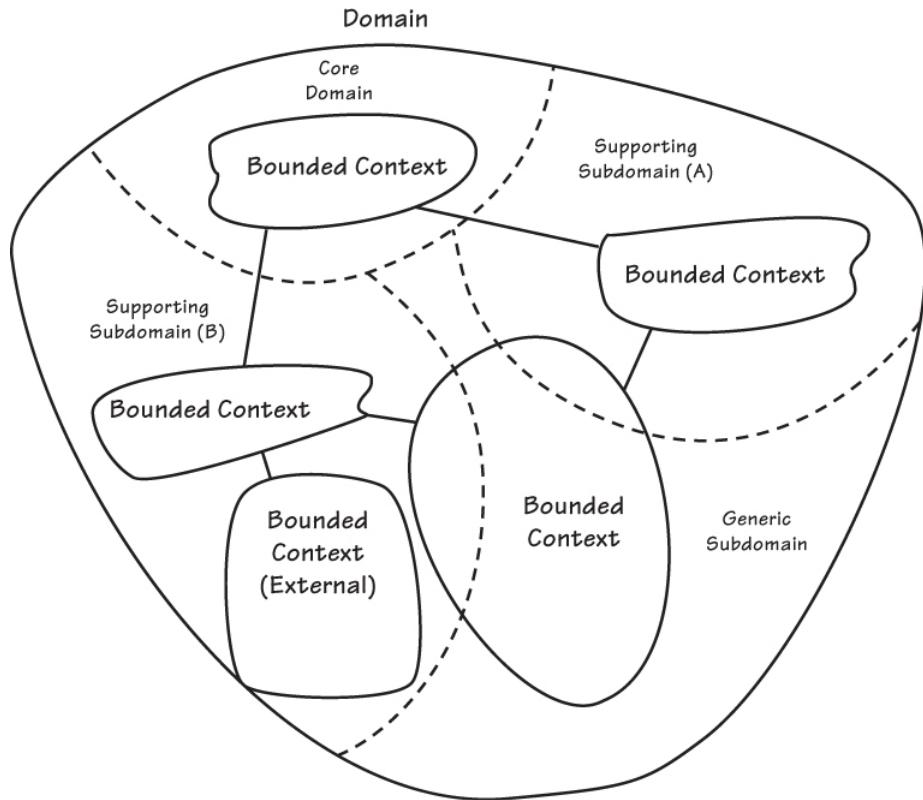


Figura 2.9: DDD Context-Domeniu Central-Subdomenii [10]

Cele trei şabloane prezentate anterior, *Bounded Context*, *Context Map* și *Continuous Integration* sunt utilizate pentru oglindirea zonelor de interes ale unui produs software. Idealistic vorbind, un context ar putea să reprezinte un singur subdomeniu pentru a evita orice încurcătură legată de sensul conceptelor în limbajul adoptat de o anumită echipă. În figura anterioară Vaughn Vernon ne prezintă cum arată în realitate delimitarea dintre context și subdomeniu și de ce DDD este o strategie atât de complicată și greu de implementat.

Având în vedere că problemele din cadrul domeniului sunt de cele mai multe ori de o complexitate sporită, apare necesitatea separării acestora de restul funcționalităților. Cel mai important șablon arhitectural utilizat pentru separarea problemelor din cadrul domeniului de restul funcționalităților este *Layered Architecture*. Eric Evans menționa următoarele straturi de bază [8]:

- *User Interface/Presentation*. Responsabilitățile acestui strat se reduc la afișarea informațiilor și la interpretarea cerințelor actorilor implicați (utilizatori/sisteme externe/servicii de planificare).
- *Application*. Responsabilitatea acestui strat este orchestrarea interacțiunii cu

domeniul astfel încât să fie acoperit orice scenariu funcțional pe care produsul software trebuie să-l suporte. Nu are o stare curentă și, în general, se poate ocupa de validarea inițială a datelor (de exemplu: că o proprietate e-mail este de forma email@provider.numededomeniul).

- *Domain*. Responsabilitățile acestui strat sunt structurarea conceptelor din realitate, păstrarea unei stări durabile și consistente, precum și definirea regulilor de validare a datelor din punct de vedere al domeniului (de exemplu: nu pot exista două comenzi având același identificator).
- *Infrastructure*. Responsabilitățile acestui strat sunt persistența datelor din domeniu (baze de date relaționale/stocarea evenimentelor), descrierea posibilităților de comunicare (canal de comunicare, mesaje), descrierea din punct de vedere arhitectural a produsului software (CQRS), precum și rezolvarea oricăror alte necesități ale straturilor superioare.

În materie de dependențe, putem afirma că orice strat superior are acces la unul sau la toate straturile inferioare. De exemplu, stratul *Presentation* are acces la *Application* și la *Domain*. Această modalitate de stratificare poate duce la o legătură foarte strânsă între straturi. Astfel, modificarea unor concepte legate de infrastructură poate afecta majoritatea straturilor. Având în vedere că stratul *Infrastructure* este global accesibil, există posibilitatea de a renunța la straturile intermediare și de a apela direct baza de date din stratul *Presentation*. Acest comportament care poate fi introdus cu ușurință în cadrul unui produs software bazat pe *Layered Architecture* a fost denumit *Smart UI Anti-Pattern*. Strategia *DDD* se delimitizează strict de acest tip de comportament.

Preluând o parte din ideile prezentate în *Layered Architecture* și aplicând principiul *Dependency Inversion* a apărut conceptul de *Onion Architecture*. Straturile din *Onion Architecture* sunt asemănătoare cu cele prezentate anterior. Astfel putem identifica patru straturi care prezintă un interes crescut:

- *Domain Model*. Responsabilitatea acestui strat este de a descrie entitățile, conceptele și obiectele care compun domeniul.
- *Domain Services*. Responsabilitatea acestui strat este de a descrie procesele din domeniu.
- *Application Services*. Responsabilitatea acestui strat este utilizarea propice a

logicii din domeniu pentru a acoperi toate funcționalitățile care trebuie descrise în cadrul produsului software.

- *Outer*. Responsabilitățile acestui strat sunt externe domeniului: teste, prezentare, stocarea datelor, etc.

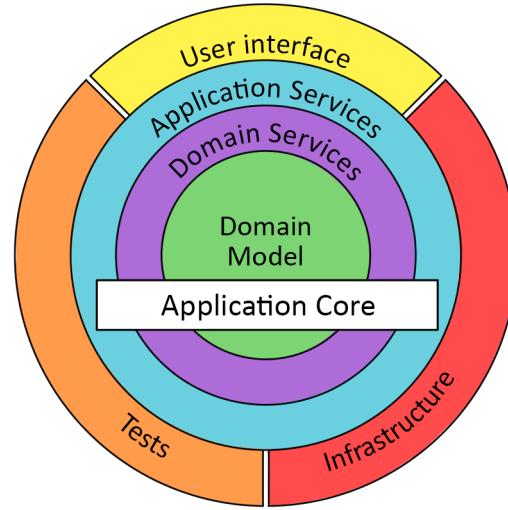


Figura 2.10: *Onion Architecture* [11]

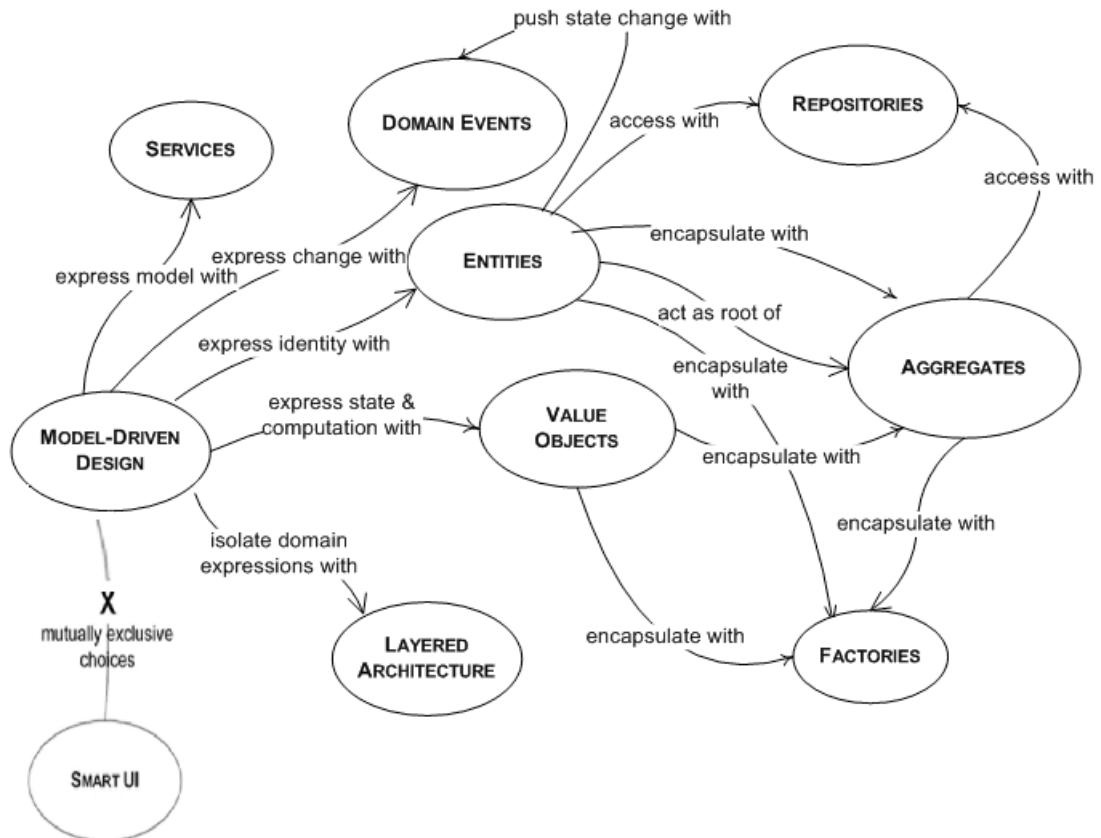


Figura 2.11: Privire de ansamblu - DDD [8]

Stratul *Domain* se împarte în următoarele două mari zone:

- *Domain Model*: *Entities* (Entități), *Value Objects* (Entități simple), *Aggregates* (Agregate), *Factories* (Fabrici), *Domain Events* (Evenimente specifice Domeniului).
- *Domain Services*: *Repositories* (Depozite de Date), *External Services* (Servicii Externe).

2.2.1 Entities (Entități)

Eric Evans definea [8] o entitate ca fiind un obiect care are o anumită identitate (poate fi identificat în mod unic), are o anumită durabilitate (un ciclu de viață) și nu este descris de calitățile sale. De obicei, aceste obiecte conțin o proprietate ID care le ajută să fie identificabile în mod unic în cadrul unui agregat (este considerată o practică defectuoasă utilizarea aceleiași entități în cadrul mai multor aggregate).

De exemplu, la camera de gardă a unui spital se prezintă un pacient având piciorul rupt. Dacă identificarea acestuia se realizează doar pe baza calităților sale (vârstă, nume, prenume) este posibil să luam în considerare un alt pacient (cu aceeași vârstă, nume și prenume) în a cărui istoric medical găsim că este diagnosticat cu hemofilie severă. Această problemă de identificare duce la costuri imense și pregătiri care nu sunt necesare pentru operarea pacientului (hematolog, anestezist, medici, asistenți). În România, într-un scenariu real, pacienții sunt identificați cu ajutorul Codului Numeric Personal.

Există posibilitatea de a fi nevoiți să păstrăm istoricul medical al unei persoane și după ce aceasta se căsătorește și își schimbă numele. Când apar asemenea situații, acestea trebuie descrise în cadrul sistemului software cu ajutorul unor mapări și prin descrierea procedurii de identificare (CNP, actualizarea datelor, etc.).

De asemenea, există posibilitatea de a fi nevoiți să păstrăm unicitatea aceleiași entități (în cazul nostru pacient) din punctul de vedere al mai multor actori diferenți: medic (în cadrul operației) și asistent (post operator).

Întotdeauna trebuie să ne asigurăm că datele pe care le stocăm sunt consistente și că acestea reprezintă ceea ce trebuie să reprezinte pentru cei interesați.

2.2.2 Value Objects (Entități simple)

O entitate simplă este o entitate care nu posedă un identificator unic. Acestea sunt utilizate pentru a descrie caracteristici ale obiectelor.

În general entitățile simple sunt utilizate atunci când suntem interesați de ceea ce descriu (de exemplu: formă, culoare, mărime, etc) și nu atunci când suntem interesați de ceea ce reprezintă/identifică. De aceea, este recomandată modelarea acestora ca fiind neschimbătoare (*immutable*).

De obicei, entitățile simple au puține proprietăți și descriu un anumit concept unitar. De exemplu, în cadrul unui oficiu poștal, avem strada, numărul, blocul, apartamentul, codul poștal, etc. care pot fi proprietăți de tipuri diferite (șir de caractere, număr întreg), dar toate acestea împreună formează conceptul de adresă care poate fi modelat ca o entitate simplă.

Mai multe entități pot referi aceeași entitate simplă (*value object*). Şablonul care ne ajută în acest caz se numește Flyweight. Un *flyweight* este un obiect care minimizează consumul de memorie prin stocarea și împărțirea informațiilor care sunt folosite în mod ușual de către entitățile care referă aceste date similare. De exemplu, am putea avea 1000 de persoane (modelate ca entități) care să aibă asigurarea obligatorie a locuinței la Generali (modelată ca entitate simplă).

2.2.3 Aggregates (Aggregate)

Un agregat este o grupare de entități (*entities*) și de entități simple (*value objects*) care definesc un unic concept delimitat clar în cadrul domeniului. Orice agregat are o entitate bine aleasă pe post de rădăcină. Astfel, această rădăcină poate fi utilizată de alte aggregate sau de alte sisteme ca referință și reprezintă singura cale de acces în spațiul delimitat de granițele aggregatului.

Spațiul definit de granițele aggregatului este unul mărginit deoarece orice sistem software trebuie să asigure integritatea datelor, și, în general, acestea trebuie validate și modificate în grupuri. Aceste grupuri sunt construite pe baza relațiilor dintre entități. De fapt, un agregat este un asemenea grup. Astfel, luând în considerare grupul de entități ca un tot unitar, putem garanta că modificările pe care le aducem sunt conforme.

De exemplu, presupunând că dorim modificarea unei programări pentru o anumită zi, trebuie să luăm în calcul dacă aparatul medical este liberă pentru a fi utilizată

și, de asemenea, dacă personalul de care avem nevoie (medic, asistent, anestezist) este disponibil pentru ora selectată. Toate aceste entități fac parte din agregatul *Programare* și trebuie să își întărească (adică starea lor în cadrul sistemului se va modifica) în momentul în care modificăm ora. Făcând o analogie între comportamentul descris și bazele de date relationale, modificarea datelor în cadrul unui agregat se face doar printr-o tranzacție în care sunt implicate toate entitățile și entitățile simple din cadrul agregatului.

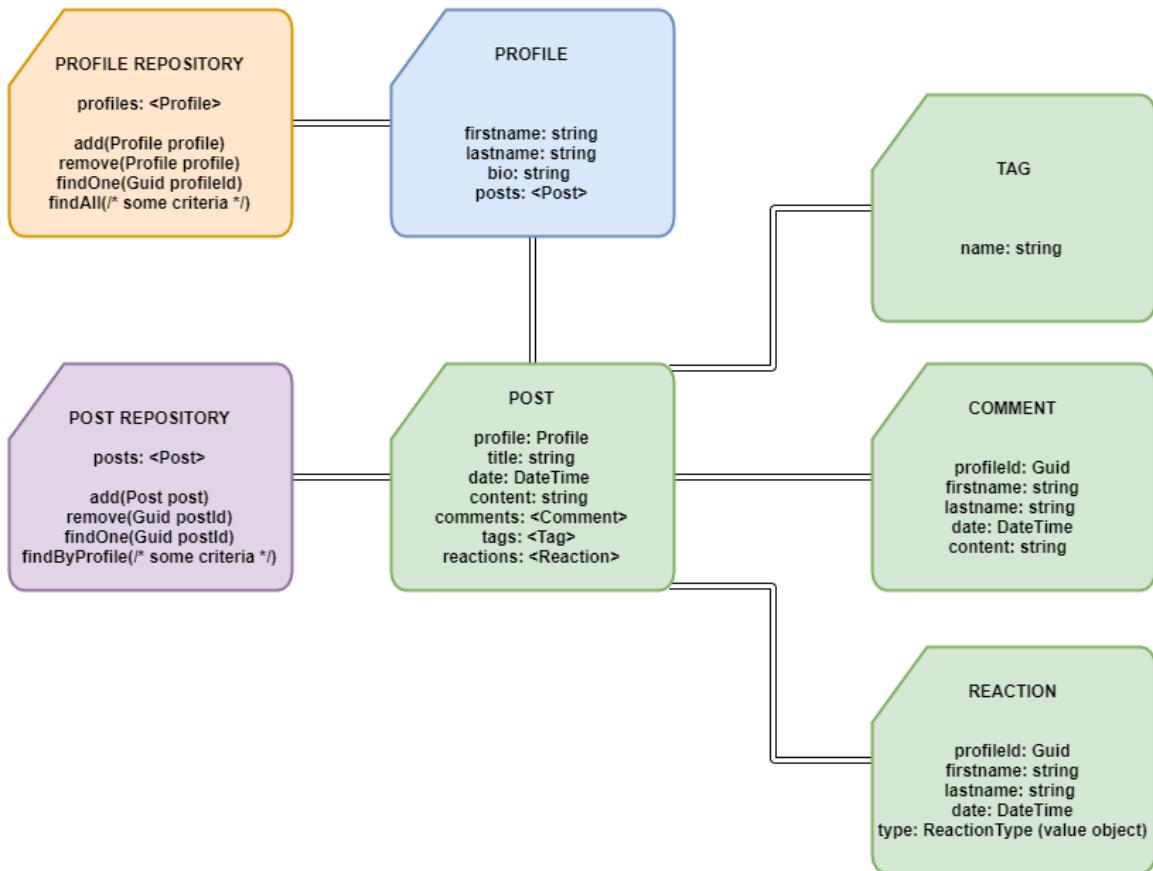


Figura 2.12: Exemplu Agregat - Post

În figura anterioară am prezentat un agregat pentru o postare din cadrul unei rețele simple de socializare și am descris relațiile dintre o postare și elementele constitutive ale acesteia: taguri, comentarii, reacții. Rădăcina agregatului este reprezentată de postare. În cadrul acesteia putem regăsi o listă de taguri, o listă de comentarii și o listă de reacții. Comentariile mențin o referință externă către profil, precum și numele și prenumele pentru afișare. Reacțiile sunt definite asemenea comentariilor, având în lipsă conținutul și în adaos o proprietate care să specifice ce tip de reacție este (*like*, *love*, *angry*, etc.), modelată ca o entitate simplă (*value object*).

2.2.4 Factories (Fabrici)

Fabricile sunt utilizate în momentul în care crearea unui anumit obiect sau a unui agregat devine un proces complicat, de lungă durată, sau în cazul în care nu dorim să expunem structura internă a acestora.



Figura 2.13: Interacțiune *Client-Factory*

Exact ca în realitate, aceste fabrici primesc toate componentele necesare (parametri) pentru creare (initializare). Eric Evans menționa [8] următoarele două cerințe pe care trebuie să le îndeplinească o fabrică:

- O fabrică trebuie să producă obiecte/aggregate care să se afle într-o stare consistentă. De exemplu, dacă o astfel de fabrică din cadrul produsului software produce un agregat care nu îndeplinește anumite cerințe din punct de vedere *business*, atunci ajungem într-o situație pe care nu ne-am dorit-o și în care suntem nevoiți să ajustăm starea agregatului pentru a îndeplini cerințele curente. Pentru o entitate, o fabrică ar trebui să producă întregul agregatul asociat împreună cu toate atributurile necesare pentru validitatea acestuia în cadrul proceselor interne. Pentru o entitate simplă (*value object*), având în vedere că aceasta este neschimbătoare, toate atributurile trebuie să fie inițializate la starea lor finală.

```

1 public interface IPetFactory
2 {
3     // Consumatorii sunt nevoiți să refere acest tip de date
4     // Astfel, ne bazăm pe o abstractizare în locul unui tip de date concret
5     IPet Create(/* parametri */);
6 }
7
8 public class PetFactory : IPetFactory
9 {
10     public IPet Create()
11     {
12         return new Pet(/* parametri */);
13     }
14 }
  
```

Extras 12: Exemplu Fabrică (Factory)

- Este recomandat ca o fabrică să returneze un tip abstract de date pentru a facilita integrarea cu diferiți consumatori.

2.2.5 Repositories (Depozite de Date)

În cadrul unui domeniu regăsim multiple asocieri între concepte care complică implementarea funcționalităților deoarece acestea trebuie descrise corect a priori. O asociere *one-to-many* se poate traduce prin prezența unei liste, dar acest lucru nu este întotdeauna posibil și nu reflectă întotdeauna realitatea. De asemenea, asocierile *many-to-many* sunt deseori bidirectionale, adăugând un nivel de complexitate sporit produsului software. Eric Evans [8] ne îndeamnă să încercăm încă din momentul designului să definim direcția asocierilor și să renunțăm la bidirectionalitate acolo unde este posibil. De asemenea, o privire de ansamblu a domeniului s-ar putea să ne ajute în eliminarea multor asocieri care nu sunt critice pentru produsul software pe care îl dezvoltăm.

Cu ajutorul asocierilor avem posibilitatea de a găsi entitățile (simple sau nu) odată ce o știm pe una dintre ele. Pentru a o găsi pe cea de bază suntem nevoiți să utilizăm un depozit de date. Acestea descriu operațiile de creare, modificare, citire și stergere pentru aggregate.

Eric Evans [8] ne îndeamnă ca pentru fiecare agregat să creăm un depozit de date care să simuleze o colecție a tuturor obiectelor care au același tip de date cu rădăcina agregatului.

Având în vedere că depozitele de date sunt utilizate pentru definirea logicii din domeniu, acestea trebuie adaptate la cerințele experților. De exemplu, căutarea unei mașini în parcul nostru auto se poate face doar după numărul de înmatriculare sau după seria motorului (pentru mașinile neînmatriculate). Aceste două posibilități de căutare se pot transforma în două metode sau într-una singură care să accepte criterii diferite de selecție.

2.2.6 Services (Servicii)

Majoritatea operațiunilor din interiorul domeniului sunt strâns legate de o anumită entitate simplă sau de o entitate. Câteodată avem de-a face cu operațiuni care nu fac parte din definiția vreunei componente, dar care au o anumită însemnatate în cadrul domeniului. Astfel, suntem nevoiți să le integrăm în modelul nostru fără a polua logica

currentă din cadrul entităților. Pentru a realiza acest lucru avem posibilitatea de a crea un serviciu independent, a cărui acțiuni/operațiuni lucrează exclusiv cu entitățile din domeniu. Acesta nu are o stare internă, în general fiind utilizat doar pentru execuție.

Eric Evans [8] afirma că un serviciu ar trebui să aibă trei caracteristici:

- Operațiunile pe care le descrie sunt în strânsă legătură cu un concept prezent în domeniu, dar care nu este inclus într-o entitate sau într-o entitate simplă (*value object*).
- Interfața acestui serviciu operează cu concepte/elemente din cadrul domeniului.
- Operațiunile pe care le descrie nu trebuie să mențină/lucreze cu o stare internă, ci din contră, trebuie să utilizeze starea curentă/globală a sistemului.

Aceste servicii pot apărea în cadrul domeniului și în urma integrării produsului software cu alte platforme/produse software. În acel moment trebuie să descriem clar conceptele și entitățile cu care un anumit serviciu lucrează în cadrul domeniului, iar acestea, precum și numele serviciului trebuie să facă parte din limbajul omniprezent.

2.2.7 Domain Events (Evenimente specifice Domeniului)

Entitățile din domeniu sunt responsabile de modificarea stării lor interne (în limitele impuse de specificații), precum și de menținea acesteia. Este aproape imposibil să aflăm care a fost sirul evenimentelor care au dus o anumită entitate în starea curentă (cea pe care o putem observa).

Menținerea tuturor modificărilor sub formă de istoric nu are aplicabilitate din punctul de vedere al experților în domeniu. Aceștia sunt interesați doar de anumite evenimente care sunt reprezentative din punct de vedere *business* sau care au dus la modificarea unor informații importante (de exemplu: modificarea stării unei comenzi). De exemplu, în cadrul unui produs software care se ocupă de logistică, experții în domeniu sunt interesați de evenimente precum: încărcarea mărfii, începerea cursei, împărțirea mărfii într-un anumit punct (depozit) pentru a fi livrată în două puncte diferite, deschiderea mărfii, finalizarea cursei. Un istoric în care cantitățile de marfă fluctuează deoarece acestea au fost încărcate/deschideate în diferite camioane nu este folositor.

Eric Evans [8] menționa că evenimentele specifice domeniului sunt o parte componentă a domeniului, fiind reprezentative pentru o anumită acțiune/operațiune care s-a întâmplat deja și care a avut ca actori principali entități din domeniu.

Un eveniment specific domeniului nu se modifică în timp, păstrând toate informațiile pe care le-a înregistrat, momentul în care s-a petrecut, precum și informații despre persoana care a declanșat acțiunea/operățiunea care a dus la înregistrarea evenimentului în cauză.

2.2.8 Implementare

Utilizând următoarea legendă pentru identificarea comenziilor și a evenimentelor care sunt folosite pentru stabilirea relațiilor de comunicare între microservicii, putem identifica agregatele și felul în care acestea se construiesc și se modifică în timp:

- CreateUserCommand. Comandă utilizată pentru înrolarea unui utilizator nou în cadrul produsului software (indiferent dacă este doctor sau deținător de animale de companie).
- UserCreatedEvent. Eveniment utilizat pentru informarea tuturor microserviciilor interesate de înrolarea unui nou utilizator.
- CreatePetCommand. Comandă utilizată pentru adăugarea unui animal de companie nou.
- PetCreatedEvent. Eveniment utilizat pentru informarea tuturor microserviciilor interesate de adăugarea unui nou animal de companie.
- CalendarCreatedEvent. În momentul înrolării unui doctor se emite evenimentul UserCreatedEvent. Microserviciul dedicat auditării calendarelor ascultă emiterea unui astfel de eveniment și creează automat un calendar de lucru utilizând orele în care clinica este deschisă. Mai apoi, publică un evenimentul CalendarCreatedEvent pentru a semnala sfârșitul procesului de înrolare.
- RequestAppointmentCommand. Comandă utilizată pentru verificarea disponibilității unei consultații în cadrul calendarului unui anumit doctor.
- AppointmentDisponibilityConfirmedEvent. Eveniment utilizat pentru informarea tuturor microserviciilor interesate de disponibilitatea creării unei consultații la ora prestabilită.

- AppointmentCreatedEvent. Eveniment utilizat pentru informarea tuturor micro-serviciilor interesate de faptul că o anumită consultație a fost aprobată (momentan aprobarea este automată în cazul în care ora prestabilită este disponibilă).
- FinalizeAppointmentCommand. Comandă utilizată pentru finalizarea consultării unui animal de companie și stocarea informațiilor referitoare la tratament/prescripții.
- AppointmentFinalizedEvent. Eveniment utilizat pentru informarea tuturor micro-serviciilor interesate de faptul că o consultăție a fost finalizată cu succes.

În următoarele imagini avem reprezentat canalul de comunicare sub forma unui *bus*. Săgețile reprezintă evenimentele sau comenzi de care sunt interesate serviciile pentru a-și actualiza starea internă (citire - actualizarea stării pentru afișare, scriere - actualizarea stării pentru consistență eventualelor verificări și validări). Liniile duble reprezintă evenimentele publicate de servicii pentru a înștiința alte servicii de modificările de care acestea ar putea fi interesate. Romburile galbene reprezintă granițele pe care le au aggregatele în cadrul structurii curente. Este posibil ca pe viitor un anumit serviciu să aibă mai multe aggregate, dar cu siguranță acestea vor face parte din contextul delimitat de granițele curente.

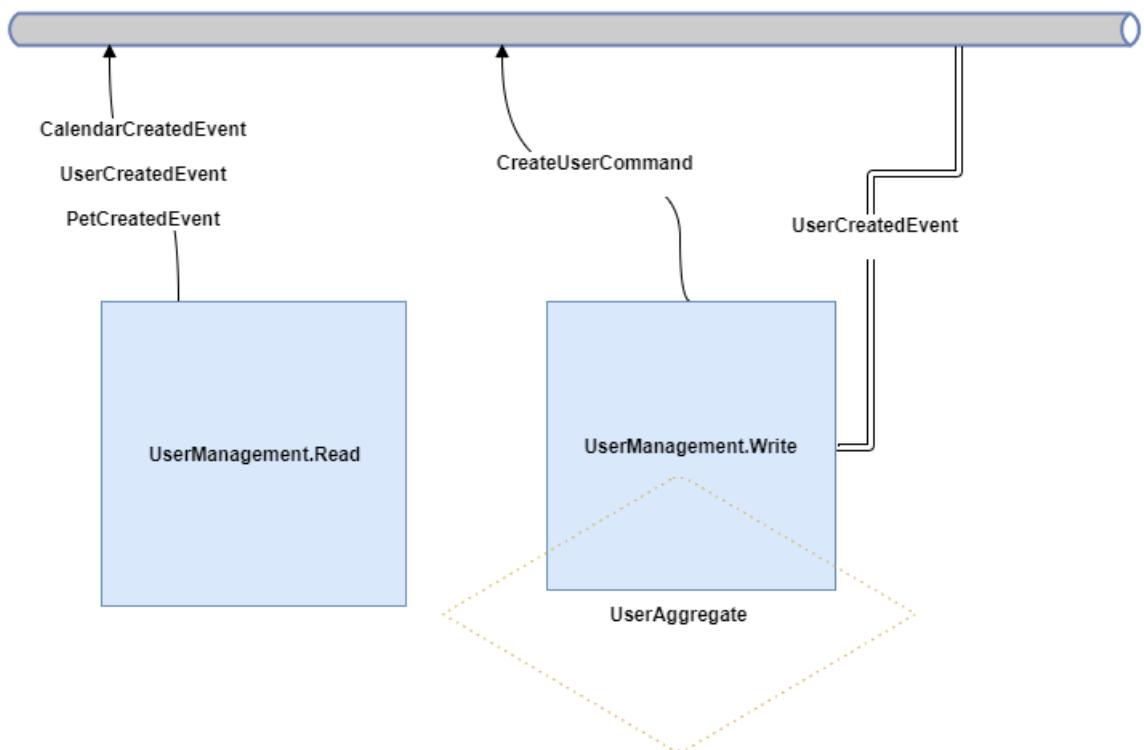


Figura 2.14: UserAggregate - Microserviciile UserManagement - Interacțiune

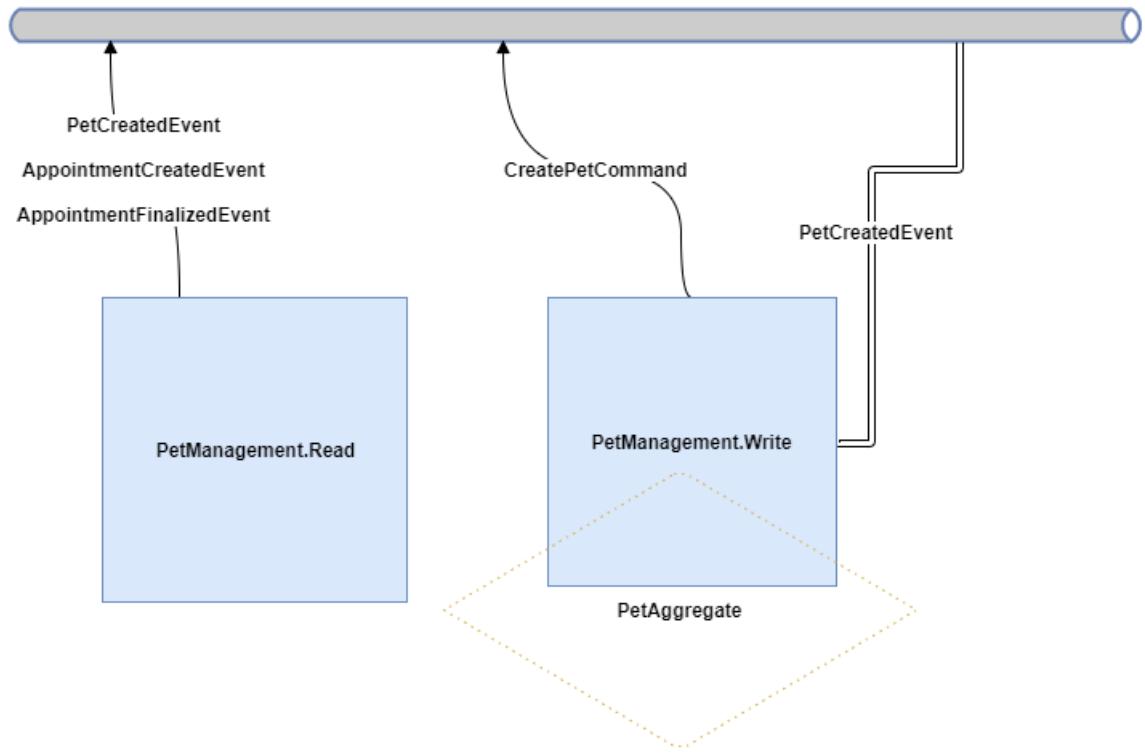


Figura 2.15: PetAggregate - Microserviciile PetManagement - Interacțiune

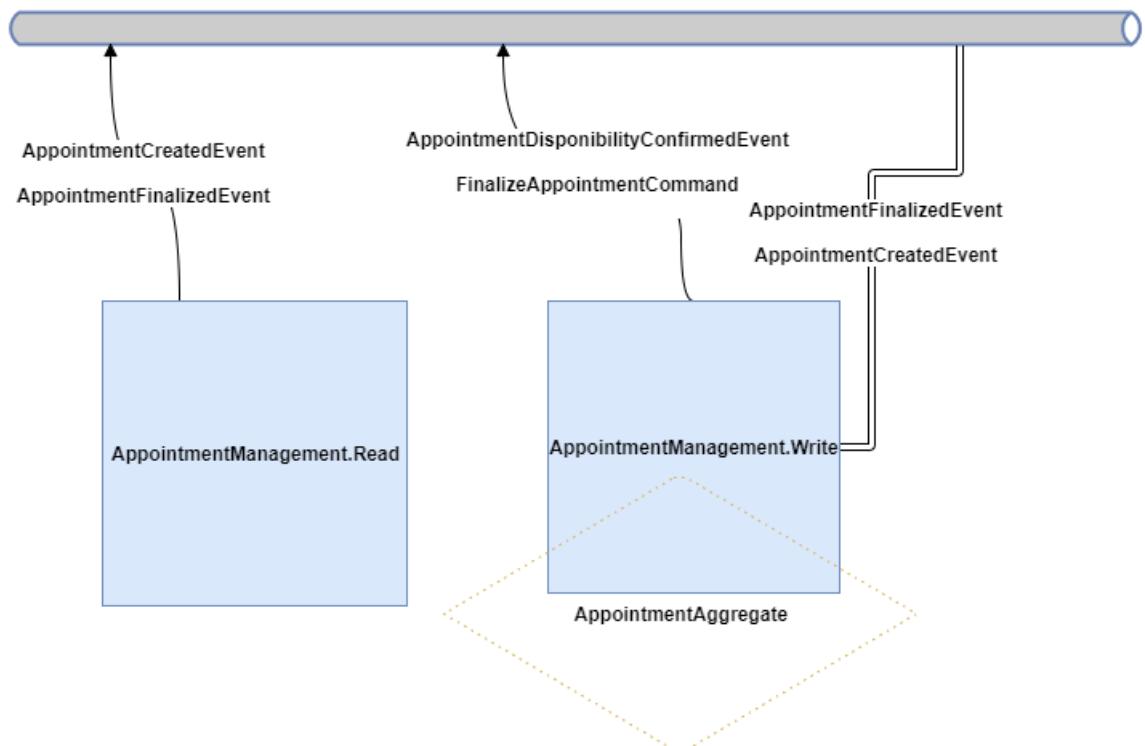


Figura 2.16: AppointmentAggregate - Microserviciile AppointmentManagement - Interacțiune

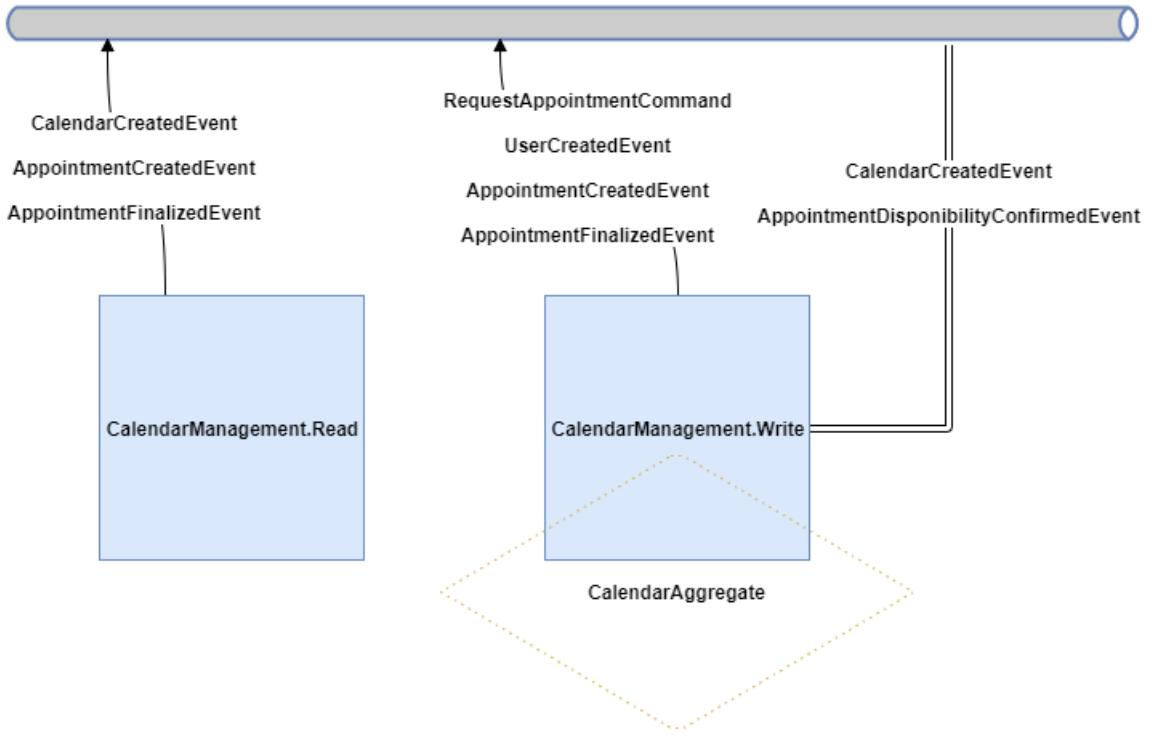


Figura 2.17: CalendarAggregate - Microserviciile CalendarManagement - Interacțiune

2.3 Event Sourcing

Dino Esposito afirma în cadrul cursului [6] că şablonul *Event Sourcing* presupune ca toate modificările aduse stării sistemului pe întreaga durată de viaţă a acestuia să fie stocate ca o serie de evenimente.

Evenimentele sunt fapte, acțiuni, întâmplări din trecut a căror natură și a căror cauze nu mai pot fi modificate. În software, asemenea evenimente sunt prezente și în aplicațiile care nu utilizează şablonul arhitectural *Event Sourcing*. De obicei, le putem observa în cadrul entităților/modelelor care necesită o oarecare formă de auditare a datelor (de exemplu: conturi bancare, comenzi în cadrul unui magazin virtual).

În general, majoritatea arhitecturilor pe care le-am întâlnit, foloseau evenimente doar pentru marcarea temporală a anumitor modificări de care experții din domeniu erau interesați. Asta se datorează faptului că în majoritatea acestor arhitecturi există un model principal care era creat pe baza a mai multor surse de date (interne: tabelele unei baze de date relationale, externe: API-uri, servicii).

Event Sourcing nu este un şablon arhitectural atât de des întâlnit, arhitecții fiind încă sceptici în adoptarea acestor practici, iar oamenii de afaceri fiind încă speriați

de lipsa unei baze de date relaționale în care să existe starea curentă a sistemului. Din punctul meu de vedere, majoritatea acestor mituri pot fi dărmate cu o cunoaștere foarte bună a domeniului, și implicit a practicilor care pot fi adoptate fără a modifica interacțiunea cu utilizatorul final. Majoritatea sistemelor care adoptă *Event Sourcing*, adoptând în același timp și un agent de intermediere a mesajelor, devin complexe și greu de explicat în termeni non-tehnici. De exemplu, consistența eventuală care vine la pachet cu aceste practici este, de obicei, cea care face ca oamenii de afaceri să refuze acest şablon arhitectural. Dar, după cum afirma Jimmy Bogard, consistența eventuală intervine doar atunci când un actor interacționează cu mai mulți actori, și implicit cu mai multe servicii pe care aceștia le oferă, acest fapt fiind unul acceptat în viața de zi cu zi, dar cumva de neacceptat în software.

Majoritatea dezvoltatorilor au întâlnit și au utilizat deja produse software care să fie bazate pe acest şablon arhitectural deși, cel mai probabil, nu au realizat asta. Git, Mercurial, TortoiseSVN sunt produse software utilizate pentru versionarea codului. De exemplu, în cazul Git, starea curentă este reprezentată de aplicarea tuturor modificărilor (*commit*) asupra versiunii inițiale.

În cadrul unui produs software care utilizează strategia DDD, după împărțirea domeniului folosind şabloane precum *Context Map*, putem observa că evenimentele de care sunt interesați experții din domeniu sunt, de fapt, toate evenimentele care se întâmplă în cadrul și în jurul agregatelor pe care le-am identificat. Deci, pentru a aplica şablonul arhitectural *Event Sourcing* suntem nevoiți să urmărim și să stocăm toate aceste evenimente, utilizând aggregatul cu care au o strânsă legătură pe post de nucleu.

De fapt, acest proces presupune trecerea de la persistarea modelelor de date într-o formă cât mai convenabilă pentru a fi servite către clienți la înregistrarea evenimentelor care s-au produs și la reconstrucția modelelor de date pe baza acestor evenimente. Acestea se pot înregistra utilizând una dintre următoarele abordări:

- Crearea unuia sau mai multor tabele în cadrul unei baze de date relaționale care să simuleze un *event store*. În acest caz trebuie să amintim faptul că, pe lângă că nu pot fi modificate, evenimentele nu pot fi șterse odată ce s-au întâmplat. Ștergerea unui eveniment din trecut presupune alterarea stării sistemului începând cu data ulterioară întâmplării evenimentului în cauză. Se poate accepta ștergerea celor mai noi x evenimente în încercarea de a implementa acțiunea de *Undo*.

- Utilizarea unei soluții precum cea amintită în cadrul subcapitolului 3.1.7. (*Event-Store*). Majoritatea acestor soluții au suport nativ (metode de extensie, adjuvante) pentru proiectarea acestor evenimente sub forma unor modele de date, utilizate mai apoi de partea responsabilă de citire.

Deși sunt deseori asociate, şabloanele CQRS și *Event Sourcing* sunt independente și pot exista unul fără celălalt. Utilizându-le împreună, partea responsabilă de scrierea datelor trebuie să suporte, de fapt, stocarea evenimentelor, iar cea responsabilă de citire poate să utilizeze proiecții ale evenimentelor stocate pentru a putea descrie starea curentă a sistemului.

Pentru a înțelege mai bine dacă produsul pe care îl dezvoltăm necesită sau nu acest şablon arhitectural, putem să cercetăm specificațiile și să fim atenți după formulări de tipul: "Este important să știm data exactă când un produs a fost adăugat (într-un coș de cumpărături, într-un camion, etc)", "Este important să știm dacă adăugarea unui produs a necesitat scoaterea altor produse și readăugarea altora (de exemplu: ajustarea costurilor asociate coșului de cumpărături, renunțarea la anumite produse pentru a menține costurile sub o anumită sumă)".

În cadrul conferinței YOW! Nights din martie 2016, Martin Fowler a prezentat o serie de motive pentru care şablonul arhitectural *Event Sourcing* merită revizuit și, de ce nu, adoptat:

- *Debugging*. Având în vedere că toate evenimentele care s-au produs până la un anumit moment dat sunt stocate în ordinea apariției lor, avem posibilitatea de a crea un mediu care să simuleze mediul de producție la o anumită dată în trecut. De exemplu, să presupunem că un utilizator al unei bănci și-a setat ca toate plățile recurente să fie făcute în data de opt a fiecărei luni. Timp de patruzeci de luni aceste plăți recurente au funcționat perfect, fără a avea vreo eroare. În ultimele două luni câteva plăți au generat o eroare. Reconstruind mediul de producție din evenimentele stocate, putem observa, de fapt, că acea eroare a fost influențată de modificarea limitei de plată pe care o setase proprietarul contului și de adăugarea unor plăți recurente noi.
- *Audit Trail*. Toate evenimentele sunt stocate într-un anume fel având posibilitatea de a identifica contextul în care acestea s-au produs (dată, versiune, etc.). Utilizând o bază de date relatională, tabelele în care am putea să stocăm evenimentele care s-au produs pot avea următoarea structură:

```

1 CREATE TABLE EventSources (
2     Id [uniqueidentifier] NOT NULL,
3     Type [nvarchar](255) NOT NULL,
4     Version [int] NOT NULL
5 );
6
7 CREATE TABLE Events (
8     Id [uniqueidentifier] NOT NULL,
9     TimeStamp [datetime] NOT NULL,
10    Name [varchar](max) NOT NULL,
11    Version [varchar](max) NOT NULL,
12    EventSourceId [uniqueidentifier] NOT NULL,
13    Sequence [bigint],
14    Data [nvarchar](max) NOT NULL
15 );
16
17 ALTER TABLE Events
18 ADD FOREIGN KEY (EventSourceId) REFERENCES EventSources(Id);

```

Extras 13: Exemplu structură tabele evenimente

- *Historic State.* Posibilitatea de a accesa starea sistemului la o anumită dată plecând de la starea inițială a sistemului și aplicând toate evenimentele stocate până la data în cauză.
- *CQRS/Variant schemas.* Având în vedere că toate evenimentele care s-au produs până la un anumit moment dat sunt stocate în ordinea apariției lor, avem posibilitatea de a proiecta aceste evenimente sub forma unor modele utilizate pentru diferite interogări. Această separare între stocarea evenimentelor produse și crearea/integrarea diferitelor scheme de interogare reprezintă suportul necesar și suficient pentru adoptarea şablonului CQRS. Adoptarea unui astfel de comportament duce în același timp la acceptarea consistenței eventuale deoarece propagarea evenimentelor către diferitele scheme de interogare creează inconsistență între acestea. Termenul *Variant schemas* se referă la faptul că schemele de interogare construite pe baza evenimentelor reprezintă, de fapt, realități diferite care co-există.
- *Distribution Support.* Posibilitatea de a rezolva starea curentă a sistemului deși acesta este un sistem distribuit. Rezolvarea stării curente presupune contopirea evenimentelor produse până la un anumit moment dat. Această contopire este oarecum asemănătoare cu acțiunea *merge* din cadrul produselor software utilizate pentru versionarea codului sursă. În cazul unui sistem distribuit, utilizatorul poate fi responsabil de rezolvarea stării curente având la dispoziție

o interfață din care poate să aleagă proprietățile/acțiune pe care le aplică și proprietățile/acțiunile la care renunță în cazul apariției unui conflict. De asemenea, avem posibilitatea de a rezolva automat anumite situații conflictuale atunci când domeniul și regulile descrise de experții din domeniu ne permit acest fapt.

- *Memory Image* [12]. Având în vedere că evenimentele stocate sunt cele care descriu toate acțiunile/operațiunile care s-au întâmplat și de care suntem interesați, starea curentă a sistemului poate să fie o structură de date ținută în memorie fără a fi replicată pe disc sau într-o bază de date relațională. În caz că această stare se alterează, putem oricând să renunțăm la ea și să o recreăm pe baza evenimentelor stocate. Într-un produs software care adoptă şablonanele arhitecturale CQRS și *Event Sourcing* există deseori confuzia că starea sistemului poate fi recreată și din comenzi care au fost executate deoarece doar acele comenzi au produs modificări. Acest fapt este parțial adevărat dar necesită o muncă asiduă care nu este în scopul acestor şablonane arhitecturale, de altfel, nefiind recomandat de către specialiști. În cadrul produsului software pe care l-am dezvoltat, fiecare comandă poate produce unul sau mai multe evenimente care sunt stocate. De obicei, stocarea acestor evenimente reprezintă modificarea stării unui agregat, ceea ce duce la crearea și emiterea unui nou eveniment pentru a îmștiința serviciile care se ocupă de citirea datelor să actualizeze starea modelelor care sunt stocate în Redis.
- *Alternative Histories*. Posibilitatea de a crea o realitate alternativă prin introducerea unui eveniment la un anumit moment dat. De exemplu, aflăm că una din ratele pe care le avem de plătit pentru a achita un credit ipotecar nu a fost înregistrată corect și a produs penalizări în cascadă timp de doisprezece luni. Starea curentă a sistemului este clar alterată, cel puțin în cazul acestui credit. Presupunem că avem posibilitatea să refacem starea sistemului până la data în care trebuia să fie înregistrată rata, să o înregistram corect, iar apoi să reaplicăm evenimentele stocate de la acea dată și până în prezent. În acest caz, am avea o stare a sistemului alternativă (care în același timp este demonstrată ca fiind și cea corectă) și am avea posibilitatea de a compara starea aceasta cu cea în care rata nu a fost înregistrată corect pentru a corecta penalizările și a rezolva situația creată.

În cadrul produsului software pe care l-am dezvoltat, citirea evenimentelor pentru construirea în memorie a aggregatului se realizează în pachete de câte cinci sute de

evenimente. Dacă numărul acestora trece de o mie de evenimente, este posibil să apară anumite probleme de performanță deoarece suntem nevoiți ca de fiecare dată când dorim să aplicăm un nou eveniment să citim o mie și să le aplicăm pentru a genera starea curentă a agregatului pe care, în general, o utilizăm pentru validare.

Există diferite tehnici pentru a evita citirea și aplicarea unui număr atât de mare de evenimente în caz că produsul software necesită acest lucru. Cea mai întâlnită dintre ele este serializarea și stocarea întregii stări a agregatului într-un anumit moment pe care îl decidem în funcție de performanță dorită.

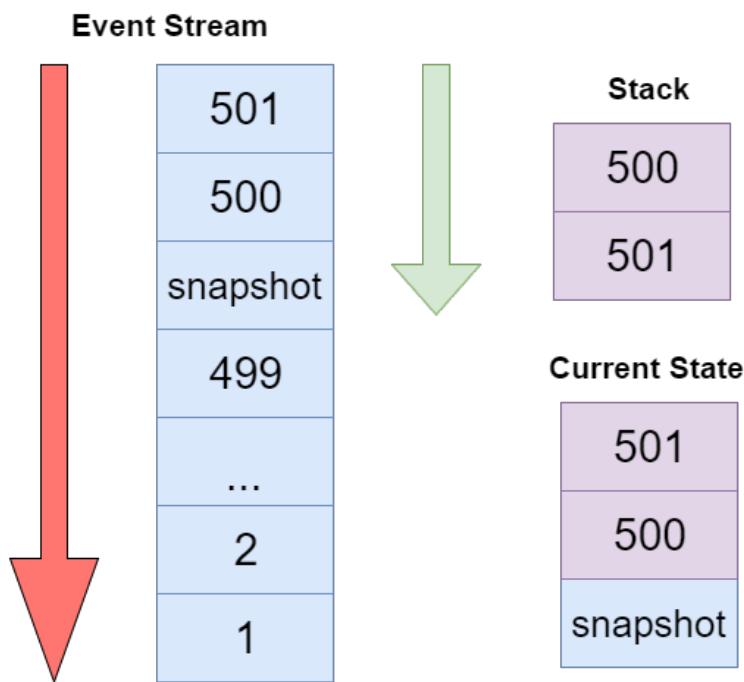


Figura 2.18: Aggregate Snapshot

Contragă practicilor discutate până în acest moment, tehnica curentă presupune citirea evenimentelor stocate începând cu ultimul care s-a produs până în momentul în care întâlnim un *snapshot*. Evenimentele citite sunt stocate într-o stivă (*Last In First Out*) și mai apoi aplicate agregatului a cărui stare o găsim stocată sau agregatului inițial în cazul în care starea acestuia nu a fost serializată până în acest moment.

```

1 public class OrderVoidedEvent_v1 : Event
2 {
3     public readonly Guid Id;
4     public readonly Guid UserId;
5
6     public OrderVoidedEvent_v1(Guid id, Guid userId)
7     {
8         Id = id;
9         UserId = userId;
10    }
11 }
12
13 public void Void(Guid userId)
14 {
15     if(_voided) throw new InvalidOperationException("Order already voided.");
16     Apply(new OrderVoidedEvent_v1(_id, userId));
17 }
18
19 private void Apply(OrderVoidedEvent_v1 e)
20 {
21     _voided = true;
22 }

```

Extras 14: OrderVoidedEvent - Inițial (presupunând că avem în vedere versionarea)

Odată cu revizuirea specificațiilor sau cu schimbarea acestora în timp există posibilitatea să se modifice structura evenimentelor. Astfel, este necesar suportul pentru mai multe versiuni ale aceluiași eveniment sau chiar suportul pentru maparea logică dintre un tip de eveniment inactual și unul care trebuie utilizat începând cu versiunea curentă. De exemplu, în cadrul unui produsului software în domeniul logistic, în momentul în care anulăm o comandă plasată în sistem avem nevoie doar de numărul comenzi și de identificatorul persoanei care a executat această acțiune pentru a putea fi trasă la răspundere. După o perioadă de timp, discutând cu persoanele care executau această acțiune, observăm că în cele mai multe cazuri clientul renunță la comandă deoarece se răzgândește sau î se comunică un timp mult prea mare de livrare. Având în vedere situația creată, pentru a nu trage la răspundere angajații și pentru a filtra cererile de anulare putem atașa un motiv pentru care comanda a fost anulată, schimbând astfel structura evenimentului.

```

1 public class OrderVoidedEvent_v2 : Event
2 {
3     public readonly Guid Id;
4     public readonly Guid UserId;
5     public readonly string Reason;
6
7     public OrderVoidedEvent_v2(Guid id, Guid userId, string reason)
8     {
9         Id = id;
10        UserId = userId;
11        Reason = reason;
12    }
13 }
14 // Modificăm metoda Void pentru a putea transmite motivul
15 public void Void(Guid userId, string reason)
16 {
17     if(_voided) throw new InvalidOperationException("Order already voided.");
18     Apply(new OrderVoidedEvent_v2(_id, userId, reason));
19 }
20 // Păstrăm și metoda veche Apply(OrderVoidedEvent_v1 e)
21 private void Apply(OrderVoidedEvent_v2 e)
22 {
23     _voided = true;
24 }

```

Extras 15: OrderVoidedEvent - versiunea a două, actualizată

2.3.1 Implementare

În acest subcapitol am exemplificat cursul evenimentelor reprezentative pentru agregatul calendar, precum și modalitatea în care acestea sunt aplicate și stocate pentru a putea fi utilizate la verificarea și validarea disponibilității unui anumit doctor. Înrolarea unui medic în cadrul produsului software pe care l-am dezvoltat presupune, de asemenea, crearea unui calendar de lucru utilizând orele în care clinica este deschisă.

Event Stream 'calendar-b7c5b8da-f76d-405b-88f4-d8b95d3dbec2'

[Pause](#) [Edit ACL](#) [Back](#)

[self](#) [first](#) [previous](#) [metadata](#)

Event #	Name	Type	Created Date	
6	6@calendar-b7c5b8da-f76d-405b-88f4-d8b95d3dbec2	CalendarWorkingHoursSpecifiedEvent	2017-09-09 16:24:50	JSON
5	5@calendar-b7c5b8da-f76d-405b-88f4-d8b95d3dbec2	CalendarWorkingHoursSpecifiedEvent	2017-09-09 16:24:50	JSON
4	4@calendar-b7c5b8da-f76d-405b-88f4-d8b95d3dbec2	CalendarWorkingHoursSpecifiedEvent	2017-09-09 16:24:50	JSON
3	3@calendar-b7c5b8da-f76d-405b-88f4-d8b95d3dbec2	CalendarWorkingHoursSpecifiedEvent	2017-09-09 16:24:50	JSON
2	2@calendar-b7c5b8da-f76d-405b-88f4-d8b95d3dbec2	CalendarWorkingHoursSpecifiedEvent	2017-09-09 16:24:50	JSON
1	1@calendar-b7c5b8da-f76d-405b-88f4-d8b95d3dbec2	CalendarWorkingHoursSpecifiedEvent	2017-09-09 16:24:50	JSON
0	0@calendar-b7c5b8da-f76d-405b-88f4-d8b95d3dbec2	CalendarCreatedEvent	2017-09-09 16:24:50	JSON

Figura 2.19: Evenimente - Inițializarea calendarului unui medic

CalendarCreatedEvent este utilizat pentru a specifica initializarea unui calendar pentru un anumit medic, iar evenimentele CalendarWorkingHoursSpecifiedEvent sunt utilizate pentru a-l face pe acesta disponibil în cadrul orelor în care clinica este deschisă (L-V 9-17 și Sâ 9-12). Momentan nu există posibilitatea de a configura disponibilitatea unui doctor, dar această funcționalitate presupune adăugarea interfeței și maparea comenzi de configurare la unul sau mai multe evenimente de tipul CalendarWorkingHoursSpecifiedEvent.

```
5@calendar-b7c5b8da-f76d-405b-88f4-d8b95d3dbec2
next prev
No Stream Type
5 calendar-b7c5b8da-f76d-405b-88f4-d8b95d3dbec2 CalendarWorkingHoursSpecifiedEvent
Data
{
  "Day": {
    "Name": "Friday"
  },
  "WorkingHours": {
    "Start": 9,
    "End": 17
  }
}

Metadata
{
  "CommitId": "f2f18671-1c32-492b-aee2-bdfbb7adfc4cd",
  "AggregateClrTypeName": "WanVet.Micro.CalendarManagement.Write.Domain.Model.CalendarModel.Calendar, WanVet.Micro.CalendarManagement.Write, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null",
  "ServerClock": "2017-09-09T13:24:49.9311651Z",
  "EventClrTypeName": "WanVet.Micro.CalendarManagement.Write.Domain.Model.CalendarModel.Events.CalendarWorkingHoursSpecifiedEvent, WanVet.Micro.CalendarManagement.Write, Version=1.0.0.0"
}
```

Figura 2.20: Evenimente - Exemplu de ajustare a calendarului unui medic

Replicarea datelor referitoare la calendar este realizată pe baza unui dicționar Dictionary<string, Tuple<int, int>> care presupune stocare în Redis a denumirii zilei săptămânii precum și a orelor de început și de final de program.

The screenshot shows a Redis interface with two sections. The top section displays a key named 'b7c5b8da-f76d-405b-88f4-d8b95d3dbec2' with a value size of 36 bytes. The bottom section displays a value named 'calendar' with a size of 344 bytes, containing a JSON object representing a calendar entry:

```
{"Id": "b7c5b8da-f76d-405b-88f4-d8b95d3dbec2", "UserId": "b1dc9293-1ea0-43af-9ea6-57b4bf6f6104", "Version": 6, "WorkingHours": {"Monday": {"Item1": 9, "Item2": 17}, "Tuesday": {"Item1": 9, "Item2": 17}, "Wednesday": {"Item1": 9, "Item2": 17}, "Thursday": {"Item1": 9, "Item2": 17}, "Friday": {"Item1": 9, "Item2": 17}, "Saturday": {"Item1": 9, "Item2": 12}}, "RedisKey": "calendar"}
```

Figura 2.21: Redis - Stocarea datelor referitoare la calendar pentru citire

Adăugarea unei noi consultații generează un eveniment de tipul AppointmentCreatedEvent care, aplicat cursului curent de evenimente modifică lista Appointments. Păstrând informații despre consultațiile curente în cadrul agregatului calendar avem

posibilitatea de a verifica și valida disponibilitatea pentru o anumită dată/oră.

No	Stream	Type
7	calendar-b7c5b8da-f76d-405b-88f4-d8b95d3dbec2	AppointmentCreatedEvent
Data		
{ "Id": "aeaca249d-dfcc-467a-a2eb-46b194c86248", "PetId": "7e427544-e9a7-4b23-aad9-fd470adf3d1e", "PetName": "Martin", "OwnerId": "c8cb4e2f-02fb-47af-87a7-a6cf2c2e1476", "OwnerFamilyName": "Barbu", "OwnerGivenName": "Ionut", "StartingTime": "2017-09-23T10:30:00+03:00", "State": { "Name": "Open" } }		
Metadata		
{ "CommitId": "97ae1b66-c7f1-4a0f-a8b4-f831fce24988", "AggregateClrTypeName": "WanVet.Micro.CalendarManagement.Write.Domain.Model.CalendarModel.Calendar", WanVet.Micro.CalendarManagement.Write. "ServerClock": "2017-09-10T12:56:35.9519103Z", "EventClrTypeName": "WanVet.Micro.CalendarManagement.Write.Domain.Model.CalendarModel.Events.AppointmentCreatedEvent", WanVet.Micro.Calenc }		

Figura 2.22: Evenimente - Exemplu de adăugare a unei consultații - calendarul unui medic

```

1 calendar.AddAppointment(new AppointmentCreatedEvent(@event.AggregateId,
2                         @event.PetId, @event.PetName, @event.OwnerId, @event.OwnerFamilyName,
3                         @event.OwnerGivenName, @event.StartingTime, @event.State));
4
5 public class Calendar : Aggregate
6 {
7     // Nu avem posibilitatea de a modifica aggregatul din extern
8     public void AddAppointment(AppointmentCreatedEvent @event){
9         RaiseEvent(@event);
10    }
11
12    public void Apply(AppointmentCreatedEvent @event)
13    {
14        Appointments.Add(new Appointment
15        {
16            Id = @event.Id,
17            OwnerFamilyName = @event.OwnerFamilyName,
18            OwnerGivenName = @event.OwnerGivenName,
19            OwnerId = @event.OwnerId,
20            PetId = @event.PetId,
21            PetName = @event.PetName,
22            StartingTime = @event.StartingTime,
23            State = @event.State
24        });
25    }
26 }
```

Extras 16: Aplicarea evenimentului AppointmentCreatedEvent

De asemenea, datele se replică și în Redis pentru menținerea stării curente a produsului software.

```

Key: size in bytes: 36
Value: size in bytes: 1169
[{"Owner": "c8cb4e2f-02fb-47af-87a7-a6cf2c2e1476", "OwnerFamilyName": "Barbu", "OwnerGivenName": "Ionut", "StartingTime": "2017-09-22T15:00:00+03:00", "State": "Open"}, {"Id": "aea249d-dfcc-467a-a2eb-46b194c86248", "PetId": "7e427544-e9a7-4b23-aad9-fd470adf3de", "PetName": "Martin", "OwnerId": "c8cb4e2f-02fb-47af-87a7-a6cf2c2e1476", "OwnerFamilyName": "Barbu", "OwnerGivenName": "Ionut", "StartingTime": "2017-09-23T10:30:00+03:00", "State": "Open"}], "RedisKey": "calendar"
}

```

Figura 2.23: Redis - Stocarea datelor referitoare la calendar după actualizare

2.4 Microservicii

Pentru a clarifica termenul încă de la început, aş dori să menţionez că microserviciile sunt, de fapt, servicii de mici dimensiuni care rezolvă o problemă punctuală sau au în grijă o zonă bine delimitată a unui produs software.

Nu există o definiție clară a acestui concept, dar poate fi definit prin comparație cu problema pe care încearcă să o rezolve și anume produsele software care au o arhitectură de tip monolit.

Din punct de vedere arhitectural, adoptarea acestei metode implică dezvoltarea mai multor servicii independente, fiecare în parte având un scop prestativ, acestea rezolvând o singură problemă de ansamblu. În general, singura cale de comunicare între servicii este cea bazată pe evenimente, acestea neavând posibilitatea de a comunica direct.

Din descrierile anterioare putem să deducem că arhitectura amintită este asemănătoare cu SOA (*Service-Oriented Architecture*), din această cauză fiind supranumita SOA w/ DevOps, SOA pentru hipsteri sau SOA 2.0. *Service-Oriented Architecture* presupune existența unui strat de servicii care operează asupra unei anumite entități și a tuturor relațiilor pe care aceasta le are în cadrul produsului software.

Din punct de vedere al dezvoltării, microserviciile ar trebui să fie agnoscibile de platforma utilizată, de limbajul de programare și de sistemul de operare pe care acestea rulează. Cerința absolut necesară ca această arhitectură să funcționeze în parametri

normali este posibilitatea unui serviciu de a comunica cu celealte servicii, nu într-un mod direct (*service locator*), ci prin utilizarea unui agent de intermediere a mesajelor.

Arhitectură bazată pe microservicii	Arhitectură monolit
Complexitate crescută a dezvoltării deoarece trebuie să fie delimitate clar (<i>business-wise</i>) zonele pe care le poate atinge (modifica/restructura) un anumit serviciu	Simplitate (codul face parte dintr-o singură soluție și modelele pot fi refolosite ușor)
Adoptarea microserviciilor presupune tempi de așteptare până când tot actorii interesați sunt înștiințați de eventualele modificări (în general comunicarea între servicii se realizează cu ajutorul unui agent de intermediere a mesajelor)	Consistență (în general comunicarea între servicii se realizează instantaneu și datele sunt afișate în timp real)
Modularitatea este păstrată, dar responsabilitatea pentru un anumit serviciu este a echipei care a lucrat la acel serviciu (în general, în organizațiile mari, o echipă este responsabilă pentru un singur serviciu și, de asemenea, este responsabilă de comunicarea datelor cărora le este proprietar către alte servicii)	Intermodularitate (toate echipele pot ajusta același cod, astfel există și posibilitatea de a se ajuta reciproc, dar și posibilitatea de a duplica modelele de date)
Produsul software poate referi multiple servicii care pot fi dezvoltate pe mai multe platforme, în multiple limbi de programare și care pot să utilizeze baze de date diferite (SQL, NoSQL)	Produsul software este dependent în totalitate de platformă și limbaj, iar acest lucru poate genera alte dependențe (baza de date - de exemplu, până în 2017, Microsoft nu oferea posibilitatea de a rula SQL Server pe Linux)

Cu siguranță că utilizarea aceluiasi limbaj de programare sau aceluiasi sistem de operare poate să scadă riscul și costurile asociate dezvoltării, dar aceste decizii pot fi influențate de rapiditatea cu care se poate dezvolta un serviciu într-un anume limbaj.

De exemplu, putem presupune că dorim automatizarea departamentului de marketing dintr-o anumită organizație. Dezvoltarea unui serviciu care să simuleze un anumit comportament dat pe o rețea de socializare (Instagram) este facilă într-un limbaj precum Python și poate fi o cerință costisitoare într-un alt limbaj.

Ritmul de dezvoltare al unui produs software care se bazează pe o arhitectură monolit este unul ridicat. În general, acest fapt este facilitat de reutilizarea modelelor de date, a modalităților de stocare, și a operațiunilor (proceselor) interne. Bazându-ne pe ideile acumulate din studierea strategiei DDD, putem afirma că toate aceste concepte fac parte din contexte diferite, dar sunt utilizate pretutindeni.

Cu siguranță că există și produse software care au fost construite folosind strategia DDD și care au contexte bine delimitate. Pentru a ajunge la acest comportament, de obicei, se utilizează proiecte separate în cadrul aceleiași soluții (*Operations, Sales, Integrations, etc.*). În acest caz, ne asigurăm că modelele de date nu sunt reutilizate de alte proiecte, dar sursa de date este deseori aceeași. Presupunând că dezvoltăm un produs software în domeniul logisticii, putem observa că informațiile despre o anumită comandă ar trebui să ajungă în majoritatea proiectelor într-o formă sau alta. Pentru a suporta acest lucru, avem câteva opțiuni:

- Să păstrăm informațiile despre comandă într-o singură tabelă în baza de date. În apoi, utilizând proiecții să creăm modele de date reprezentative pentru fiecare proiect în parte
- Să păstrăm informațiile despre comandă în multiple tabele în baza de date reprezentative pentru fiecare proiect în parte. De exemplu, putem diferenția tabelele utilizând o schema de baze de date (*operations.order, sales.order, integrations.order*)

În primul caz, decuplarea așteptată nu este în totalitate realizată. În cel de-al doilea caz suntem nevoiți să dezvoltăm și modalități de sincronizare între informațiile care acum se află în tabele diferite, dar care reprezintă aceeași entitate.

În articolul *Microservices. A definition of this new architectural term* [13] Martin Fowler definea o componentă ca fiind o parte a unui produs software care poate fi înlocuită, versionată și actualizată independent de restul părților care compun produsul software. Aceste componente pot fi de două feluri: librării și servicii:

- Librăriile interacționează între ele în mod direct, această comunicare fiind generatoare de legături strânse precum reutilizarea modelelor de date, structurarea

asemănătoare a parametrilor utilizați, etc. Versionarea unui produs software a cărui componente sunt librării presupune reevaluarea și actualizarea întregului produs.

- Serviciile interacționează între ele prin cereri web sau prin *remote procedure call*.

În cazuri avansate, acestea sunt separate total și interacționează cu ajutorul unor agenți de intermediere a mesajelor. Versionarea unui produs software a cărui componente sunt servicii presupune reevaluarea și actualizarea fiecărui serviciu în parte. Se creează astfel posibilitatea de a lucra la un singur set de funcționalități, reprezentative pentru un singur serviciu, precum și posibilitatea de a actualiza independent componente ale produsului software.

Alan Morrison identifica trei factori de care depinde posibilitatea serviciilor de a fi complet independente [14]:

- Serviciile trebuie să de mici dimensiuni, să rezolve o problemă specifică, iar această problemă să fie complet independentă din punct de vedere logic de orice altă problemă existentă în cadrul produsului software.
- Agentul de intermediere a mesajelor nu trebuie să fie intelligent (de exemplu: Kafka, RabbitMQ). Utilizarea unui agent de intermediere a mesajelor de tip *enterprise* care să posede o logică proprie duce, de fapt, la crearea unor legături între servicii de tipul *remote procedure call* (serviciul care citește mesajul aflat din cadrul acestuia ce să facă). Pentru a fi complet independente serviciile trebuie să fie reactive adică să aibă logică proprie pentru a identifica ce operațiuni interne trebuie să execute atunci când citesc un mesaj de un anumit tip.
- Existența mai multor versiuni ale aceluiași serviciu (pentru a putea coordona lansările produsului și a avea întotdeauna un plan de rezervă).

Adoptarea unei arhitecturi bazate pe microservicii presupune și schimbări din punct de vedere organizațional. Presupunând că dezvoltăm un produs software utilizat în domeniul logisticii, serviciile fără de care acest produs nu poate să existe sunt: *Operations* (monitorizarea stării comenzi, de la creare și până în momentul în care aceasta este livrată), *Tracking* (monitorizarea flotei), *FABP* (*financial, accounting, billing, processing*). Restructurarea echipelor presupune și responsabilizarea acestora asupra serviciilor pe care le dezvoltă. Pentru a putea fi responsabile de un serviciu, echipele trebuie să

să fie formate din persoane care să acopere toate arile de dezvoltare. În urma acestui proces, este posibil să apară noi roluri precum cel de integrator, acesta fiind o punte de comunicare între servicii.

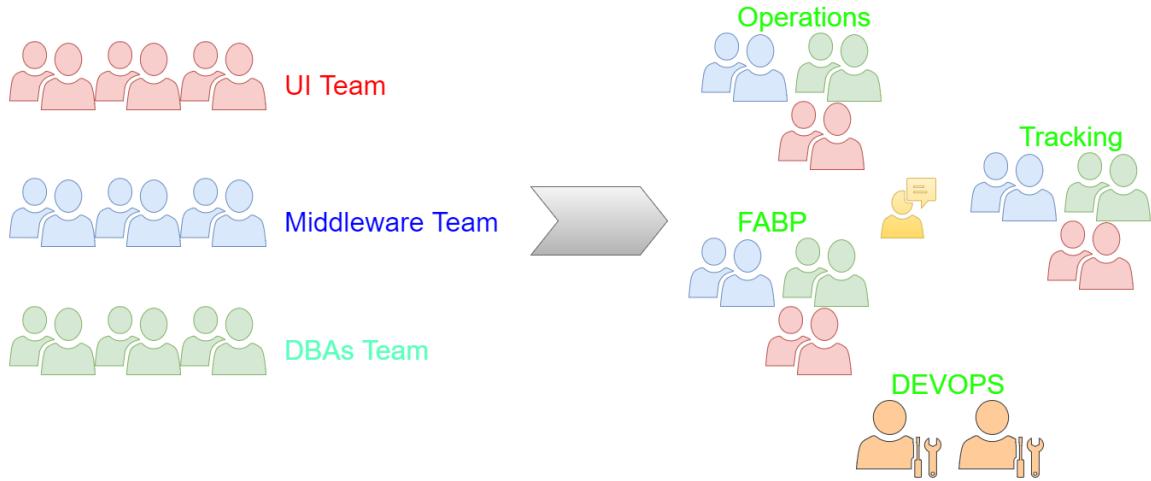


Figura 2.24: Restructurarea organizației

De asemenea, trebuie redefinită echipa care se ocupă de integrarea continuă și de lansările diferitelor servicii. Această responsabilitate poate fi a fiecărei echipe în parte pentru serviciul pe care îl dezvoltă sau se poate crea o echipă nouă (*DevOps*) care să se ocupe strict de aceste aspecte.

Există două modalități de restructurare a echipelor:

- *Decompose By Business Capability*. Presupune definirea serviciilor în funcție de anumite obiecte importante din cadrul domeniului nostru. De exemplu, în cazul unui produs software utilizat în domeniul logisticii, putem avea *Order Management*, *Customer Management*, *Carrier Management*, etc.
- *Decompose By Subdomain*. Presupune definirea serviciilor în funcție de subdomeniile identificate în urma aplicării strategiei DDD.

Pe lângă ideile arhitecturale și şabioanele menționate anterior, am putea să mai amintim și:

- *API Gateway*. Presupune implementarea unui API care să reprezinte poarta de intrare a produsului software pe care îl construim. Astfel, în cadrul API-ului putem adapta funcționalitățile expuse de servicii în funcție de client.
- *Database Per Service*. Presupune descentralizarea bazei de date utilizate în cadrul unui produs bazat pe arhitectură monolit. Astfel, fiecare serviciu este responsabil

de modalitatea în care își stochează datele și de înștiințarea celorlalte servicii de modificări de care acestea ar putea fi interesate. De exemplu, dacă suntem nevoiți să dezvoltăm posibilitatea de a posta articole, avem posibilitatea de a alege o bază de date NoSQL orientată-document precum MongoDB, CouchDB, etc.

- *Design for failure.* Presupune existența unor mecanisme de restartare a serviciilor atunci când observăm că acestea nu răspund cererilor. De asemenea, am putea redirecționa traficul care ar trebui să ajungă într-un anumit serviciu către unul asemănător pe care să-l creăm automat în momentul în care observăm că nu primim nici un răspuns. Michael Stahnke, afirma într-o postare pe Twitter în mod ironic, că orice companie dispune de un mediu de test, și doar câteva dispun și de un alt mediu în care să ruleze versiunea de producție. În cadrul organizației Netflix, direct în mediul în care rulează versiunea de producție, se introduc sistematic hibe pentru a observa dacă serviciile au posibilitatea de a redresa automat situația creată.

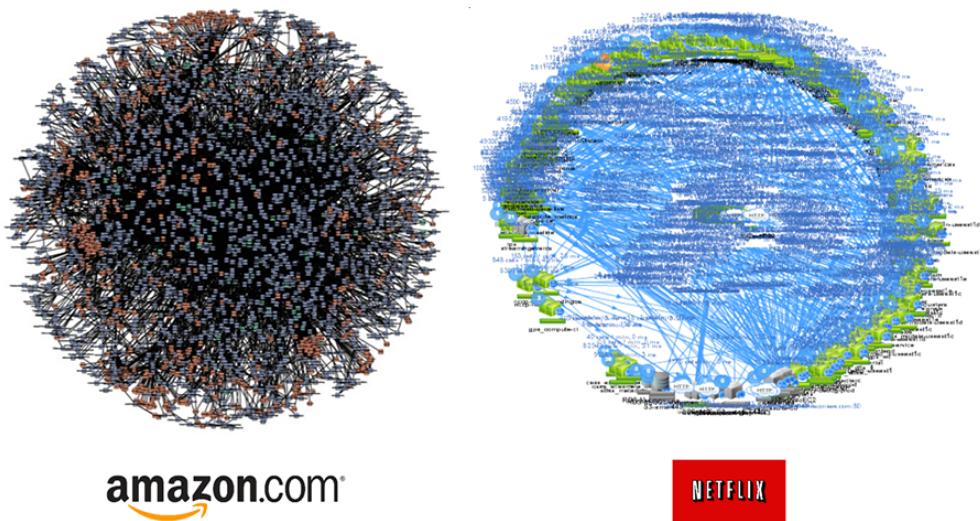


Figura 2.25: Comparație comunicare - Amazon & Netflix [15]

Serviciile ar trebui să rezolve strict problemele de domeniu pentru care au fost create. Aceste probleme sunt, de fapt, cele de care sunt întotdeauna interesați investitorii. Aceștia nu o să dorească să știe care au fost situațiile de criză sau care au fost situațiile în care serverul nu a răspuns cererilor.

Chestiuni precum *logging*, *continuous integration*, *continuous deployment* nu fac parte din ariile pe care ar trebui să le cuprindă serviciile și pot fi acoperite cu ajutorul unor utilitare externe precum ELK Stack (*logging*), Jenkins, TeamCity (*CI & CD*).

În cadrul prezentării *Domain Service Aggregators: A Structured Approach to Microservice Composition*, Caoilte O'Connor a exemplificat procesul de identificare a contextelor specifice unei platforme de difuzare a serialor. La baza acestui proces au stat următoarele trei reguli:

- Utilizarea organizației, a structurii acesteia, a echipelor și a responsabilităților acestora pentru a găsi cele mai bune nuclee pentru viitoarele servicii.
- Definirea unei politici de coeziune, precum și a unei politici de evitare a cuplării din punct de vedere logic.
- Structurarea acestora în funcție de resursele de care avem nevoie.

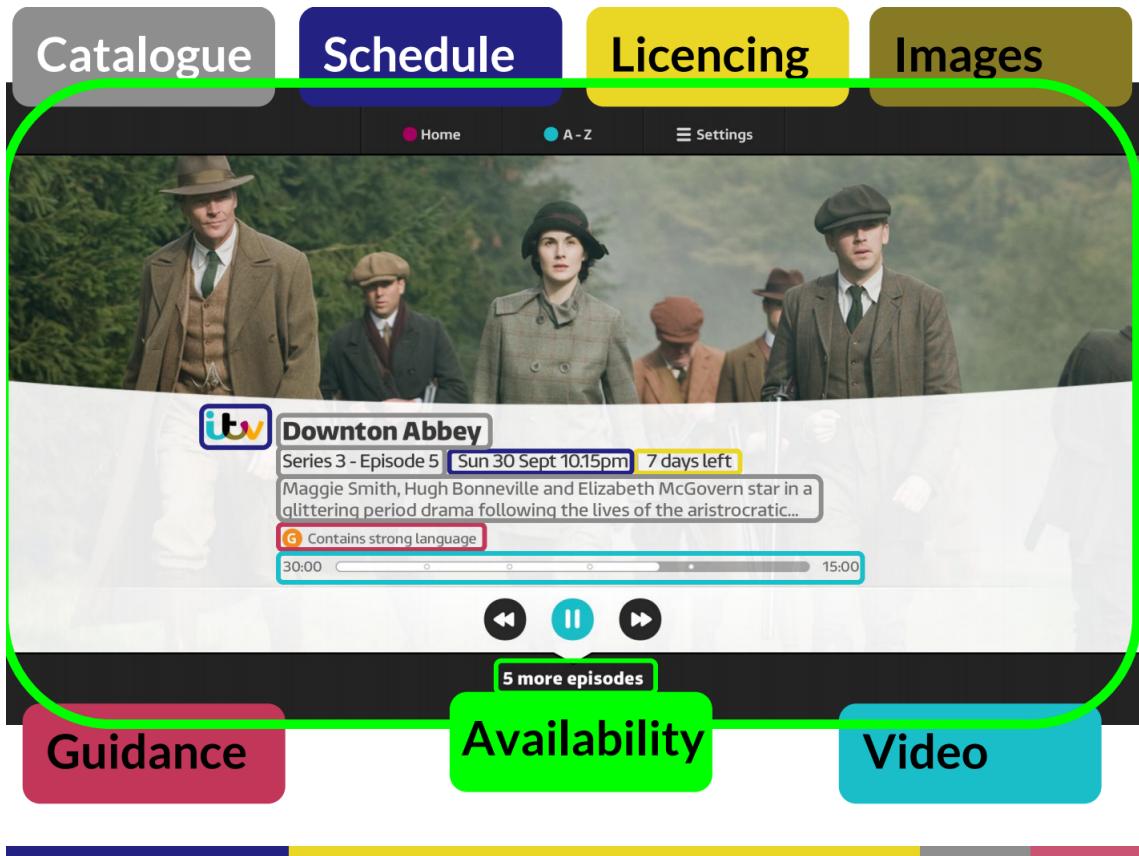


Figura 2.26: Exemplu construcție microservicii în jurul contextelor identificate [16]

În cazul unui distribuitor precum Netflix ar fi existat și alte contexte precum: *Recommendations*, *Ratings*. Conform ultimelor date, mai mult de 80% din conținutul (filme, seriale, showuri TV) la care s-au uitat cei abonați la Netflix a fost descoperit utilizând serviciul de recomandări.

2.4.1 Implementare

Domeniul produsului software *WanVet* a fost împărțit în patru contexte diferite: *UserManagement*, *PetManagement*, *CalendarManagement*, *AppointmentManagement*. Pentru fiecare astfel de context au fost construite două servicii independente, unul pentru partea de citire și unul pentru partea de scriere a datelor. Pentru dezvoltarea acestora, mai precis pentru posibilitatea de a rula o aplicație .NET Core ca un serviciu (Windows) sau ca un proces daemon (Linux), am utilizat pachetul *Das-Mulli.Win32.ServiceUtils*. Una din limitările acestei soluții o reprezintă faptul că mesajele de eroare și excepțiile sunt tratate ca fiind *Win32Exception*. Din păcate, alternativa la această soluție era pachetul *NuGet Topshelf* care, în momentul dezvoltării acestui produs nu avea o versiune funcțională pentru .NET Core 1.1, având în plan să ofere suport doar pentru .NET Core 2.0.

Name	PID	Description	Status
WanVet.Micro.UserManagement.Write	36088	WanVet.Micro.UserManagement.Write	Running
WanVet.Micro.UserManagement.Read	60032	WanVet.Micro.UserManagement.Read	Running
WanVet.Micro.PetManagement.Write	56904	WanVet.Micro.PetManagement.Write	Running
WanVet.Micro.PetManagement.Read	20916	WanVet.Micro.PetManagement.Read	Running
WanVet.Micro.CalendarManagement.Write	64896	WanVet.Micro.CalendarManagement.Write	Running
WanVet.Micro.CalendarManagement.Read	64792	WanVet.Micro.CalendarManagement.Read	Running
WanVet.Micro.AppointmentManagement.Write	59560	WanVet.Micro.AppointmentManagement.Write	Running
WanVet.Micro.AppointmentManagement.Read	58004	WanVet.Micro.AppointmentManagement.Read	Running

Figura 2.27: Serviciile utilizate în cadrul aplicației *WanVet*

În perioada de dezvoltare, reinițializarea tuturor serviciilor este necesară la fiecare modificare a codului din domeniu. A fost realizată automatizarea acestui proces, precum și a procesul de lansare, atât la nivel de serviciu, cât și la nivelul întregii suite.

```
1 #WanVet.Micro.PetManagement.Read.Restart.bat
2 call WanVet.Micro.PetManagement.Read.Unregister.bat
3 call WanVet.Micro.PetManagement.Read.Register.bat
4
5 #WanVet.Micro.PetManagement.Read.Unregister.bat
6 dotnet run --unregister-service
7
8 #WanVet.Micro.PetManagement.Read.Register.bat
9 dotnet restore
10 dotnet run --register-service --urls http://*:5084
```

Extras 17: Exemplu - Reinițializarea unui singur serviciu

```

1 pushd "%~dp0\src\WanVet.Micro.UserManagement.Read\"  

2 call WanVet.Micro.UserManagement.Read.Restart.bat  

3 pushd "%~dp0\src\WanVet.Micro.UserManagement.Write\"  

4 call WanVet.Micro.UserManagement.Write.Restart.bat  

5  

6 pushd "%~dp0\src\WanVet.Micro.PetManagement.Read\"  

7 call WanVet.Micro.PetManagement.Read.Restart.bat  

8 pushd "%~dp0\src\WanVet.Micro.PetManagement.Write\"  

9 call WanVet.Micro.PetManagement.Write.Restart.bat  

10  

11 pushd "%~dp0\src\WanVet.Micro.CalendarManagement.Read\"  

12 call WanVet.Micro.CalendarManagement.Read.Restart.bat  

13 pushd "%~dp0\src\WanVet.Micro.CalendarManagement.Write\"  

14 call WanVet.Micro.CalendarManagement.Write.Restart.bat  

15  

16 pushd "%~dp0\src\WanVet.Micro.AppointmentManagement.Read\"  

17 call WanVet.Micro.AppointmentManagement.Read.Restart.bat  

18 pushd "%~dp0\src\WanVet.Micro.AppointmentManagement.Write\"  

19 call WanVet.Micro.AppointmentManagement.Write.Restart.bat

```

Extras 18: Reinițializarea tuturor serviciilor aplicației WanVet

Pentru a avea întotdeauna posibilitatea de a verifica dacă un serviciu rulează, fără a fi nevoie să ne conectăm *remote* la serverul pe care rezidă, am expus rute publice pentru fiecare dintre acestea. Serviciile care se ocupă de partea de scriere suportă o singură rută, cea de *home*, unde este afișat numele serviciului în cazul în care acesta rulează, iar cele care se ocupă de partea de citire suportă rute Nancy în funcție de posibilitățile de interogare definite.

Nume serviciu	Port	Rute Nancy
WanVet.Micro.AppointmentManagement.Read	5090	Da
WanVet.Micro.AppointmentManagement.Write	5089	Nu
WanVet.Micro.CalendarManagement.Read	5088	Da
WanVet.Micro.CalendarManagement.Write	5087	Nu
WanVet.Micro.PetManagement.Read	5084	Da
WanVet.Micro.PetManagement.Write	5083	Nu
WanVet.Micro.UserManagement.Read	5086	Da
WanVet.Micro.UserManagement.Write	5085	Nu

Capitolul 3

Descrierea aplicației și dezvoltare ulterioară

3.1 Generalități

3.1.1 TypeScript

Limbajul JavaScript este considerat de foarte mulți dezvoltatori unul dintre cele mai viciene. În foarte multe cazuri lipsa unei verificări (dacă variabila curentă este definită sau nu), contextul local/global, valorile *truthy* ("false", "0", etc.)/*falsy* (NaN, 0, etc.) au produs ambiguitate atât în rândul celor care încercau să învețe limbajul, cât și în rândul celor care erau nevoiți să citească și să înțeleagă funcțiile/librăriile scrise de terți (acestea nefiind puține la număr; practic, la ora actuală, există o multitudine de *plugin-uri* jQuery care pot fi utilizate, de exemplu, pentru validarea formularelor, sau pentru adăugarea de noi funcționalități pentru diferite componente: meniuri verticale, calendare, zone pentru text).

Au existat de-a lungul timpului mai multe încercări de a adăuga diferite îmbunătățiri prin definirea unui nou limbaj care să se compileze în JavaScript. Dintre cele mai utilizate, putem să le menționăm pe următoarele: CoffeeScript, ClojureScript, GorillaScript. Majoritatea au fost acceptate pe scară largă deoarece îi dădeau posibilitatea dezvoltatorului să utilizeze diferite concepte (de exemplu: detectarea tipurilor de date, clase, iteratori, programare asincronă) cu care era familiar din alte limbiage (de exemplu: C#, Java).

Una dintre cele mai mari probleme pe care o are limbajul JavaScript este că nu suportă detectarea/validarea tipurilor de date, acestea fiind rezolvate la *runtime*. De

exemplu, suntem nevoiți să scriem o porțiune de cod care menține valoarea curentă a unui coș de cumpărături. Pentru aceasta, am initializat valoarea curentă a coșului cu 0 (tipul de date primitiv: *number*), iar apoi am adăugat valoarea produselor pe care utilizatorul dorește să le cumpere. Pentru fiecare dintre produse, avem un *input* ascuns cu valoarea acestuia. Atributul HTML *value* este un sir de caractere din care trebuie extrasă valoarea în virgulă mobilă pentru a putea calcula corect valoarea finală a coșului de cumpărături. Din păcate, nu primim o eroare dacă însumăm cele două valori, deoarece însumarea dintre o variabilă de tip *number* și una de tip *string* înseamnă, de fapt, concatenarea celor două valori după ce variabila de tip *number* a fost convertită la un sir de caractere (se va apela automat metoda *.toString()*).

```

1 // Utilizarea decoratorilor - DI Angular
2 // Declarăm că acest serviciu este disponibil pentru a fi instantiat
3 @Injectable()
4 export class AuthService
5 {
6     private headers: Headers;
7     private storage: any;
8     isAuthorized: boolean;
9     constructor(private router: Router,
10                 private configuration: Configuration,
11                 private http: Http) { /* */ }
12     // Tipul returnat este void deoarece ștergem din local storage
13     // informațiile despre utilizatorul curent
14     logoff(): void { /* */ }
15     authorize(): void { /* */ }
16     getUserPicture(): string { /* */ }
17     // Tipul returnat este sir de caractere - email-ul utilizatorului
18     getUserEmail(): string {
19         return this.user(null) ? this.user(null).email : '';
20     }
21     getUserName(): string {
22         return this.user(null) ? this.user(null).givenName + ' '
23                               + this.user(null).familyName : '';
24     }
25     userIsAdmin(): boolean {
26         return this.user(null) ?
27             this.user(null).roles.indexOf("wanvet.admin") > -1 : false;
28     }
29     userIsDoctor(): boolean {
30         return this.user(null) ?
31             this.user(null).roles.indexOf("wanvet.doctor") > -1 : false;
32     }
33     // Tipul returnat nu este unul primitiv (number, string, etc), ci unul definit
34     private user(user: User): User { /* */ }
35     /* */
36 }
```

Extras 19: Serviciul de autentificare dezvoltat în TypeScript

În TypeScript tipurile de date nu mai sunt rezolvate la *runtime*, acestea fiind moștenite din modul în care sunt utilizate variabilele (De exemplu, portiunea următoare de cod `var message = "hello world"` este echivalentă cu cea în care definim explicit tipul de date `var message : string = "hello world"`). Aceste modificări au fost repede adoptate de mediile de dezvoltare (Visual Studio, WebStorm) pentru o mai bună observare a erorilor în momentul scrierii codului sursă. Pe lângă aceaste îmbunătățiri, au fost importate și adaptate o serie de funcționalități care erau deja descrise în versiunile limbajului JavaScript ES6 și ES7 (versiuni care nu sunt suportate de majoritatea navigatoarelor). Dintre aceste funcționalități, amintim suportul pentru definirea claselor (ES6) și a interfețelor (propunere ES7), precum și posibilitatea utilizării decoratorilor (ES7).

În extrasul anterior am definit serviciul folosit pentru autentificarea utilizatorilor. Metoda *user* este folosită pentru inițializarea unui utilizator (salvarea rolurilor, a numelui și a email-ului acestuia în *local storage*), returnând modelul utilizatorului nou creat sau *null*, în cazul în care este apelată cu parametrul *null*. Metodele *userIsAdmin*, *userIsDoctor* sunt utilizate pentru a ascunde diferite zone / funcționalități ale aplicației în funcție de rolurile utilizatorului curent. Putem observa că tipul returnat de acestea este *boolean*: *true* sau *false*.

3.1.2 Webpack

Webpack este utilizat pentru transformarea modulelor (împreună cu dependențele lor) în fișiere statice. A fost dezvoltat având ca principal scop managementul acestor module în cadrul produselor software *single-page*, dar a fost acceptat rapid ca fiind calea cea mai bună de a refactoriza partea de *front-end* a oricărei aplicații care utilizează JavaScript, CSS, imagini statice.

Din punct de vedere al mecanicii, Webpack poate procesa doar fișiere JavaScript, dar avem posibilitatea de a utiliza diferite utilitare pentru a transforma orice tip de resursă într-un fișier JavaScript, care mai apoi va deveni un modul în cadrul aplicației noastre. De exemplu, cu ajutorul coffee-loader avem posibilitatea de a utiliza toate resursele care au fost definite în CoffeScript fără a fi nevoiți să le rescriem într-un limbaj comun. Avem parte de același mecanism și pentru stiluri (style-loader) și pentru imagini (url-loader), pentru cele din urmă având posibilitatea de a specifica o limită maximă a mărimii imaginii. Această limită este utilizată pentru a genera sau nu un

URL *inline* în baza 64 pentru imaginea procesată.

În ultimii ani, site-urile au tendința generală de a deveni aplicații web în toată regula. Având în vedere acest fapt deloc de neglijat, au apărut diferite soluții pentru încărcarea fișierelor statice doar în momentul în care este nevoie de funcționalitățile pe care acestea le descriu, precum și pentru înglobarea acestor multiple fișiere într-unul singur pentru a spori astfel timpul de încărcare (utilizând faptul că acest fișier final va ajunge în *cache*). În Webpack există noțiunea de punct de intrare (entrypoint) și avem posibilitatea de a ne defini o suită (bundle) de fișiere JavaScript care să fie încărcate atunci când ajungem pe o anumită pagină a aplicației web. De exemplu, să presupunem că avem o aplicație web cu două pagini, una pentru managementul angajaților, una pentru managementul materiilor prime. Un utilizator din cadrul resurselor umane nu ar trebui să aștepte descărcarea funcționalităților care sunt utilizate în pagina pentru managementul materiilor prime, deoarece nu reprezintă funcționalități pe care el le utilizează. Este probabil ca cele două pagini menționate anterior să folosească aceleși componente Bootstrap, un *framework* precum React sau stiluri comune. În Webpack, avem posibilitatea de a crea o suită (*bundle*) de fișiere comune în care sunt definite funcționalități specifice mai multor pagini și care poate fi memorată (utilizând mecanismul de *caching*) și reutilizată.

Hot Module Replacement e o funcționalitate opțională în Webpack, care din punctul meu de vedere este foarte utilă în mediul de dezvoltare. Ea presupune înlocuirea în timp real a modulelor învechite cu cele modificate de programator. Astfel, acesta are posibilitatea de a modifica stilurile sau fișierele JavaScript/TypeScript reprezentative pentru zona de lucru curentă, iar modulele care le conțin pe acestea sunt actualizate automat și retrimită către *browser*, fără ca pagina curentă să fie reîncărcată, păstrând astfel starea aplicației web, precum și posibilitatea de depanare.

```
1 using Microsoft.AspNetCore.SpaServices.Webpack;
2 public void Configure(IApplicationBuilder app, IHostingEnvironment env,
3                         ILoggerFactory loggerFactory)
4 {
5     app.UseWebpackDevMiddleware(new WebpackDevMiddlewareOptions {
6         HotModuleReplacement = true
7     });
8 }
```

Extras 20: Activarea funcționalității HMR - Startup.cs

```

1 // acest plugin este utilizat pentru extragerea unumitor
2 // portiuni de cod într-un fisier separat
3 // exemplu: extragerea stilurilor din suita curentă pentru optimizarea încărcării lor
4 var ExtractTextPlugin = require('extract-text-webpack-plugin');
5 var extractCSS = new ExtractTextPlugin('vendor.css');
6 var CopyWebpackPlugin = require('copy-webpack-plugin');
7
8 module.exports = {
9   module: {
10     rules: [
11       { test: /\.jpe?g|gif|png|woff|woff2|eot|ttf|svg$/,
12         loader: 'url-loader?limit=100000' },
13       { test: /\.scss$/i,
14         loaders: extractCSS.extract(['css-loader?minimize', 'sass-loader']) },
15       { test: /\.css$/,
16         loader: extractCSS.extract({ fallbackLoader: 'style-loader',
17                                       loader: 'css-loader' }) },
18       { test: /\.json$/,
19         loader: 'json-loader' }
20     ]
21   },
22   entry: {
23     // acest punct de intrare va fi utilizat de orice pagină a aplicației web
24     vendor: [
25       // ...
26       '@angular/common',
27       '@angular/core',
28       // directive angular pentru bootstrap - ex: calendar, typeahead
29       'ngx-bootstrap',
30       'bootstrap',
31       // admin-lte este tema de bază a aplicației web
32       'admin-lte/dist/js/app.js',
33       'admin-lte/dist/css/AdminLTE.min.css',
34       // seturi de pictograme: ionicons, font-awesome
35       'ionicons/dist/css/ionicons.css',
36       'angular2-toaster/toaster.css',
37       'font-awesome/scss/font-awesome.scss'
38     ],
39   },
40   output: {
41     path: path.join(__dirname, '../wwwroot', 'dist'),
42     filename: '[name].js',
43     library: '[name]_[hash]',
44   },
45   plugins: [
46     extractCSS,
47     // copierea imaginilor utilizate ca avatar pentru diferitele roluri
48     // pe care le suportă aplicația web
49     new CopyWebpackPlugin([
50       { from: 'node_modules/admin-lte/dist/img/avatar.png',
51         to: 'assets/avatar_mad.png' },
52       { from: 'node_modules/admin-lte/dist/img/avatar2.png',
53         to: 'assets/avatar_fcl.png' }
54     ]),
55   ],
56 };

```

Extras 21: Fișierul de configurare webpack.config.vendor.js

3.1.3 Angular

Angular, mai precis Angular 2/4, este un *framework* susținut de Google pentru dezvoltarea interfeței client a produselor software *single-page*. Versiunea cu numărul 2.0 a presupus modificări ample aduse arhitecturii, precum și o rescriere a tuturor modulelor de bază. Aceste modificări au fost unele asumate deoarece prima versiune s-a dovedit a avea o curbă de învățare abruptă.

Începând cu versiunea cu numărul 2.0, s-a renunțat la sufixul JS din cadrul numelui deoarece alături de Angular pot fi utilizate limbi precum TypeScript, JavaScript, Dart, alte limbi care se compilează în JavaScript (CoffeeScript, GorillaScript, etc.). Microsoft a colaborat cu Google pentru extinderea limbajului TypeScript astfel încât să existe posibilitatea adnotării claselor (@Component, @Injectable, etc.).

Dezvoltarea interfeței utilizând Angular presupune următoarele etape:

- Dezvoltarea șabloanelor HTML utilizând limbajul de *markup* nativ, precum și cel disponibil în Angular.
- Dezvoltarea claselor @Component utilizate pentru a inițializa/modifica aceste șabloane.
- Dezvoltarea serviciilor utilizate, de exemplu, pentru schimbul de informații între componente, precum și pentru interacțiunea cu un server HTTP.
- Crearea de module care să înglobeze funcționalitățile descrise în componente și servicii. Modulele sunt, de fapt, clase care au decoratorul @NgModule.

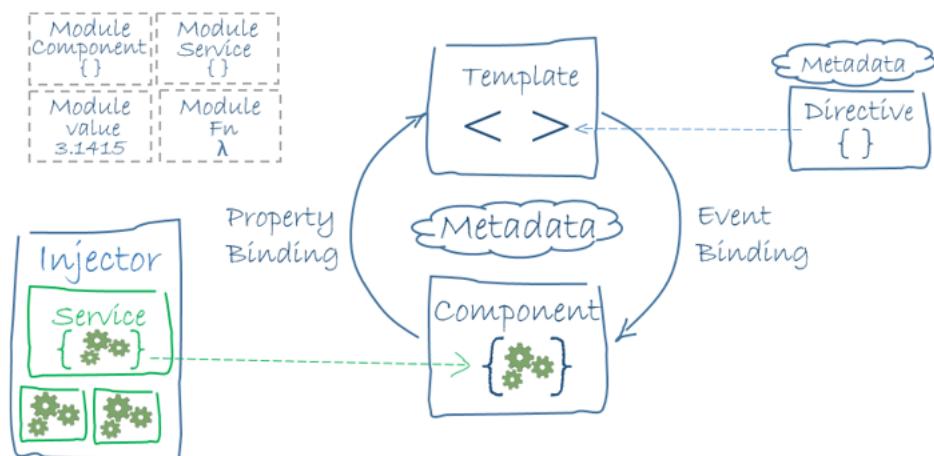


Figura 3.1: Priveliște de ansamblu - Arhitectură Angular [17]

Dacă alegem Angular 2/4 pentru dezvoltarea interfeței client a produselor software suntem nevoiți să definim cel puțin un modul, *AppModule*, care ține loc de modul de bază (rădăcină).

```

1  @NgModule({
2      // Avem nevoie de aceste module în cadrul componentelor
3      // pe care le declarăm în acest modul
4      imports: [
5          MyDatePickerModule,
6          CommonModule,
7          FormsModule,
8          ReactiveFormsModule,
9          RouterModule
10     ],
11     // Components, Directives, Pipes
12     declarations: [
13         // Componente generice care acoperă cazurile de eroare
14         // precum: zone pentru care utilizatorul nu este autorizat sau
15         // cereri către server care nu primesc răspuns într-un timp alocat
16         ErrorMessageComponent,
17         UppercasePipe,
18         UnauthorizedComponent
19     ],
20     // Pot fi utilizate în cadrul altor componente
21     exports: [
22         // Module utilizate pentru crearea formularelor
23         CommonModule,
24         FormsModule,
25         ReactiveFormsModule,
26         RouterModule,
27         MyDatePickerModule
28         // Providers, Components, Directives, Pipes
29         ErrorMessageComponent,
30         UppercasePipe,
31         UnauthorizedComponent
32     ]
33 }
34 })
35 export class SharedModule {
36     static forRoot(): ModuleWithProviders {
37         return {
38             ngModule: SharedModule,
39             // Services, Guards
40             providers: [
41                 ErrorHandlerService,
42                 ApiGatewayService,
43                 UserService,
44                 AuthService,
45                 CanActivateGuard,
46                 NotificationService
47             ]
48         };
49     }
50 }
```

Extras 22: Exemplu @NgModule - *SharedModule*

După cum aminteam anterior în spatele fiecărui şablon HTML stă o clasă care este decorată cu `@Component` și care controlează acea parte a interfeței client.

```

1 import { Component, OnInit } from '@angular/core';
2 import { FormGroup, FormBuilder, Validators, AbstractControl,
3 ValidatorFn, FormArray } from '@angular/forms';
4 import { IMyDpOptions } from 'mydatepicker';
5 import { PetsService } from '../pets.service';
6 // Şablonul HTML și eventualele stiluri se află la adresele specificate
7 @Component({
8   selector: 'appc-pets-add',
9   templateUrl: './pets-add.component.html',
10  styleUrls: ['./pets-add.component.scss']
11 })
12 export class PetsAddComponent implements OnInit {
13   storage = localStorage;
14   // Reactive Forms - petForm este o colecție de obiecte definită ulterior și care
15   // ne oferă posibilitatea de a referi aceste obiecte direct din şablonul HTML
16   // utilizând elemente native
17   petForm: FormGroup;
18   // PetsService înglobează funcționalitățile unui HttpClient
19   constructor(private fb: FormBuilder, private petsService: PetsService) { }
20   ngOnInit() {
21     this.petForm = this.fb.group({
22       name: ['', [Validators.required, Validators.minLength(3)]],
23       breed: ['', [Validators.required]],
24       sex: 'male', species: 'cat', colorHex: ['', []],
25       birthDate: ['', [Validators.required]],
26       profileImageUrl: ['', [Validators.required]]
27     });
28   };
29   // Pentru încărcarea imaginilor suntem nevoiți să autentificăm orice cerere
30   getHeaders(): any {
31     let tokenValue = 'Bearer ' + this.retrieve("authorizationData");
32     return ({ header: 'Accept', value: 'application/json' },
33     { header: 'Authorization', value: tokenValue });
34   };
35
36   retrieve(key: string): string { /* Extrage din localStorage informațiile
37   utilizate la autentificare */;
38   onSubmit(): void { /* Trimit o cerere către server pentru salvarea unui animal
39   de companie */;
40   // Evenimentul se declanșează atunci când un utilizator încarcă o nouă imagine
41   profileImageUploaded($event): void {
42     this.petForm.patchValue({
43       profileImageUrl: $event.serverResponse._body
44     });
45   };
46   // Evenimentul se declanșează atunci când un utilizator șterge o imagine
47   profileImageRemoved($event): void {
48     this.petForm.patchValue({
49       profileImageUrl: '',
50     });
51   }
52 }
```

Extras 23: Exemplu `@Component` - *PetsAddComponent*

3.1.4 ASP.NET Core

ASP.NET Core este un *framework open-source*, care a avut prima versiune stabilă la sfârșitul verii anului 2016. În primele apariții, acest *framework* era denumit ASP.NET 5 pentru că păstra aceeași modalitate de dezvoltare a aplicațiilor, specifică versiunilor anterioare ASP.NET.

Principalele îmbunătățiri sunt următoarele [18]:

- Este complet modular. Toate funcționalitățile din cadrul *framework-ului* sunt pachete NuGet care pot fi importate în cazul în care sunt utilizate pentru dezvoltarea aplicației.
- Noi versiuni ale *framework-ului* ASP.NET Core sunt așteptate cu o cadență de două-trei luni. Astfel, este de apreciat rapiditatea cu care acestea vor ajunge la dezvoltatori (având în vedere că vor fi livrate via NuGet).
- A fost dezvoltat având ca ţintă aplicațiile web care se bazează pe *cloud* [19]. Astfel, există posibilitatea configurării aplicației în funcție de mediul în care aceasta rulează.
- Este *open-source*, acest fapt fiind foarte apreciat de comunitate deoarece majoritatea problemelor / direcțiilor de dezvoltare vor fi dezbatute deschis, fiecare participant având posibilitatea de a contribui.
- Dezvoltarea aplicațiilor care utilizează acest *framework* necesită doar acces la linia de comandă și un simplu editor de text. Instrumentele de lucru, precum și sistemul de operare utilizat sunt la alegerea dezvoltatorului. De asemenea, putem remarcă apariția unui nou editor de cod, *Visual Studio Code*, care poate fi utilizat pe toate cele trei mari sisteme de operare: *Windows*, *Mac*, *Linux*.
- Apariția unui manager de pachete pentru *client-side*, *bower*, care poate fi utilizat pentru importul librăriilor *jQuery*, *bootstrap*, *knockoutJS*, etc.
- Apariția directorului *wwwroot* care cuprinde toate fișierele *JavaScript*, *CSS*, precum și imaginile (pe scurt, fișierele statice ale aplicației).
- Prințipiu *Dependency Injection* a devenit un standard. În acest sens, a fost dezvoltat un *container* în cadrul *framework-ului*, iar serviciile pe care acesta le

cunoaște pot fi configurate în cadrul metodei *ConfigureServices* din clasa *Startup* (putem regăsi un exemplu concludent în subcapitolul 3.3).

- Avem posibilitatea de a configura parcursul unei cereri *HTTP* cu ajutorul unor componente (*middleware*) dezvoltate în cadrul framework-ului: rutare, autenticare, localizare, tratarea erorilor, etc. De asemenea, avem posibilitatea de a ne defini astfel de componente personalizate în funcție de necesitățile pe care trebuie să le satisfacem. Astfel, aceste componente înlocuiesc vechile *HTTP Handlers*, *HTTP Modules*.
- Pentru utilizatorii ASP.NET MVC care foloseau *Razor* pentru crearea paginilor *HTML*, a fost dezvoltat un nou sistem de definire a elementelor HTML care au o legătură directă cu proprietățile modelului. Definirea unui *input* pentru emailul unui utilizator, se poate face utilizând *tag-ul* *asp-for="*"* astfel:
`<input asp-for="Email" class="form-control" />`. În versiunile anterioare ale ASP.NET MVC, pentru a îndeplini acest scop, era utilizat un *HTML Helper*:
`@Html.TextBoxFor(m => m.Email, new { @class = "form-control" })`

Aplicațiile implementate cu ajutorul ASP.NET Core pot rula atât pe mașini virtuale, cât și pe mașini fizice, indiferent de sistemul de operare pe care acestea îl utilizează. De asemenea, putem folosi modalitatea de virtualizare bazată pe recipiente (de exemplu: aplicațiile pot rula în cadrul unor recipiente *Docker* [20]).

3.1.5 RabbitMQ

Agentul de intermediere a mesajelor asigură atât trimiterea mesajelor de la sursă cât și primirea acestora la destinație, având totodată posibilitatea de a stoca aceste mesaje pe disc pentru eventualele întreruperi ale sistemului (pană de electricitate, etc.). De obicei, este o aplicație care rulează în background, dar există și posibilitatea de a ne defini cozi de mesajele în cadrul bazelor de date relationale (*MSSQL*).

În general, în cadrul sistemelor distribuite, apare nevoie de comunicare pentru sincronizarea datelor. Putem presupune că ne aflăm în cadrul unui serviciu care dorește înștiințarea altui serviciu de o anume modificare. De cele mai multe ori, nu există o referință către acesta, deci nu-l poate înștiința direct. De asemenea, există posibilitatea să nu cunoască toate serviciile care sunt interesante de acest mesaj (consumatorii mesajului). Astfel, singura posibilitate este să delege această logică agentului de inter-

mediere. Acesta are un mecanism intern de rutare a mesajelor către serviciile care ar trebui să-l primească. Majoritatea agenților de intermediere nu trimit mesajul către serviciile care ar trebui să-l primească, ci le așteaptă pe acestea să-l ridice. În viața reală scenariul acesta se transpune prin livrarea unui colet în căsuța poștală având următorii actori:

- Expeditorul (serviciul care dorește înștiințarea) - are responsabilitatea de a împacheta coletul (mesajul) și de a-l livra în căsuța poștală (agentul de intermediere)
- Căsuța poștală (agentul de intermediere) - are responsabilitatea de a ține coletul (mesajul) în siguranță până când destinatarul (serviciul care consumă mesajul) îl ridică
- Destinatarul (serviciul care consumă mesajul)

RabbitMQ este un agent de intermediere a mesajelor *open-source* scris în Erlang. Protocolul de mesagerie inclus în varianta de bază este AMQP. De asemenea, trebuie menționat faptul că există API-uri pentru majoritatea limbajelor de programare.

Protocolul AMQP 0.9 reflectă în totalitate funcționalitățile pe care le suportă RabbitMQ:

- **Configurare.** Un serviciu se conectează la RabbitMQ și este deținătorul unei conexiuni stabile. În cadrul acelei conexiuni pot fi unul sau mai multe canale de comunicare. Acestea pot fi utilizate de mai multe fire de execuție, dar acest fapt nu este recomandat în scenariul concurrent. De asemenea, o altă recomandare este reutilizarea aceleiași conexiuni.
- **Producer.** Este serviciul care trimite mesajul utilizând protocolul AMQP. Mesajul ajunge într-un *exchange* și de acolo este rutat instantaneu către un alt *exchange* sau către o coadă de procesare (*queue*) utilizând o anumită legătură (*binding*) și rămâne stocat în cadrul acelei cozi până când este consumat de un alt serviciu. Într-un scenariu mai detaliat, rămâne în cadrul acelei cozi până când serviciul care îl consumă trimite o confirmare (*acknowledgement - Ack*) că l-a procesat cu succes.
- **Consumer.** Este serviciul care consumă mesajul. Acesta nu trebuie să funcționeze în momentul în care primește mesajul deoarece îl primește în cadrul cozii la care este abonat. În momentul în care acesta intră în funcțiune, citește mesajele din

coadă în ordinea în care le-a primit (FIFO). În cazul în care comunicarea s-ar fi întâmplat direct (*Service-Oriented-Architecture*) și serviciul care ar fi trebuit să consume mesajul nu era funcțional, atunci apărea o eroare în cadrul sistemului. Această decuplare facilitată de concepte precum *exchange*, coadă de mesaje specifică duce la o îmbunătățire a disponibilității.

Tipuri de *exchange*:

- *Direct Exchange*

Acest tip de *exchange* rutează mesajul pe care îl primește utilizând cheia de rutare care a fost definită de coada în care acesta este așteptat să ajungă. În momentul definirii unei cozi, se creează automat un *exchange* implicit care are cheia de rutare egală cu numele cozii. Cu ajutorul clientului de .NET pentru RabbitMQ acesta se definește ca având tipul *RabbitMQ.Client.ExchangeType.Direct*.

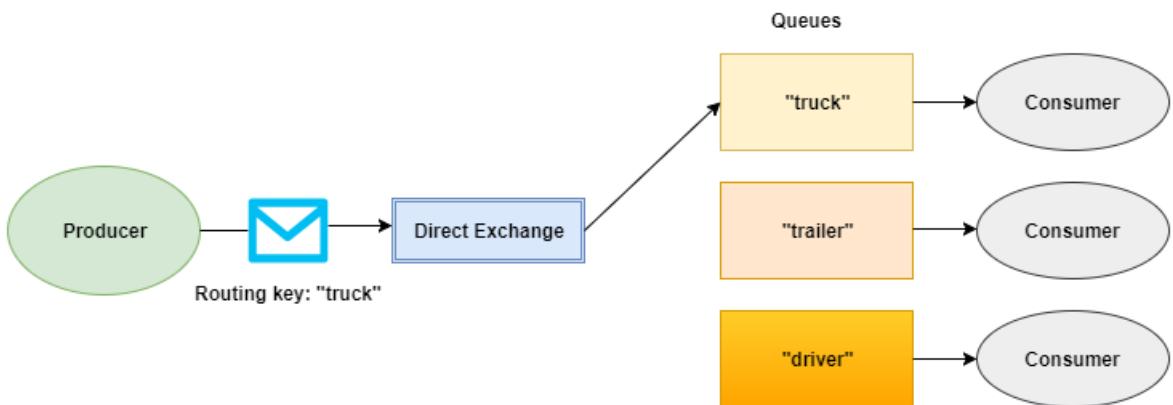


Figura 3.2: *Direct Exchange*

- *Fanout Exchange*

Acest tip de *exchange* rutează mesajul pe care îl primește către toate cozile cu care are definită o legătură (*binding*). Dacă în corpul mesajul primit există o cheie de rutare specificată, aceasta nu se utilizează.

Acest tip de *exchange* este utilizat cu precădere în aplicațiile care implementează şablonul *Publish-Subscribe*. Avem astfel posibilitatea de a declara mai multe cozile a căror consumatori să proceseze mesajul în moduri diferite (raportare, *push notifications*, *logging*)

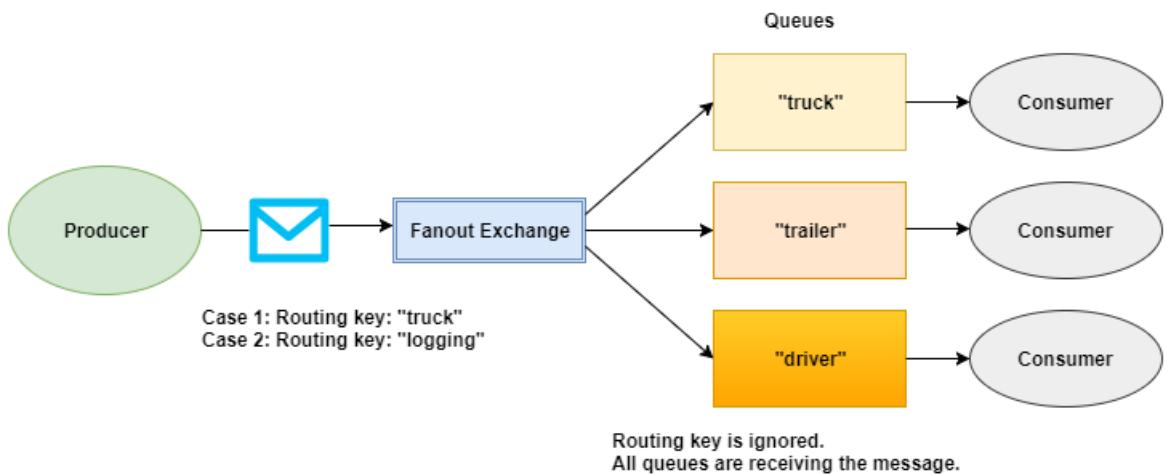


Figura 3.3: *Fanout Exchange*

- *Headers Exchange*

Acest tip de *exchange* rutează mesajul pe care îl primește către toate cozile a căror antet (dicționarul *headers*) îndeplinește condițiile de potrivire cu antetul (dicționar *headers*) mesajului. În antetul cozilor putem specifica dacă dorim ca pentru potrivire să se utilizeze toate obiectele din dicționar (x-match: all) sau dacă mesajul trebuie să aibă cel puțin unul (x-match: any) dintre acestea.

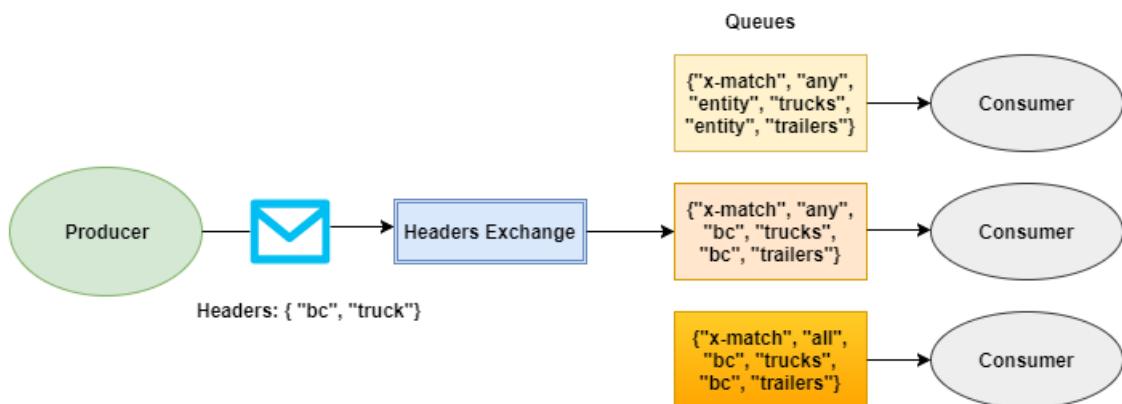


Figura 3.4: *Headers Exchange*

Practic, acest tip de *exchange* este utilizat în cazul în care se dorește filtrarea mesajelor pe baza mai multor criterii.

- *Topic Exchange*

Acest tip de *exchange* rutează mesajul pe care îl primește către toate cozile a căror cheie de rutare îndeplinește condițiile de potrivire cu cheia de rutare specificată în mesaj. Condițiile de potrivire sunt declarate cu ajutorul unor şablonane:

- * - înlocuiește un singur cuvânt din cheia de rutare. Practic menționăm faptul că nu suntem interesați de acel cuvânt.
- # - ține locul tuturor cuvintelor următoare din cheia de rutare. Practic menționăm faptul că nu suntem interesați dacă mai urmează și alte cuvinte în cheia de rutare față de cele pentru care s-a realizat deja potrivirea.

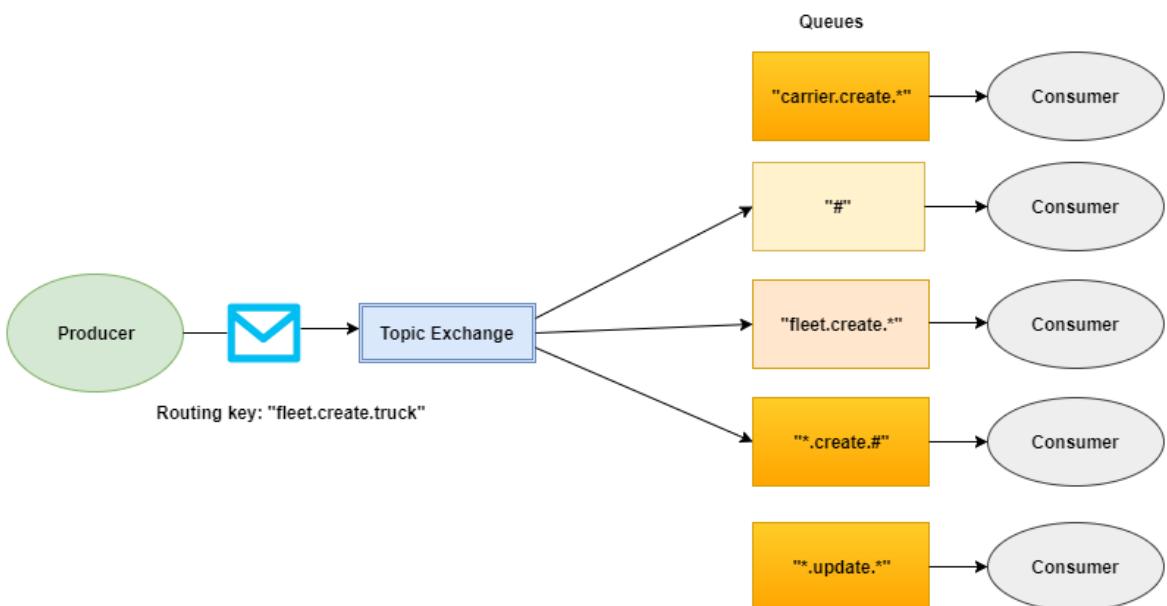


Figura 3.5: *Topic Exchange*

De exemplu, un mesaj care are cheia de rutare `fleet.create.truck`: - îndeplinește condițiile de potrivire cu cozile care au următoarele chei de rutare: `*.create.#` (raportare), `#` (logging), `fleet.create.*` (putem presupune că avem un cache în care salvăm toate entitățile dintr-o flotă, consumatorul acestei cozi fiind responsabil să salveze entitățile nou create), etc. - nu îndeplinește condițiile de potrivire cu cozile care au următoarele chei de rutare: `*.update.*`, `carrier.create.*`, etc

3.1.6 MassTransit

MassTransit este un canal de comunicare (*Bus*) gratuit, *open source*, utilizat pentru dezvoltarea sistemelor distribuite. Nu este o soluție *enterprise*, deși este utilizată cu succes în acest mediu. A fost gândit ca un canal de comunicare între serviciile interne ale unei companii (rulând în spatele unui *firewall*). Este preferat în detrimentul clientului pus la dispoziție de cei care au dezvoltat RabbitMQ deoarece oferă în plus următoarele funcționalități [21]:

- Managementul conexiunii. Tratează cu succes scenariul în care pierdem conexiunea către agentul de intermediere a mesajelor restaurând atât conexiunea cât și orice *exchange*, coadă sau legătură (*binding*) la starea anterioară, oferind astfel durabilitate.
- Serializare. Agentul de intermediere al mesajelor, în cazul nostru RabbitMQ, utilizează octeți, astfel introducând problema formatării mesajului pe perioada transferului. MassTransit suportă serializatori precum JSON, BSON, XML, .NET *binary formatter*. Există posibilitatea encriptării conținutului mesajelor (AES-256) pentru a îndeplini cerințele PCI/HIPAA (*Health Insurance Portability and Accountability Act*).
- Concurență.
- Tratarea excepțiilor. Politică de reîncercare. Mesaje poluate.
- Corelarea mesajelor.
- Rutarea mesajelor. MassTransit utilizează doar cozi pentru definirea comunicării. Se definesc automat multiple obiecte de tipul *exchange*.
- Integrarea cu Quartz.NET pentru trimiterea pe canalul de comunicare a unor mesaje programate anterior. Integrarea cu Entity Framework și NHibernate
- Managementul duratei de viață a consumatorilor.
- Implementarea unor abstracții: Saga.
- *Reactive Extensions*
- Metrici de performanță. Testare Unitară.

```

1 public class BusConfigurator
2 {
3     public IBusControl ConfigureBus(
4         Action<IRabbitMqBusFactoryConfigurator, IRabbitMqHost> registrationAction = null)
5     {
6         return Bus.Factory.CreateUsingRabbitMq(cfg =>
7             {
8                 var host = cfg.Host(new Uri(RabbitMQConstants.RabbitMQUri), hst =>
9                     {
10                         hst.Username(RabbitMQConstants.RabbitMQUserName);
11                         hst.Password(RabbitMQConstants.RabbitMQPassword);
12                     });
13                     UseInnerRetry(cfg);
14                     UseInnerCircuitBreaker(cfg);
15                     UseInnerRateLimiter(cfg);
16                     registrationAction?.Invoke(cfg, host);
17                 });
18             );
19     }
}

```

Extras 24: Configurare canal de comunicare

Utilizând metoda prezentată în extrasul anterior avem posibilitatea să ne definim consumatorii pentru un anumit serviciu. De exemplu, în cadrul serviciului care se ocupă cu salvarea și stocarea utilizatorilor, ne-am definit un consumator care are responsabilitatea de a digera mesajul *CreateUserCommand*.

```

1 // Au fost utilizate pachetele Autofac.Extensions.DependencyInjection
2 // și MassTransit.AutofacIntegration
3 public class DefaultModule : Module
4 {
5     protected override void Load(ContainerBuilder builder)
6     {
7         containerBuilder.RegisterType<Repository>().As< IRepository>();
8         containerBuilder.RegisterConsumers(typeof(DefaultModule).GetTypeInfo()
9             .Assembly).AsSelf().AsImplementedInterfaces();
10        containerBuilder.RegisterGeneric(typeof(AutofacConsumerFactory<>))
11            .WithParameter(new NamedParameter("name", "message"))
12            .As(typeof(IConsumerFactory<>)).InstancePerLifetimeScope();
13        containerBuilder.Register((c) =>
14        {
15            var busControl = BusConfigurator.Instance.ConfigureBus((cfg, host) =>
16                {
17                    ConfigureEndPoints(cfg, host, c);
18                });
19                busControl.Start();
20                return busControl;
21            }).SingleInstance().AutoActivate();
22        }
23    }
}

```

Extras 25: Modul Autofac - WanVet.Micro.UserManagement.Write

Un simplu mesaj este de cele mai multe ori parte a unui flux de lucru complex. Dacă privim un sistem distribuit în ansamblu, putem observa multiple mesaje care circulă între servicii, dar nu putem identifica un anumit flux. Un saga este utilizat pentru coordonarea fluxului de lucru, fiind necesară stocarea internă a unui cumul de informații care definesc procesul de *business* curent. De fapt, un saga nu este nici mai mult nici mai puțin decât o mașină cu un număr finit de stări (are stare inițială, suportă evenimente pe baza cărora își schimba starea). Pentru a putea defini un astfel de flux avem nevoie și de pachetul Automatonymous.

```

1 public class UserSaga : MassTransitStateMachine<UserSagaState>
2 {
3     public State Received { get; private set; }
4     public State Registered { get; private set; }
5
6     public Event<ICreateUserCommand> CreateUser { get; private set; }
7     public Event<IUserCreatedEvent> UserCreated { get; private set; }
8
9     public UserSaga()
10    {
11        InstanceState(s => s.CurrentState);
12
13        // În momentul înregistrării unui utilizator suntem nevoiți să avem
14        // o proprietate care ne ajută pentru identificarea unică a acestora
15        Event(() => CreateUser,
16            cc =>
17                cc.CorrelateBy(state => state.Email, context =>
18                    context.Message.Email)
19                    .SelectId(context => Guid.NewGuid());
20        // Apoi, relația dintre saga și mesaje este descrisă de CorrelationId
21        Event(() => UserCreated, x => x.CorrelateById(context =>
22            context.Message.CorrelationId));
23
24        Initially(
25            When(CreateUser)
26                .Then(context =>{ /* */ })
27                .TransitionTo(Received)
28                .Publish(context => new UserReceivedEvent(context.Instance))
29        );
30
31        During(Received,
32            When(UserCreated)
33                .Then(context =>{ /* */ })
34                .Finalize()
35        );
36
37        SetCompletedWhenFinalized();
38    }
39 }
```

Extras 26: Exemplu Saga - MassTransit & Automatonymous

MassTransit creează automat câte un *exchange fanout* pentru fiecare tip de mesaj pe care îl primește. În cazul în care tipul de mesaj primit este unul derivat, se creează un *exchange* separat și pentru tipul de bază. În cadrul produsului software pe care l-am dezvoltat, interfața *IUserCreatedEvent* moștenește interfața *IEvent*. În momentul în care un nou utilizator este înrolat în sistem se emite un eveniment de tipul *IUserCreatedEvent*. În figura următoare putem observa existența unui *exchange* atât pentru *IUserCreatedEvent*, cât și pentru tipul de bază *IEvent*.

Virtual host	Name	Type	Features	Message rate in	Message rate out	+/-
wanvet	(AMQP default)	direct	D			
wanvet	WanVet.Messaging.Commands:ICreateUserCommand	fanout	D			
wanvet	WanVet.Messaging.Events:IUserCreatedEvent	fanout	D	2.6/s	2.6/s	
wanvet	WanVet.Messaging:IEvent	fanout	D			
wanvet	WanVet.Messaging:IMessage	fanout	D			
wanvet	amq.direct	direct	D			
wanvet	amq.fanout	fanout	D			
wanvet	amq.headers	headers	D			
wanvet	amq.match	headers	D			
wanvet	amq.rabbitmq.trace	topic	D I			
wanvet	amq.topic	topic	D			
wanvet	domains.usermanagement.queue	fanout	D	2.8/s	2.8/s	
wanvet	domains.usermanagement.queue_skipped	fanout	D	0.60/s	0.60/s	
wanvet	ignore	fanout	D AD Exp			

Figura 3.6: Rata de mesaje/secundă - RabbitMQ

3.1.7 EventStore

EventStore este un proiect *open-source* utilizat pentru stocarea informațiilor sub forma unor evenimente imuabile. Din punct de vedere al disponibilității, EventStore poate rula pe un singur nod sau ca un cluster de noduri, rămânând stabil atât timp cât jumătate dintre noduri sunt active.

Integrarea cu EventStore este facilitată de interfața nativă HTTP pe care acesta o pune la dispoziția clientilor. De asemenea, există posibilități de integrare pentru majoritatea limbajelor de programare: .NET, Erlang. În cazul în care numărul abonaților la evenimente este foarte mare, este recomandată utilizarea interfeței HTTP care se bazează pe protocolul AtomPub.

Din punct de vedere al performanței, luând în calcul şablonanele de proiectare și configurările interne, EventStore suportă până la 15000 de scrieri pe secundă și 50000 de citiri pe secundă.

În cadrul aplicației pe care am dezvoltat-o, am ales pentru integrare pachetul NuGet EventStore.ClientAPI.NetCore care se află momentan la versiunea 4.0.0 alpha.

```
1 // Fabrica utilizată pentru crearea unei noi conexiuni
2 public class EventStoreConnectionFactory
3 {
4     public static IEventStoreConnection Create()
5     {
6         var eventStoreConnection = EventStoreConnection.Create(new IPPEndPoint(
7             EventStoreConfiguration.Address, EventStoreConfiguration.Port));
8         eventStoreConnection.ConnectAsync().Wait();
9         return eventStoreConnection;
10    }
11 }
12 // Înregistrarea fabricii definite anterior în cadrul unui modul Autofac
13 public class DefaultModule : Autofac.Module
14 {
15     protected override void Load(ContainerBuilder containerBuilder)
16     {
17         containerBuilder.Register(c => new EventTypeResolver(
18             typeof(DefaultModule).GetTypeInfo().Assembly)).As<IEventTypeResolver>();
19         containerBuilder.Register(c => EventStoreConnectionFactory.Create())
20     }
21 }
```

Extras 27: Configurare conexiune EventStore

```
1 var readStreamSize = 500;
2 var streamName = "exampleStream";
3 var eventNumber = 0L;
4 StreamEventsSlice currentSlice;
5
6 do
7 {
8     currentSlice = await _eventStoreConnection.ReadStreamEventsForwardAsync(
9                 streamName, eventNumber, readStreamSize, false);
10    eventNumber = currentSlice.NextEventNumber;
11 } while (!currentSlice.IsEndOfStream);
```

Extras 28: Exemplu citire - EventStore

În următoarele două figuri am prezentat structura evenimentelor stocate în EventStore în urma creării unui utilizator și adăugării unui animal de companie care să-l aibă pe acesta ca deținător.

0@user-c37b40e8-01ca-42f2-b9c4-300f0c6171f6

No	Stream	Type
0	user-c37b40e8-01ca-42f2-b9c4-300f0c6171f6	UserCreatedEvent
Data		
<pre>{ "Id": "c37b40e8-01ca-42f2-b9c4-300f0c6171f6", "Email": "dumitru@yahoo.com", "FamilyName": "Alex", "GivenName": "Razvan", "PhoneNumber": "754621692", "Gender": "Male" }</pre>		
Metadata		
<pre>{ "CommitId": "d51b05a0-de49-4751-aea2-ba83c2fd5622", "AggregateClrTypeName": "WanVet.Micro.UserManagement.Write.Domain.Model.UserModel.User, WanVet.Micro.UserManagement.Write, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null", "ServerClock": "2017-08-31T12:57:54.9479176Z", "EventClrTypeName": "WanVet.Micro.UserManagement.Write.Domain.Model.Events.UserCreatedEvent, WanVet.Micro.UserManagement.Write, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" }</pre>		

Figura 3.7: Stocarea unui eveniment utilizat la crearea unui utilizator

0@pet-8c5b3356-5b04-477a-a789-1fe44e54edce

No	Stream	Type
0	pet-8c5b3356-5b04-477a-a789-1fe44e54edce	PetCreatedEvent
Data		
<pre>{ "Id": "8c5b3356-5b04-477a-a789-1fe44e54edce", "OwnerId": "c37b40e8-01ca-42f2-b9c4-300f0c6171f6", "Name": "Linda", "Breed": "European", "Sex": "female", "Species": "cat", "ColorHex": "#857979", "BirthDate": "2014-09-12T21:00:00Z", "ProfileImageURL": "http://i.imgur.com/YU4U8z6.jpg" }</pre>		
Metadata		
<pre>{ "CommitId": "a14fa6c6-e435-42e4-ac98-adb939d6eda7", "AggregateClrTypeName": "WanVet.Micro.PetManagement.Write.Domain.Model.PetModel.Pet, WanVet.Micro.PetManagement.Write, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null", "ServerClock": "2017-08-31T13:12:11.7348015Z", "EventClrTypeName": "WanVet.Micro.PetManagement.Write.Domain.Model.PetModel.Events.PetCreatedEvent, WanVet.Micro.PetManagement.Write, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" }</pre>		

Figura 3.8: Stocarea unui eveniment utilizat la crearea unui animal de companie

3.1.8 Redis

Redis este un depozit de date *open-source* utilizat în funcție de necesități ca un strat de *caching*, ca un agent de intermediere a mesajelor sau ca o bază de date.

The screenshot shows the Redis web interface with the following details:

- Hash:** user
- Key:** dumitru@yahoo.com
- Value:**

```
[
  {
    "Id": "c37b40e8-01ca-42f2-b9c4-300f0c6171f6",
    "Email": "dumitru@yahoo.com",
    "FamilyName": "Alex",
    "GivenName": "Razvan",
    "PhoneNumber": "754621692",
    "Gender": "Male",
    "Version": 0,
    "Pets": [],
    "RedisKey": "user"
  }
]
```

Figura 3.9: *UserReadModel* - Inițial

Avem posibilitatea de a dezvolta scripturi și de a accesa funcționalități *low-level* precum lucrul cu semafoare utilizând limbajul Lua.

Redis suportă următoarele tipuri de structuri de date: șiruri de caractere, liste simple și ordonate, hashuri, bitmaps, hyperloglogs, precum și posibilitatea de a descrie date geo-spațiale care pot fi interogate specificând o anumită rază.

Din punct de vedere al disponibilității, Redis are un sistem propriu de replicare a datelor, suportând în același timp multiple scenarii de persistență. Durabilitatea datelor se bazează pe partaionarea automată a acestora în cadrul unui cluster.

În cadrul aplicației pe care am dezvoltat-o, am ales pentru integrare pachetul NuGet `NETCore.RedisKit` care există în două versiuni în funcție de versiunea .NET Core aleasă: 1.0.1 (pentru .NET Core 1.1), 2.0.0 (pentru .NET Core 2.0).

HASH: pet			Size: 1	TTL: -1	Rename	Delete	Set TTL
row	key	value					
1	8c5b3356-5b04-477a-a...	{"Id": "8c5b3356-5b04-477a-a789-1fe44e54edce", "OwnerId": "c37b40e8-01ca-42f2-b9c4-300f0c6171f6", "Version": 0, "Name": "Linda", "Breed": "European", "S..."} 8c5b3356-5b04-477a-a789-1fe44e54edce	+ Add row - Delete row				

Key: size in bytes: 36
Value: size in bytes: 295

```
{
  "Id": "8c5b3356-5b04-477a-a789-1fe44e54edce",
  "OwnerId": "c37b40e8-01ca-42f2-b9c4-300f0c6171f6",
  "Version": 0,
  "Name": "Linda",
  "Breed": "European",
  "Sex": "female",
  "Species": "dog",
  "ColorHex": "#857979",
  "BirthDate": "2014-09-12T21:00:00Z",
  "ProfileImageURL": "http://i.imgur.com/YU4U8z6.jpg",
  "RedisKey": "pet"
}
```

View as: Plain Text | JSON | Save

Figura 3.10: *PetReadModel* - Inițial

În cele două figuri anterioare am prezentat modelele *UserReadModel* și *PetReadModel* stocate independent în cadrul a două hashuri diferite: *user* și *pet*, cheia fiind reprezentată în primul caz de adresa de email, iar în cel de-al doilea caz de un identificator unic generat automat.

În momentul în care se înscrive un nou animal de companie în sistem, dorim să actualizăm lista de animale de companie a detinătorului astfel încât să nu fim nevoiți să citim datele acestuia și apoi datele pentru fiecare animal de companie în parte.

În extrasul următor am prezentat modalitatea prin care am actualizat lista în cauză în momentul în care interceptăm un eveniment de tipul *IPetCreatedEvent*.

```

1 public async Task Consume(ConsumeContext<IPetCreatedEvent> context)
2 {
3     var @event = context.Message;
4     var user = new UserReadModel();
5     user = _redisService.HashGet<UserReadModel>($"{{user.RedisKey}}",
6                                     $"{{@event.OwnerEmail}}", CommandFlags.PreferMaster);
7     user.Pets.Add(new PetReadModel
8     {
9         Id = @event.AggregateId,
10        Breed = @event.Breed,
11        Name = @event.Name,
12        ProfileImageUrl = @event.ProfileImageUrl,
13        Sex = @event.Sex,
14        Species = @event.Species
15    });
16    _redisService.HashSet($"{{user.RedisKey}}", $"{{@event.OwnerEmail}}",
17                           user, When.Always, CommandFlags.PreferMaster);
18 }

```

Extras 29: Exemplu utilizare Redis - Actualizare valoare din hashul *user*

În figura următoare am prezentat valoarea stocată în cadrul hashului *user* având cheia "dumitruar@yahoo.com", după ce aceasta a fost actualizată în concordanță cu evenimentul procesat.

The screenshot shows a Redis interface with a table for the 'user' hash. There is one row:

row	key	value
1	dumitruar@yahoo.com	{"Id": "c37b40e8-01ca-42f2-b9c4-300f0c6171f6", "Email": "dumitruar@yahoo.com", "FamilyName": "Alex", "GivenName": "Razvan", "PhoneNumber": "+754621692", "Gender": "Male", "Version": 0, "Pets": [{"Id": "8c5b3356-5b04-477a-a789-1fe44e54edce", "Name": "Linda", "Breed": "European", "Sex": "Female", "Species": "cat", "ProfileImageUrl": "http://i.imgur.com/YU4U8z6.jpg"}], "RedisKey": "user"}

Below the table, there are two sections: 'Key: size in bytes: 19' containing 'dumitruar@yahoo.com' and 'Value: size in bytes: 359' containing the JSON representation of the UserReadModel object.

Figura 3.11: *UserReadModel* - După adăugarea unui animal de companie

3.2 Arhitectură

Aplicația dezvoltată ca urmare a studierii practicilor amintite anterior este formată din două părți:

- *IdentityProvider* - Produs software ASP.NET Core 1.1 (MVC) utilizat pentru emiterea unui token de autentificare pentru *WanVet* Web API.

- WanVet - Produs software ASP.NET Core 1.1 - Web API a cărui interfață este reprezentată de o aplicație *single-page* dezvoltată în Angular și o serie de micro-servicii care reprezintă, de fapt, nucleele logice ale produsului.

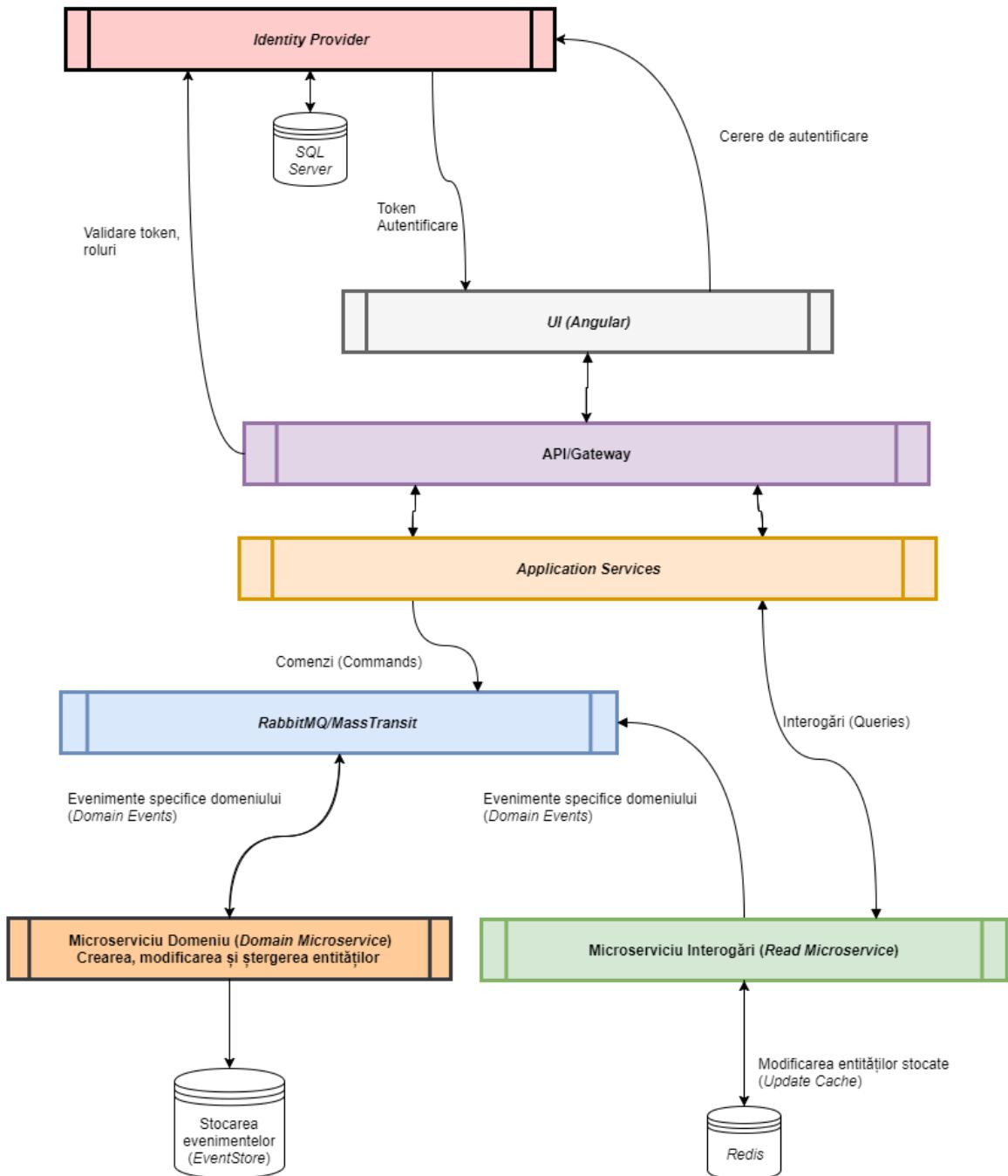


Figura 3.12: Arhitectura aplicației WanVet - Interacțiune *IdentityProvider*

Identitatea utilizatorilor este stocată într-o bază de date SQL Server (*IdentityDB*) având în schema *dbo* următoarele tabele: *AspNetRoleClaims*, *AspNetRoles*, *AspNetUserClaims*, *AspNetUserLogins*, *AspNetUserRoles*, *AspNetUsers*, *AspNetUserTokens*.

În figura anterioară am prezentat o privire de ansamblu asupra arhitecturii în care putem observa şablonul CQRS. În să precizez că cele două microservicii (*Domain Microservice* și *Read Microservice*) sunt reprezentative doar pentru un singur context identificat, în soluția finală existând patru astfel de contexte.

Soluția este organizată pe patru straturi:

- *Presentation*. Este compus din două părți: client și server. Partea de client este reprezentată de aplicația *single-page*. Partea de server este reprezentată de *WanVet Web API*.
- *Application*. Este compus din logica utilizată pentru coordonarea cererilor primite de la clienți spre a fi rezolvate de serviciile din stratul *Domain*.
- *Domain*. Este compus din multitudinea serviciilor care se comportă ca nuclee logice specifice fiecărui context identificat (DDD - *Bounded Context*).
- *Infrastructure*. Este compus din două proiecte, cel pentru citire fiind responsabil de conexiunea cu Redis pentru menținerea stării aplicației, iar cel pentru scriere fiind responsabil de conexiunea cu EventStore pentru stocarea evenimentelor.

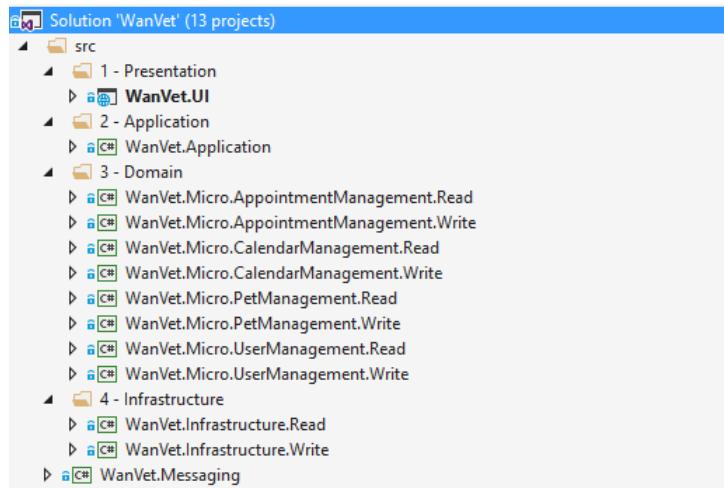


Figura 3.13: Soluția aplicației *WanVet*

3.3 Autentificare

Pentru a păstra modularitatea produsului final, serviciul de autentificare a fost dezvoltat separat de restul microserviciilor care au ca scop interacțiunea cu domeniul.

De fapt, este un produs software independent (*IdentityProvider*), dezvoltată utilizând şablonul *Model-View-Controller*.

Pentru definirea modelelor cu care vom opera, a fost utilizat sistemul de *membership* pus la dispoziţie de Microsoft odată cu lansarea noului *framework* ASP.NET Core, şi anume: ASP.NET Core Identity.

După cum aminteam şi în subcapitolul dedicat prezentării acestui *framework*, procesul de *deployment* şi cel de dezvoltare a aplicaţiilor ASP.NET s-au modificat destul de mult odată cu apariţia ASP.NET Core. Astfel, la *runtime*, este chemată metoda *ConfigureServices* din clasa *Startup*, în care avem posibilitatea de a ne defini specificaţiile serviciilor pe care le utilizăm.

```
1 public void ConfigureServices(IServiceCollection services)
2 {
3     services.AddDbContext<ApplicationDbContext>(options =>
4         options.UseSqlServer(
5             Configuration.GetConnectionString("DefaultConnection")));
6
7     services.AddIdentity< ApplicationUser, ApplicationRole>()
8         .AddEntityFrameworkStores<ApplicationDbContext>()
9         .AddDefaultTokenProviders();
10
11    // Adăugarea rolurilor iniţiale
12    // Iniţializarea administratorului
13    services.AddTransient<SeedDbData>();
14 }
```

Extras 30: *Configure Services - IdentityProvider*

În extrasul anterior am specificat care a fost modalitatea de stocare a datelor reprezentative pentru identitatea utilizatorilor. După cum putem observa, am ales stocarea acestora într-o bază de date SQL Server, şi am definit clasele prin care am modelat utilizatorii şi rolurile acestora: *ApplicationUser* şi *ApplicationRole*. Aceste două clase moştenesc clasele *default* din pachetul *Identity.EntityFrameworkCore*, şi anume *IdentityUser* şi *IdentityRole*.

Serviciile pe care le configuroam în metoda specificată anterior sunt disponibile la nivel general prin *dependency injection*. Presupunând că avem un *controller* pentru administrarea utilizatorilor, înseamnă că avem la dispoziţie două clase generice pentru adăugarea, editarea şi ştergerea utilizatorilor, precum şi a rolurilor acestora.

```

1 [Authorize(Roles = "Admin")]
2 public class UsersAdminController : Controller
3 {
4     private readonly UserManager< ApplicationUser> _userManager;
5     private readonly RoleManager< ApplicationRole> _roleManager;
6
7     public UsersAdminController(
8         UserManager< ApplicationUser> userManager,
9             RoleManager< ApplicationRole> roleManager)
10    {
11        _userManager = userManager;
12        _roleManager = roleManager;
13    }
14}

```

Extras 31: *Controller* pentru administrarea utilizatorilor

După cum putem observa în extrasul anterior, *UsersAdminController* poate fi accesat doar de utilizatorii care au rolul de administrator. În configurația curentă au fost definite cinci roluri: *Admin*, *wanvet.user*, *wanvet.admin*, *wanvet.staff* și *wanvet.doctor*. Administratorul are drepturi depline în cadrul produsului software *IdentityProvider*. Celelalte patru roluri sunt prefixate cu numele clientului, și descriu accesibilitatea la resursa denumită în concordanță cu aceasta (*wanvet*).

În metoda utilizată pentru configurarea serviciilor, pe care am amintit-o anterior, avem posibilitatea de a defini resursele, precum și clienții care se ocupă de autentificare. Aceștia cer de la *IdentityProvider* un *token*, iar pe baza acestuia pot avea acces la resursele specificate (în cazul nostru *wanvet*).

```

1 public void ConfigureServices(IServiceCollection services)
2 {
3     var cert = new X509Certificate2(
4         Path.Combine(_environment.ContentRootPath, "localhost.pfx"),
5                                         "local");
6
7     services.AddIdentityServer()
8         .AddSigningCredential(cert)
9         // Definirea resurselor
10        .AddInMemoryScopes(Config.GetScopes())
11        // Definirea clientilor și a accesului acestora la resurse
12        .AddInMemoryClients(Config.GetClients())
13        .AddAspNetIdentity< ApplicationUser >()
14        .AddProfileService< IdentityWithAdditionalClaimsProfileService >();
15}

```

Extras 32: Definirea resurselor și a clientilor

La baza implementării produsului software *IdentityProvider* a stat pachetul NuGet *IdentityServer 4*. Mare parte din metodele de configurare prezentate în acest subcapitol fac parte din acest pachet.

În extrasul anterior, metoda *GetScopes()* întoarce o listă de scopuri (*IEnumerable<IdentityServer4.Models.Scopes>*) care sunt utilizate în configurarea accesului clientilor la resurse. Scopurile reprezintă, într-un limbaj trivial, ceea ce avem voie să facem. În cadrul produsului software *IdentityProvider* a fost creat scopul generic *wanvet*. De obicei, scopurile se definesc la nivel de citire / scriere (de exemplu: *wanvet.read* și *wanvet.write*), dar pentru scenariul curent nu avem nevoie de acest nivel de granularitate, deoarece ne vom baza pe rolurile definite pentru a descrie accesibilitatea.

În următorul extras am prezentat inițializarea unui client care utilizează scopul unic, identitate (*Identity - OpenID Connect*), precum și un scop de tip resursă (pe care l-am utilizat pentru a defini accesul la resursa generică *wanvet*).

```
1 new Client
2 {
3     ClientName = "WanVetClientDev",
4     ClientId = "WanVetClientDev",
5     AccessTokenType = AccessTokenType.Reference,
6     AllowedGrantTypes = GrantTypes.Implicit,
7     AllowAccessTokensViaBrowser = true,
8     // redirectionarea utilizatorilor autentificați
9     RedirectUris = new List<string>{ "https://localhost:44328" },
10    // redirectionarea utilizatorilor neautentificați
11    PostLogoutRedirectUris = new List<string>
12    {
13        "https://localhost:44328/unauthorized"
14    },
15    AllowedCorsOrigins = new List<string>
16    {
17        "https://localhost:44328",
18        "http://localhost:44328"
19    },
20    AllowedScopes = new List<string>
21    {
22        "openid",
23        "wanvet"
24    }
25 }
```

Extras 33: *WanVetClientDev* - clientul utilizat în mediul de dezvoltare

Ambele aplicații, și cea utilizată pentru autentificarea utilizatorilor, precum și aplicația client, au fost configurate pentru a rula sub un certificat creat *a priori* pentru *localhost*. În mediul de dezvoltare am utilizat IIS Express, iar aplicațiile rulează

pe porturile 44348 (*IdentityProvider*) și 44328 (*WanVetClientDev*). Pentru a simula mediul de producție am utilizat Local IIS, iar aplicațiile rulează la următoarele adrese: <https://localhost/IdentityProvider> și <https://localhost/Wanvet> (*WanVetClient*).

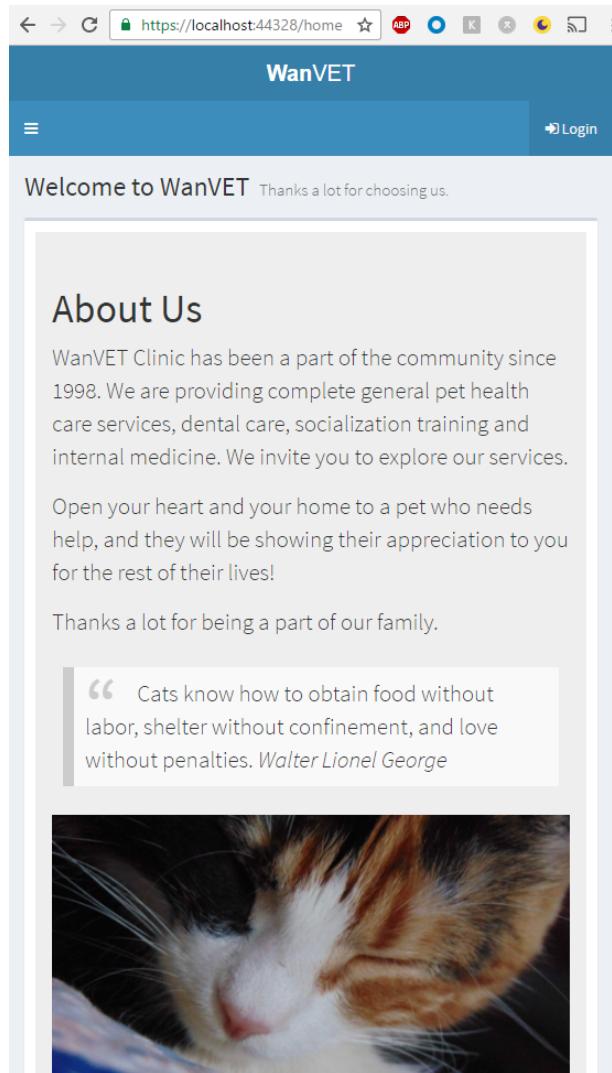


Figura 3.14: Pagina principală a clientului - utilizator neautentificat

În figura anterioară putem observa pagina principală a clientului în contextul unui utilizator neautentificat. Autentificarea se realizează prin simpla apăsare a butonul *Login*. Utilizatorul este redirecționat către <https://localhost:44348/> unde rezidă (în mediul de dezvoltare) aplicația *IdentityProvider*. După introducerea email-ului și a parolei sau după ce utilizatorul își completează datele, înregistrându-se astfel pentru un cont nou, acesta va fi redirecționat către o pagină în care poate selecta resursele la care va avea acces aplicația client (în cazul nostru: *OpenID* și *wanvet*), după cum se poate observa în figura următoare.

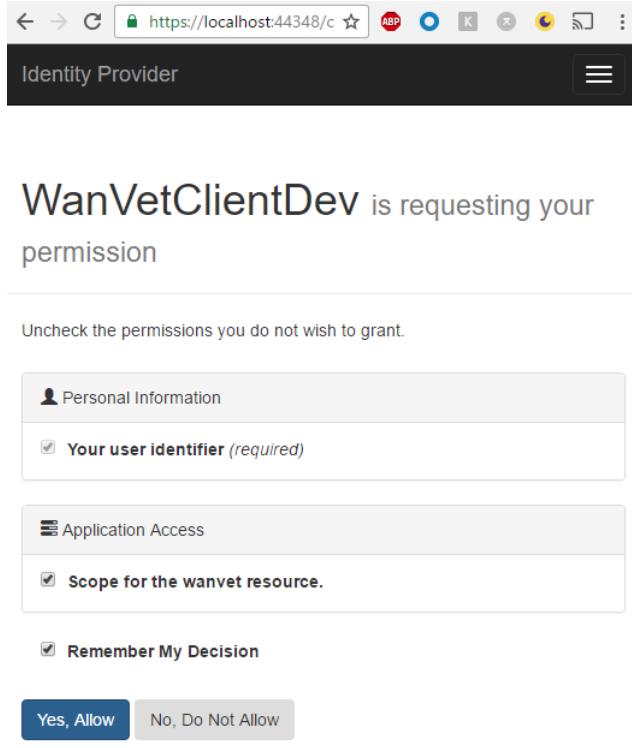


Figura 3.15: Consimțământul pentru accesul clientului la resursele *OpenID* și *wanvet*

După ce utilizatorul își dă acordul ca aplicația client să îi poată accesa diferite informații (care sunt bazate pe scopurile pe care aplicația client le-a cerut), acesta este redirectionat la adresa *https://localhost:44328* (unde rezidă aplicația client în mediul de dezvoltare). În momentul în care utilizatorul nu mai este autentificat (prin apăsarea butonului *Log off* sau în momentul în care acesta accesează o zonă pentru care nu are drepturi), el este redirectionat la adresa *https://localhost:44328/unauthorized*, iar datele acestuia sunt sterse din *localStorage*.

Mențiuni:

- În aplicația client a fost definită o rută către care vor fi redirectionați utilizatorii neautentificați
- În explicația anterioară a fost utilizată aplicația client din mediul de dezvoltare, care rulează folosind serverul web *IIS Express*. Pentru aplicația client din mediul de producție, a fost definit un alt client *OpenID Connect*, care are alt nume (*WanVetClient*), precum și alte adrese pentru redirectare, și care, după caz, ar putea avea acces la scopuri distințe.

Utilizatorul nu va putea fi autentificat dacă nu își dă acordul ca aplicația client specificată în partea de sus a paginii să-i folosească informațiile. În cazul în care acesta

își dă acordul, răspunsul primit de aplicația client conține proprietățile specificate în extrasul următor.

```
1 {
2   access_token: "aaecc318f9...",
3   expires_in: "3600",
4   id_token: "eyJhbGciOiJSUzI1NiIsI...",
5   scope: "wanvet%20openid",
6   session_state: "PErVrNw...",
7   state: "14820055154660.16826147042029205",
8   token_type: "Bearer"
9 }
```

Extras 34: Răspuns autentificare

În extrasul anterior, în cadrul proprietății *id_token* (care este codată în baza 64), se regăsesc următoarele componente: antet (informații despre tip, algoritm de encriptare), corp (informațiile despre utilizator), semnătură (semnătura criptografică). În cazul curent, suntem interesați de a doua componentă, care după decodare, arată astfel:

```
1 {
2 //...
3 // numele clientului
4 aud: "WanVetClientDev",
5 // data autentificării
6 auth_time: 1482005516,
7 email: "jenna.mclittle@gmail.com",
8 // data expirării acestui \textit{token}
9 exp: 1482005818,
10 family_name: "McLittle"
11 gender: "Female"
12 given_name: "Jenna",
13 //...
14 idp: "local",
15 // emitătorul acestui \textit{token}
16 iss: "https://localhost:44348",
17 //...
18 phone_number: "+40751223655",
19 // rolurile utilizatorului curent
20 role: ["wanvet.user"],
21 sid: "3381981538e621767e7eb3193461f449cd25853f5e8b7249eacee5e1426241bc",
22 sub: "2a4b8d45-3229-4e72-b76d-47dc0f260f83"
23 }
```

Extras 35: Informațiile din corp după decodare

În partea de început a subcapitolului aminteam posibilitatea configurării serviciilor în cadrul metodei *ConfigureServices* din clasa *Startup*. Astfel, a fost specificat un

serviciu folosit pentru descrierea profilului complet al unui utilizator (în cazul nostru: nume, prenume, număr de telefon, email, sex, și nu în ultimul rând, rolurile pe care acesta le va avea în cadrul clientului).

Aceste informații sunt folosite ca atare având în vedere că aplicația *IdentityProvider* garantează veridicitatea lor. În extrasul următor am exemplificat crearea profilului minimal al unui utilizator. Cu siguranță acest profil poate fi extins cu unele dintre următoarele proprietăți:

- O proprietate care să semnalizeze dacă utilizatorul și-a verificat email-ul, pentru a-i putea trimite notificări
- O proprietate care să semnalizeze dacă utilizatorul și-a verificat numărul de telefon (în același scop, pentru ca acesta să poată fi notificat)
- Setări importante ale utilizatorului precum: email secundar, dacă acesta a ales să folosească un nivel de securitate suplimentar [22]

```
1 public async Task GetProfileDataAsync(ProfileDataContext context)
2 {
3     var sub = context.Subject.GetSubjectId();
4     var user = await _userManager.FindByIdAsync(sub);
5     var principal = await _claimsFactory.CreateAsync(user);
6     var roles = await _userManager.GetRolesAsync(user);
7     var claims = principal.Claims.ToList();
8
9     claims.Add(new Claim(JwtClaimTypes.GivenName, user.Firstname));
10    claims.Add(new Claim(JwtClaimTypes.FamilyName, user.Lastname));
11    claims.Add(new Claim(JwtClaimTypes.PhoneNumber, user.PhoneNumber));
12    if (roles.Any(roleName => roleName == "wanvet.admin"))
13    {
14        claims.Add(new Claim(JwtClaimTypes.Role, "wanvet.admin"));
15    }
16    if (roles.Any(roleName => roleName == "wanvet.staff"))
17    {
18        claims.Add(new Claim(JwtClaimTypes.Role, "wanvet.staff"));
19    }
20    if (roles.Any(roleName => roleName == "wanvet.doctor"))
21    {
22        claims.Add(new Claim(JwtClaimTypes.Role, "wanvet.doctor"));
23    }
24    claims.Add(new Claim(JwtClaimTypes.Role, "wanvet.user"));
25    claims.Add(new Claim(StandardScopes.Email.Name, user.Email));
26    claims.Add(new Claim(JwtClaimTypes.Gender, (Gender)Enum.ToObject(
27                                typeof(Gender), user.Gender).GetDescription()));
28
29    context.IssuedClaims = claims;
30 }
```

Extras 36: Metoda *GetProfileDataAsync* - configurarea profilului

Toate aceste informații (*token*, roluri, profil) sunt stocate *client-side* în *localStorage* și au fost utilizate pentru a defini modul în care utilizatorul interacționează cu aplicația:

- Inițializarea profilului (poză de profil, sex, roluri)
- Afisarea/ascunderea diferitelor funcționalități (în funcție de rolul utilizatorului curent)
- Accesibilitatea la diferite zone ale produsului software (la nivel de rută). În Angular 2 a fost definit un nou concept (*guard*), cu ajutorul căruia putem defini verificări suplimentare. Aceste verificări vor fi validate în momentul în care utilizatorul dorește să părăsească ruta curentă, sau când acesta dorește să acceseze o altă rută. De exemplu, acest mecanism poate fi utilizat pentru a verifica dacă informațiile din formularul curent au fost salvate, înainte ca utilizatorul să părăsească această zonă
- Accesibilitatea la date (resursa *wanvet*). Fiecare cerere către API, indiferent de verbul utilizat (GET, POST, DELETE, etc.) va avea descris în antet tipul de autorizare însotit de un *token* (care a fost generat *a priori* de *IdentityProvider*).

```
1 private setHeaders(options: ApiGatewayOptions)
2 {
3     options.headers['Accept'] = 'application/json';
4     var token = this.authService.getToken();
5     if (token !== '') {
6         let tokenValue = 'Bearer ' + token;
7         options.headers['Authorization'] = tokenValue;
8     }
9 }
```

Extras 37: Metoda *setHeaders* utilizată în cadrul pregătirii oricărei cereri către API

Token-ul care va însobi fiecare cerere către API trebuie să fie un *token* valid, generat de aplicația utilizată pentru autentificarea utilizatorilor. În cadrul configurării clientului, a fost utilizată metoda *UseIdentityServerAuthentication* pentru a defini această modalitate de verificare în funcție de mediul în care va rula aplicația (dezvoltare: <https://localhost:44348/>, producție: <https://localhost/IdentityProvider>).

```

1 var identityServerValidationOptions = new IdentityServerAuthenticationOptions
2 {
3     Authority = env.IsDevelopment() ?
4         "https://localhost:44348/" : "https://localhost/IdentityProvider",
5     // Resursele la care va avea acces
6     AllowedScopes = new List<string> { "wanvet" },
7     ApiSecret = "wanvetSecret",
8     ApiName = "wanvet",
9     AutomaticAuthenticate = true,
10    SupportedTokens = SupportedTokens.Both,
11    AutomaticChallenge = true,
12 };
13 // Aplicația client va utiliza Identity Server cu opțiunile definite anterior
14 app.UseIdentityServerAuthentication(identityServerValidationOptions);

```

Extras 38: Protejarea API-ului - Validation Middleware

Există posibilitatea configurării unui *middleware* diferit sau a însiruirii mai multor validări în funcție de scenariul dat:

- Dacă avem de procesat o cerere API (am putea să verificăm utilizatorul curent procesând *token-ul* atașat) sau dacă cererea este pentru o pagină HTML (am putea să verificăm utilizatorul curent procesând *cookie-ul* atașat)
- Dacă dorim să diferențiem *middleware-ul* utilizat în funcție de mediul de rulare (dezvoltare, producție)

```

1 public static class AppBuilderExtensions
2 {
3     //https://github.com/aspnet-contrib
4     public static IApplicationBuilder UseWhen(this IApplicationBuilder app
5             , Func<HttpContext, bool> condition
6             , Action<IApplicationBuilder> configuration){/* */}
7 }
8 // configurarea clientului având următoarea formă
9 Func<HttpContext, bool> isApiRequest =
10     (HttpContext context) =>
11         context.Request.Path.ToString().StartsWith("/api/");
12 app.UseWhen(context => !isApiRequest(context), appBuilder =>
13 {
14     app.UseCookieAuthentication(options => {/* */})
15 }
16 app.UseWhen(context => isApiRequest(context), appBuilder =>
17 {
18     app.UseIdentityServerAuthentication(options => {/* */})
19 }

```

Extras 39: Configurare - Validation Middleware

Deoarece metodele de definire a politicilor de acces care erau utilizate până acum în cadrul aplicațiilor *Model-View-Controller* sau pentru securizarea *API-urilor*, și anume: *Simple Authorization*, *Role Based Authorization*, *Claims Based Authorization* nu puteau să cuprindă toate scenariile de modelare a accesibilității, în cadrul *framework-ului* ASP.NET Core au apărut noi metode: *Resource Based Authorization*, *Custom Policy-Based Authorization*.

Autorizarea bazată pe resurse (*Resource Based Authorization* [23]) poate fi utilizată dacă avem nevoie de date referitoare la resursa accesată pentru a decide dacă utilizatorul curent are acces sau nu la aceasta. Să presupunem că ne aflăm în scenariul în care un document poate fi modificat doar de utilizatorul care l-a creat. Pentru a decide dacă utilizatorul curent are dreptul să modifice un document, trebuie să identificăm care este utilizatorul care l-a creat, deci implicit trebuie să căutăm această informație în baza de date și să o încărcăm în memorie pentru verificare.

```

1 // politica de acces la pagina principală a produsului software
2 var guestPolicy = new AuthorizationPolicyBuilder()
3             .RequireAuthenticatedUser()
4             .RequireClaim("scope", "wanvet")
5             .Build();
6
7 // politicile de acces - API
8 services.AddAuthorization(options =>
9 {
10     options.AddPolicy("WanVetAdmin", policyAdmin =>
11     {
12         policyAdmin.RequireClaim("role", "wanvet.admin");
13     });
14     options.AddPolicy("WanVetUser", policyUser =>
15     {
16         policyUser.RequireClaim("role", "wanvet.user");
17     });
18     options.AddPolicy("WanVetStaff", policyStaff =>
19     {
20         policyStaff.RequireClaim("role", "wanvet.staff");
21     });
22     options.AddPolicy("WanVetDoctor", policyDoctor =>
23     {
24         policyDoctor.RequireClaim("role", "wanvet.doctor");
25     });
26 });
27
28 // înregistrarea filtrului de autorizare pentru pagina principală
29 services.AddMvc(options =>
30 {
31     options.Filters.Add(new AuthorizeFilter(guestPolicy));
32 })

```

Extras 40: Înregistrarea politicilor de acces

Cea de-a doua, *Custom Policy-Based Authorization*, păstrează proprietățile din metodele de definire a politicilor de acces anterioare, dar este o structură solidă care poate fi reutilizată (de exemplu: nu depinde de doar de rolurile definite) și care poate fi testată cu ușurință.

În extrasul anterior au fost definite următoarele politici de acces: *WanVetAdmin*, *WanVetUser*, *WanVetStaff*, *WanVetDoctor*. Acestea au fost utilizate pentru a putea defini o împărțire clară a informațiilor pe care le pot citi/modifica/șterge diferitele tipuri de utilizatori implicați.

Controller-ul de bază al *API-ului* (cel din care vor moșteni toate celelalte) a fost decorat cu următorul atribut: `[Authorize("WanVetUser")]` care descrie faptul că toate rutele descrise pot fi accesate doar de un utilizator care are rolul *wanvet.user* setat. Metoda *GetProfileDataAsync* prezentată anterior a fost implementată pentru a susține această abordare. De exemplu, dacă un utilizator are rolul de doctor, *token-ul* generat va conține următoarele două roluri: *wanvet.user* și *wanvet.doctor*.

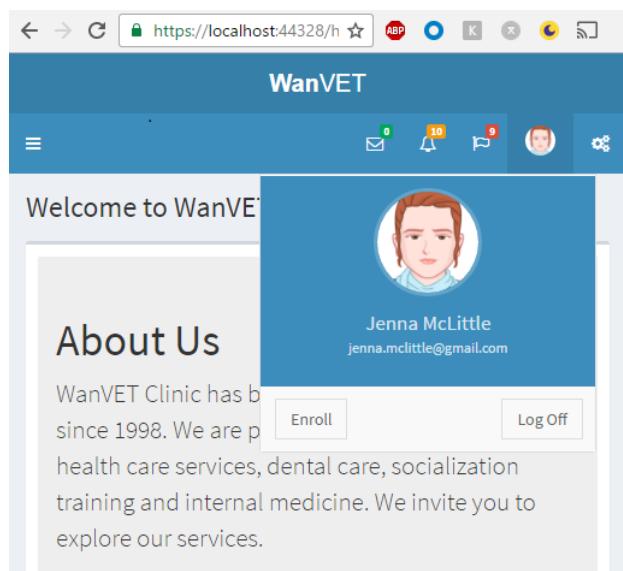


Figura 3.16: Pagina principală WanVet - utilizator autentificat

Așa cum aminteam anterior, *WanVet* este un produs software cu o singură pagină (*Single Page Application*), pagina principală fiind generată *server-side*, iar restul zonelor sunt generate *client-side*, utilizând informațiile pe care le primim cu ajutorul cererilor către *API*. În figura anterioară am prezentat pagina principală împreună cu zona în care este descris profilul utilizatorului, în contextul în care acesta este autentificat.

Utilizatorul, deși este autentificat, este nevoit să se înroleze în cadrul clinicii ve-

terinare pentru a avea acces la funcționalitățile definite: Adăugarea animalelor de companie, Programarea acestora pentru consultații/operații, Vizualizarea istoricului programărilor. Pentru a se înrola, în zona dedicată profilului există butonul *Enroll* care trimite o cerere de înregistrare a posesorului. Această operațiune se realizează o singură dată, manual, imediat după prima autentificare.

Pentru o mai bună înțelegere a politicilor de acces definite în cadrul produsului software *IdentityProvider* și utilizate pentru accesul la diferite resurse putem consulta proiectul pe GitHub. [24]

3.4 Pet Management

În cadrul produsului software, utilizatorul are posibilitatea de a adăuga un nou animal de companie prin accesarea modulului *Pets*, ramura *Add a new pet*. În cadrul formularului putem regăsi specificații precum specia, rasa din care face parte, și sexul acestuia.

The screenshot shows the 'Add Pet' form. At the top, there's a green button labeled 'SELECT PROFILE IMAGE!' and a red 'CLEAR' button. Below that is a placeholder text 'Drop your image here!' with a small 'X' icon. Underneath is a preview area showing a blue cat's face. The form has several input fields: 'Birth Date' (set to 11/09/2013), 'Name' (Magnus), 'Species' (radio button selected for 'Cat'), 'Sex' (radio button selected for 'Male'), 'Breed' (British Shorthair), and 'Color' (hex code #bcaeae). To the right of the color input is a color picker with a gradient from red to black.

Figura 3.17: Pagină pentru adăugarea unui animal de companie

De asemenea, avem posibilitatea de a completa culoarea dominantă a blănii și suntem nevoiți să încărcăm o fotografie de profil pentru o identificare cât mai precisă a animalului de companie. Ziua de naștere este orientativă, nu trebuie completată exact, dar poate fi utilizată pentru programarea etapelor de vaccinare.



Figura 3.18: Mesaj de succes - Adăugarea unui animal de companie

Accesând ramura *View existing pets* avem posibilitatea de a vedea o listă cu toate animalele de companie pe care le-am adăugat până în acest moment.

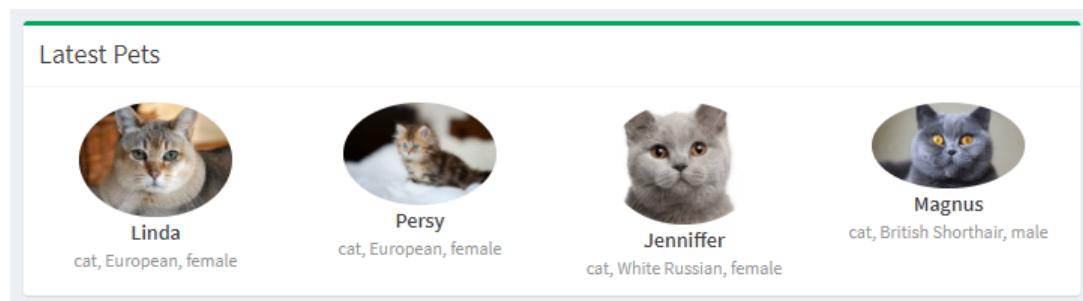


Figura 3.19: Lista curentă a animalelor de companie

3.5 Appointment Management

Pentru a extinde interacțiunea dintre doctori și detinătorii de animale de companie, am dezvoltat o zonă utilizată pentru managementul consultațiilor.

În linii mari, din punctul de vedere al detinătorului, fluxul este descris de următoarele reguli:

- Detinătorii de animale de companie au dreptul de a cere o consultăție avându-l ca doctor pe oricare doctor disponibil din clinică.
- Doctorii nu au dreptul de a refuza o consultăție, dar data și ora consultăției sunt validate pe baza calendarului intern înainte de a o înregistra.
- Data de început a consultăției se construiește pe baza celor două câmpuri (data, slot de timp).

Add Appointment

Pet
Linda x ▾

Date
09-22-2017 x

Doctor
Razvan Alex x ▾

Starting Time

12:30:00
13:00:00
13:30:00
14:00:00
14:30:00
15:00:00
15:30:00
16:00:00

Figura 3.20: Pagină pentru adăugarea unei noi consultații

- Dacă nu se mai pot face consultației pentru o anumită dată, deținătorul nu va avea posibilitatea de a alege un slot de timp.

Add Appointment

Pet
Jeniffer x ▾

Date
09-22-2017 x

Doctor
Razvan Alex x ▾

Starting Time

12:30:00
13:00:00
13:30:00
14:00:00
14:30:00
15:30:00
16:30:00
17:00:00

Figura 3.21: A doua încercare pentru adăugarea unei noi consultații

- Programările sunt generate utilizând calendarul intern (care momentan reprezintă orele de funcționare ale clinicii). În viitor această funcționalitate poate fi extinsă pentru fiecare doctor în parte prin adăugarea posibilității de configurare a calendarului personal.

- Data de final a consultației este generată automat ca fiind data de start plus 30 de minute.

În linii mari, din punctul de vedere al doctorului, fluxul este descris de următoarele reguli:

- Are posibilitatea de a revizui calendarul de lucru la începutul zilei sau oricând are posibilitatea.
- De asemenea, intervalul între consultații este de 30 de minute.
- La începutul consultației aceasta este într-o stare incipientă (*Open*). Doctorul are posibilitatea de a finaliza consultațiile (*Completed*) completând un formular ce conține simptomele confirmate și diagnosticul final.

The screenshot shows a user interface for managing an appointment. At the top, it says "Manage Appointment". Below that, there's a section for "Appointment" with a date and time set to "9/22/2017, 3:00:00 PM-Martin (Ionut Barbu)". There's also a small "X" icon and a dropdown arrow. Underneath is a "Diagnostic" section containing the text "Digestive upset with a clear loss of appetite. Tense abdomen.". Below that is a "Medical History" section with the text "Changes like withdrawing from contact with the family in order to sleep or overly clingy behavior.". At the bottom of the form is a green "Submit" button.

Figura 3.22: Pagină pentru finalizarea unei consultații

3.6 Calendar Management

Pentru a extinde interacțiunea doctorilor cu produsul software de care dispun, am dezvoltat un modul utilizat pentru managementul calendarului de lucru. Momentan avem posibilitatea de a regăsi și a vizualiza toate consultațiile efectuate, precum și cele viitoare.

- Colorația acestor două tipuri de consultații: (finalizate, încă nefinalizate) este diferită pentru a le putea distinge mai ușor (verde, albastru).

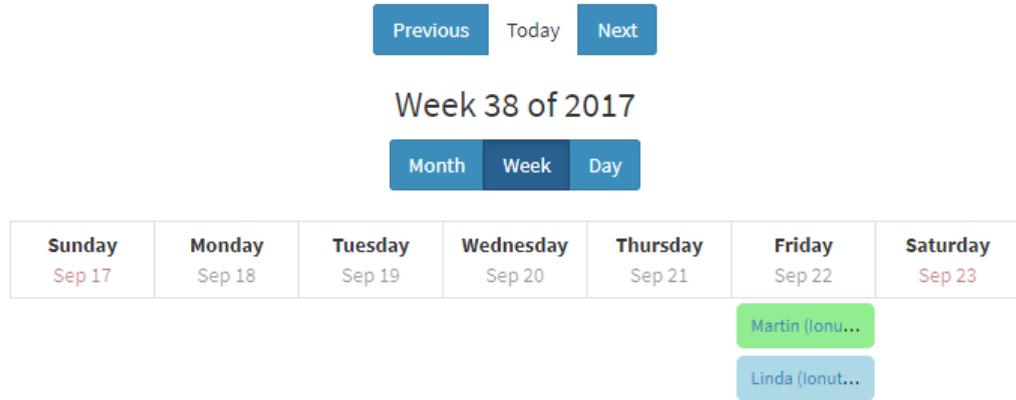


Figura 3.23: Vizualizarea calendarului - modul săptămână

- Există posibilitatea de a modifica modul de vizualizare - zi, lună, săptămână. Fiecare modificare a modului de vizualizare înseamnă o nouă căutare pentru a aduce consultațiile care se încadrează între datele specificate.

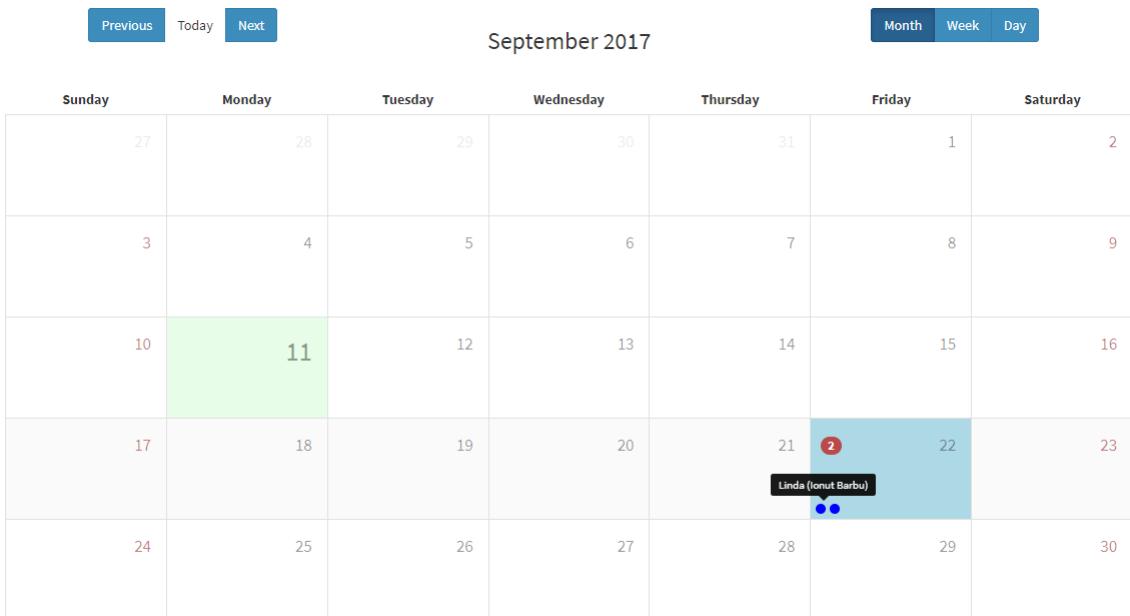


Figura 3.24: Vizualizarea calendarului - modul lună

În figurile anterioare putem revedea unele dintre consultațiile pe care le-am prezentat în cadrul subcapitolului 3.5.

3.7 Pet Timeline

Pentru a conchide fluxul de lucru prezentat în subcapitolele anterioare, am ales să prezint modulul dedicat istoricului medical al unui animal de companie. Această pagină este dezvoltată sub forma unui *timeline* în cadrul căruia putem regăsi atât consultațiile efectuate alături de concluziile medicale ale acestora, cât și consultațiile la care suntem așteptați pentru a ne prezenta.

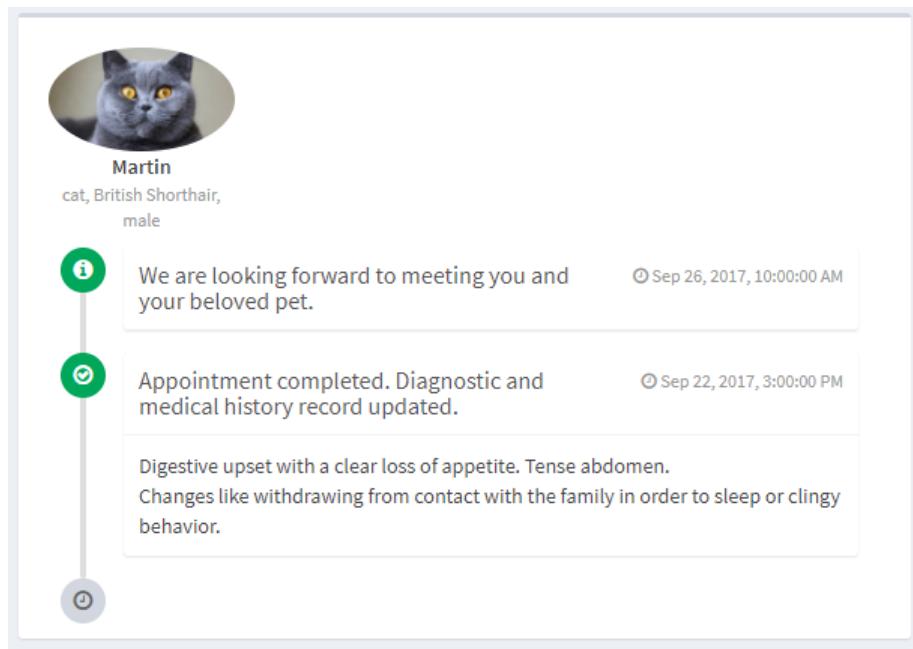


Figura 3.25: Istoricul medical al unui animal de companie

3.8 Dezvoltare ulterioară

Din punct de vedere tehnic, în momentul în care maturitatea produsului o va permite, zonele mele de interes ar fi următoarele:

1. Actualizarea versiunii .NET Core atât pentru *WanVet* Web API, cât și pentru serviciile dezvoltate (2.0).
2. Înlocuirea pachetului NuGet DasMulli.Win32.ServiceUtils cu Topshelf pentru evitarea viitoarelor neplăceri provocate de tratarea rudimentară a excepțiilor.
3. După cum aminteam, (*MemoryImage*) presupune printre altele existența unui mecanism de recreere a stării sistemului pe baza evenimentelor stocate. În cadrul produsului software pe care l-am dezvoltat acest mecanism nu există deoarece

am presupus că sistemul de replicare a datelor din Redis este suficient pentru o variantă inițială a produsului.

Din punct de vedere funcțional, în momentul în care maturitatea produsului o va permite, zonele mele de interes ar fi următoarele:

1. Adăugarea posibilității de configurare a calendarului pentru a suporta contractele cu jumătate de normă sau un program flexibil. Momentan doctorii sunt disponibili în programul clinicii.
2. Adăugarea posibilității de a genera facturi pentru fiecare consultație în parte.
3. Achitarea facturilor generate integrând o soluție externă pentru plata online.
4. Adăugarea posibilității de înștiințare în timp real a utilizatorului utilizând sistemul integrat de notificări.

Capitolul 4

Concluzii

Teza curentă a folosit ca sursă de inspirație prezentarea lui Eric Evans din cadrul conferințelor GOTO 2015 *DDD & Microservices: At Last, Some Boundaries!*. Aplicarea strategiei DDD în cadrul unui produs de mare anvergură este o decizie grea și deseori evitată de marile corporații.

În ultimii ani, marii actori din piața de IT au încercat o schimbare bruscă a modalităților de mențenanță a produselor software pe care le aveau în portofoliu. Această schimbare a presupus izolarea unui set de funcționalități de bază în cadrul unui monolit și construirea de servicii satelit pentru descrierea noilor funcționalități. Având în vedere că aceste funcționalități sunt deseori descrise în cadrul unui context separat (*Bounded Context*), apariția serviciilor a presupus relansarea în currențul de gândire predominant a principiilor care stau la baza strategiei DDD.

Acum este mai ușor ca niciodată să dezvoltăm servicii care să se mapeze clar la un anumit context, comunicarea dintre contexte fiind favorizată de apariția diferenților agenții de intermediere a mesajelor. Avem posibilitatea de a alege un astfel de agent în funcție de importanță pe care dorim să aibă în cadrul produsului software pe care îl descriem. Într-un mod surprinzător, ultimele tendințe din zona *enterprise* ne arată disponibilitatea marilor corporații de a accepta și de a se implica în dezvoltarea agenților de intermediere a mesajelor care expun doar funcționalități de bază, un exemplu fiind succesul MercadoLibre (NASDAQ: MELI) cu agentul RabbitMQ.

După cum afirma și Eric Evans, dorința de schimbare vine și din faptul că a fost general acceptată ideea conform căreia nu toate produsele de mare anvergură sunt bine proiectate. Chiar și cele care se bazează pe microservicii pot să aibă greșeli grave de proiectare, dar având în vedere posibilitățile extinse de versiune și actualizare a

acestora, pot fi remediate rapid, ușor, în același timp minimizând impactul modificărilor asupra întregului produs.

Încercarea de a proiecta o asemenea arhitectură pe cont propriu a avut un impact masiv asupra modului în care privesc adoptarea noilor practici tehnologice. Deși până în acest moment eram familiar cu şablonul CQRS, nu avusesem ocazia de a separa modalitatea de stocare și proiectare a datelor în funcție de scopul în care acestea sunt utilizate (*Command Stack* - validare și verificare, *Query Stack* - proiectare, citire și afișare). De asemenea, din punct de vedere arhitectural, separarea domeniului în multiple microservicii utilizându-le pe acestea pentru a declara granițele contextelor identificate în momentul aplicării strategiei DDD, a fost un experiment nou și care mi-a alimentat interesul pentru a încerca o asemenea separare în cadrul unui produs software real.

În încheiere, aş dori să menționez una dintre ideile expuse de Bilgin Ibryam la sfârșitul anului 2016. Aceasta identificase trei grupuri distincte din punct de vedere organizațional împreună cu rolurile pe care le au acestea în procesul tehnologic: *Pioneers* (cei care au posibilitatea de a explora și de a experimenta), *Settlers* (cei care posedă îndemânarea necesară de a transforma un prototip, fie el și arhitectural, în ceva concret care poate fi utilizat la scară mare), *Town planners* (cei care se ocupă de industrializare). În cadrul microserviciilor, era în care Netflix explora a trecut, era în care cei de la Spring ajutau firmele mari să accepte aceste tehnologii este pe sfârșite, iar în piață apar giganții Amazon, Google, și Microsoft transformând acest şablon arhitectural în standardul viitorului.

Bibliografie

- [1] Kamil Lelonek. How micromanagement kills creativity and productivity of developers, February 2016. URL <https://blog.lelonnek.me/how-micromanagement-kills-creativity-and-productivity-of-developers-c40a2bd5eb68>. Citat în data de 8 Aprilie 2017.
- [2] Arun Gupta & Vineet Reynolds. Getting Started with Microservices, 2015. URL <https://dzone.com/refcardz/getting-started-with-microservices>. Citat în data de 14 Aprilie 2017.
- [3] Neal Ford. Microservices as an Evolutionary Architecture, March 2016. URL <https://www.thoughtworks.com/insights/blog/microservices-evolutionary-architecture>. Citat în data de 14 Aprilie 2017.
- [4] Greg Young. CQRS Documents, 2010. URL https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf. Citat în data de 29 Aprilie 2017.
- [5] Alexandru-Răzvan Dumitru. Desing patterns for ASP.NET applications, June 2014. URL <https://github.com/RazvanADumitru/Hesira/blob/master/Razvan%20Dumitru%20-%20Desing%20patterns%20for%20ASP.NET%20applications.pdf>. Citat în data de 14 Aprilie 2017.
- [6] Dino Esposito. Modern Software Architecture: Domain Models, CQRS, and Event Sourcing, 2015. URL <https://app.pluralsight.com/library/courses/modern-software-architecture-domain-models-cqrs-event-sourcing/table-of-contents>. Citat în data de 29 Aprilie 2017.
- [7] Bertrand Meyer. Eiffel: a language for software engineering, 2012. URL http://laser.inf.ethz.ch/2012/slides/Meyer/eiffel_laser_2012.pdf. Citat în data de 29 Aprilie 2017.

- [8] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003. Citat în data de 24 Iulie 2017.
- [9] Joseph W. Foote, Brian & Yoder. Big ball of mud. In B. & Rohnert H. Harrison, N. & Foote, editor, *Pattern Languages of Program Design*, volume 4, pages 654–692. 2000. URL <http://www.laputan.org/mud/mud.html>. Citat în data de 24 Iulie 2017.
- [10] Vaughn Vernon. *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013. Citat în data de 24 Iulie 2017.
- [11] Grzegorz Ziemoński. Onion Architecture Is Interesting, 2017. URL <https://dzone.com/articles/onion-architecture-is-interesting>. Citat în data de 17 August 2017.
- [12] Martin Fowler. Memoryimage, 2011. URL <https://martinfowler.com/bliki/MemoryImage.html>. Citat în data de 30 August 2017.
- [13] James Lewis & Martin Fowler. Microservices. A definition of this new architectural term, 2014. URL <https://martinfowler.com/articles/microservices.html>. Citat în data de 24 August 2017.
- [14] Alan Morrison. What is microservices architecture? Think ant colonies, beehives, or termite mounds, 2014. URL <http://usblogs.pwc.com/emerging-technology/what-is-microservices-architecture-think-ant-colonies-beehives-or-termite-mounds/>. Citat în data de 24 August 2017.
- [15] Community AppCentrica. THE RISE OF MICROSERVICES, 2016. URL <https://www.appcentrica.com/the-rise-of-microservices/>. Citat în data de 24 August 2017.
- [16] Caoilte O'Connor. Domain Service Aggregators: A Structured Approach to Microservice Composition, 2015. URL <https://www.youtube.com/watch?v=iS6p-SWVuac>. Citat în data de 24 August 2017.
- [17] Community. Architecture Overview, 2017. URL <https://angular.io/guide/architecture>. Citat în data de 24 August 2017.

- [18] TechEd Europe. Explore web development with Microsoft ASP.NET Core 1.0, November 2016. URL <https://www.youtube.com/watch?v=gF4LR67PLiI>. Citat în data de 6 Ianuarie 2017.
- [19] Rick Blaisdell. What are main differences (in implementation) between cloud aware application and normal web application?, November 2013. URL <https://www.quora.com/What-are-main-differences-in-implementation-between-cloud-aware-application-and-normal-web-application/answer/Rick-Blaisdell?sr&id=JEs9>. Citat în data de 6 Ianuarie 2017.
- [20] Matt Weinberger. Contain yourself: The layman's guide to Docker, November 2014. URL <http://www.computerworld.com/article/2849619/contain-yourself-the-laymans-guide-to-docker.html>. Citat în data de 6 Ianuarie 2017.
- [21] Dru Sellers Chris Patterson and Travis Smith. What does MassTransit add to RabbitMQ?, 2015. URL <http://docs.masstransit-project.com/en/latest/overview/valueadd.html>. Citat în data de 10 August 2017.
- [22] Echipa Google Security. Google 2-Step Verification, 2016. URL <https://www.google.com/landing/2step/>. Citat în data de 18 Decembrie 2016.
- [23] Rick Anderson & Luke Latham & alții. Resource Based Authorization, October 2016. URL <https://docs.microsoft.com/en-us/aspnet/core/security/authorization/resourcebased>. Citat în data de 28 Decembrie 2016.
- [24] Alexandru-Răzvan Dumitru. Identityprovider.WV, 2017. URL <https://github.com/RazvanADumitru/IdentityProvider.WV>. Citat în data de 24 Iulie 2017.