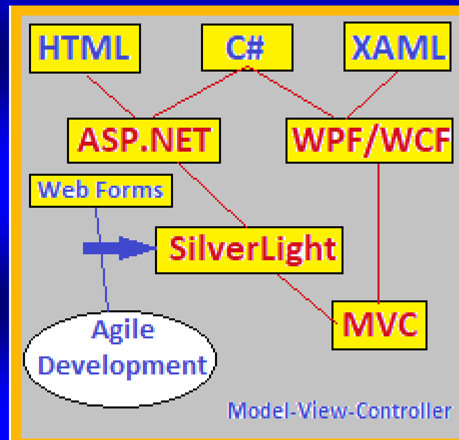
გია სურგულაძე, თალიკო ჟვანია,
ნინო კივილაძე, სოფიკო ქობულაშვილი

WEB-აპლიკაციის აგების ტექნოლოგია მიკროსერვისული არქიტექტურით



სტუ-ს „IT კონსალტინგის სამეცნიერო ცენტრი"

საქართველოს ტექნიკური
უნივერსიტეტი
1922 წლიდან
GEORGIAN TECHNICAL
UNIVERSITY
SINCE 1922

გია სურგულაძე, თალიკო ქვანია,
ნინო კივილაძე, სოფიკო ქობულაშვილი

Web-აპლიკაციის აგების ტექნოლოგია მიკროსერვისული არქიტექტურით

დამტკიცებულია:
სტუ-ს „IT კონსალტინგის სამეცნიერო ცენტრის“ სარედაქციო კოლეგიის მიერ - ოქმი N5 2.02.24

თბილისი - 2024

უაკ 004.5

განხილულია თანამედროვე დესკტოპ- და ვებ-ტექნოლოგიები და მათი გამოყენების გზები კორპორაციული მენეჯმენტის ბიზნესაპლიკაციების და მიკროსერვისული აპლიკაციების შემუშავების მიზნით. გაანალიზებულია ASP.NET Web Forms, Silverlight, MVC დაპროგრამების პლატფორმები, მოყვანილია მათი მახასიათებლების შედარება. გამოკვლეულია პროგრამულ-ლი უზრუნველყოფის ინფრასტრუქტურის მნიშვნელობა რეალური ამოცანების გადასაჭრელად, კერძოდ, წარმოდგენილია სუფთა არქიტექ-ტურის (CA) და დომენორიენტირებული დიზაინის (DDD) მიდგომები. მათ საფუძველზე ადამიანური რესურსების მართვისა და მიკროსერვისული აპლიკაციების აგების რეალური სისტემები. პროგრამული რეალიზაცია განხორციელებულია მაიკროსოფთის Visual Studio.NET Framework პლატფორმაზე, ASP.NET, SilverLight, WPF/WCF/WF ტექნოლოგიებით, C# და XAML ენების საფუძველზე, აგრეთვე მონაცემთა საცავების, რელაციური და NoSQL ბაზების გამოყენებით. მონოგრაფია განკუთვნილია ICT სფეროს პროგრამული ინჟინერიის კონცენტრაციის სტუდენტებისა და მართვის საინფორმაციო სისტემების პროგრამისტ-დეველოპერებისათვის. აგრეთვე, საწარმოო პროცესების ავტომატიზაციით დაინტერესებული მკითხველისა და დოქტორანტებისათვის.

რეცენზენტები:

- პროფ. გ. გოგიჩაიშვილი - საქ. მეცნიერებათა ეროვნული აკადემიის წევრ-კორესპოდენტი, ტ.მ.დ.
- ი. ბულია – ინფორმატიკის აკად.დოქტორი (TBC ბანკი)

რედკოლეგია:

ა. ფრანგიშვილი (თავმჯდომარე), ზ. აზმაიფარაშვილი, მ. ახობაძე, გ. გოგიჩაიშვილი, ზ. ბოსიკაშვილი, ე. თურქია, რ. კაკუბავა, თ. ლომინაძე, ნ. ლომინაძე, პეტრიაშვილი, გ. სურგულაძე. ბ. შანშიაშვილი, ა. ცინცაძე, ო. შონია, ზ. წვერაიძე

© სტუ-ს „IT კონსალტინგის სამეცნიერო ცენტრი", 2024
ISBN 978-9941-8-6333-2

ყველა უფლება დაცულია, ამ წიგნის არც ერთი ნაწილი (იქნება ეს ტექსტი, ფოტო, ილუსტრაცია თუ სხვა) გამოყენება არც ერთი ფორმით და საშუალებით (ელექტრონული თუ მექანიკური), არ შეიძლება გამომცემლის წერილობითიი ნებართვის გარეშე. საავტორო უფლებების დარღვევა ისჯება კანონით.

Georgian Technical University

Surguladze Gia, Zhvania Taliko,
Kiviladze Nino, Kobulashvili Sopiko

WEB-APPLICATION BUILDING TECHNOLOGY WITH MICROSERVICE ARCHITECTURE

Modern desktop and web technologies and ways of using them to develop corporate management business applications and microservices applications are discussed. ASP.NET Web Forms, Silverlight, MVC programming platforms are analyzed. A comparison of their characteristics is given. The importance of software infrastructure for solving real problems is investigated, in particular, pure architecture and domain-oriented design (DDD) approaches are presented. Real systems for building human resource management and microservice applications based on them. Software implementation is implemented on the Microsoft Visual Studio.NET Framework platform, with ASP.NET, SilverLight, WPF/WCF/WF technologies, based on C# and XAML languages, as well as using data stores, relational and NoSQL databases. The monograph is intended for students of software engineering concentration in the field of ICT and programmer-developers of management information systems. Also, for readers and doctoral students interested in the automation of production processes.

© „IT-Consulting scientific center" of GTU, Tbilisi, 2024
ISBN 978-9941-8-6333-2

ავტორთა შესახებ:

გია სურგულაძე – სტუ-ის ინფორმატიკის ფაკ-ის „პროგრამული ინჟინერიის“ დეპარტამენტის უფროსი, პროფესორი. ტექნიკის მეცნიერებათა დოქტორი (1993), გაეროსთან არსებული „ინფორმატიზაციის საერთაშორისო აკადემიის“ ნამდვილი წევრი (1994, IIA). 100 წიგნის ავტორი (მათ შორის 27 მონოგრაფია, 20 სახელმძღვანელო, 53 დამხმარე და მეთოდსახელმძღვანელო). 300-ზე მეტი სამეცნიერო ნაშრომის ავტორი საინფორმაციო სისტემების პროგრამული ინჟინერიასა და ინფორმატიკის დიდაქტიკაში. სტუ-ის „ინფორმატიკის“ სადოქტორო პროგრამის ხელმძღვანელი. 50 სადოქტორო დისერტაციის და 80 მაგისტრის ხელმძღვანელი. მსოფლიო ბანკის, USAID-ის და NATO-ს პროექტების მონაწილე. *წვლილი:* შეიმუშავა საინფორმაციო სისტემების ჰიბრიდული დეველოპმენტის მეთოდოლოგია სერვის-ორიენტირებული არქიტექტურის ბაზაზე.

თალიკო ქვანია – სტუ-ის ინფორმატიკის ფაკულტეტის დეკანი, „პროგრამული ინჟინერიის“ დეპარტამენტის პროფესორი. სტუ-ის საგამოცდო და ელექტრონული სასწავლო რესურსების ცენტრის უფროსი (2021-2024). ტექნიკის მეცნიერებათა კანდიდატი (2005). სამეცნიერო გრანტების, სტაჟირების და კონფერენციების აქტიური მონაწილე ევროპის მრავალი ქვეყნის უნივერსიტეტში. 100-ზე მეტი სამეცნიერო სტატიის ავტორი. მისი ხელმძღვანელობით დაცულია სადოქტორო დისერტაციები და სამაგისტრო ნაშრომები. *წვლილი:* წინამდებარე ნაშრომში შეიმუშავა სერვის-ორიენტირებილი არქიტექტურის სისტემისთვის მონაცემთა საცავის აგების მეთოდი და მდგრადი მოდელები.

ნინო კივილაძე – აკადემიური დოქტორი ინფორმატიკაში (2016). ივ. ჯავახიშვილის თსუ-ის ბაკალავრი და მაგისტრი, სტუ-ს „ინფორმატიკის“ პროგრამის დოქტორანტი (2013-2016, მეცნ. ხელმძღ, გ. სურგულაძე). 2 მონოგრაფიის, 7 სამეცნიერო სტატიის ავტორი. ფინანსთა სამინისტროს და TBC ბანკის პროგრამისტ-დეველოპერი. *წვლილი:* საჯარო ორგანიზაციების საწარმოო პროცესების სრულყოფა ვებ-აპლიკაციებით SilverLight ფრეიმვორკის, MVC ტექნოლოგიის და SOA არქიტექტურით.

სოფიკო ქობულაშვილი – აკადემიური დოქტორი ინფორმატიკაში (2022). ივ. ჯავახიშვილის თსუ-ის ბაკალავრი და მაგისტრი, სტუ-ს „ინფორმატიკის“ პროგრამის დოქტორანტი (2019-2022, მეცნ. ხელმძღ, გ. სურგულაძე). წვლილი: შეიმუშავა დისტრიბუციული სისტემების არქიტექტურის დაგეგმარების მეთოდოლოგია მცირე და საშუალო ბიზნესის ობიექტებისათვის მიკროსერვისების ბაზაზე.

შინაარსი

შესავალი 11

თავი 1. ბიზნესაპლიკაციების Web-დაპროგრამების ტექნოლოგიების კრიტიკული ანალიზი 21

1.1. ბიზნესაპლიკაციები და ადამიანური რესურსების მართვის სისტემა 21

1.2. ASP.NET ტექნოლოგიის განვითარების ისტორია 23

1.3. რა ნაკლი აქვს ASP.NET Web Forms ტექნოლოგიას ? 26

1.4. ASP.NET MVC ტექნოლოგია 29

1.5. ვებ-დაპროგრამების ინსტრუმენტი Silverlight 37

1.6. RIA აპლიკაციებით ამოცანის გადაჭრა 42

1.7. Silverlight ტექნოლოგიის უპირატესობები 45

1.8. MVVM არქიტექტურული სტანდარტი Silverlight აპლიკაციებში 49

1.9. View და ViewModel დონეების ურთიერთკავშირი 53

1.10. შედარებები პოპულარულ პლატფორმებს შორის (ASP.NET Web Forms, Silverlight, ASP.NET MVC) 55

1.11. ამოცანის დასმა: HRMS with SOA 59

1.12. პირველი თავის დასკვნა 63

თავი 2. პროგრამული აპლიკაციის არქიტექტურის დაპროექტების და აგების მეთოდოლოგია 65

2.1. პროგრამული აპლიკაციის არქიტექტურა 65

2.2. პროგრამული აპლიკაციის არქიტექტურის პრინციპები და დაგეგმვის პროცესი 68

2.3. 3- და N-დონიანი არქიტექტურული სტილები 75

2.4. პროგრამული აპლკიკაცია ლოგიკური დონეებით (Layers) 78
2.5. არქიტექტურის დაგეგმვის საფეხურები 83
2.6. RIA აპლიკაციების დაპროექტება 87
2.7. RIA დიზიანის ზოგადი რეკომენდაციები 91
2.8. RIA არქიტექტურის სპეციფიკა 94
2.9. RIA აპლიკაციების შესაბამისი არქიტექტურული სტანდარტები 103
2.10. არქიტექტურული სტანდარტი Command 105
2.11. პრეზენტაციის დონე. კომუნიკაცია View და ViewModel დონეებს შორის 113
2.12. ანიმაციები Silverlight ტექნოლოგიაში 115
2.13. ვებაპლიკაციის რეპორტების ინტეგრაციის მეთოდები. 123
2.14. სერვერული და ლოკალურად ჩაშენებული რეპორტების შედარება 127
2.15. ანგარიშგებების გენერაცია მომხმარებლისთვის ადვილად გამოყენებადი ინტერფეისიდან 131
2.16. მეორე თავის დასკვნა 136
თავი 3. ადამიანური რესურსების მართვის სისტემა 137
3.1. არქიტექტურა და იმპლემენტაცია 137
3.2. კადრების მართვის ერთიანი ელექტრონული სისტემა საჯარო სტრუქტურებში 138
3.3. ელექტრონული მართვის სისტემის მოდულები: თანამშრომლის პირადი ბარათი 141
3.4. მონაცემთა სათავსოსთან წვდომა 147

3.5. ბიზნეს-ლოგიკის დონე 151
3.6. სერვისის დონე 157
3.7. Silverlight-ით მდიდარი და ინტერაქტიული სამომხმარებლო ინტერფეისის იმპლემენტაცია 162
3.8. Silverlight Value Converter ობიექტი 175
3.9. ICommand ინტერფეისის რეალიზაცია კალენდრის მაგალითზე 181
3.10. მესამე თავის დასკვნა 183
თავი 4. მონოლითური და მიკროსერვისული არქიტექტურების მიმოხილვა 185
4.1. გავრცელებული არქიტექტურული მიდგომები 185
4.1.1. მონოლითური არქიტექტურა 186
4.1.2. მიკროსერვისული არქიტექტურა 190
თავი 5. მცირე და საშუალო ბიზნესის მართვის ციფრული პლატფორმის არქიტექტურული მოდელი 199
5.1. სისტემის სერვისის სახით მიწოდების უპირატესობა ... 200
5.2. სისტემის საბაზისო პაკეტის ფუნქციონალის აღწერა 201
5.3. პროდუქციის ელექტრონული შესყიდვის ბიზნესპროცესი 203
5.3.1. ონლაინ გაყიდვების ბიზნესპროცესში მონაწილე მიკროსერვისები 204
5.3.2. პროდუქციის ელექტრონულად შესყიდვის ბიზნეს-პროცესის ბიჯები ……………………………………………... 205
5.4. სისტემის ზოგადი ტექნიკური აღწერა …………………. 206

5.5. სისტემის მომხმარებლის პორტალის არქიტექტურული მოდელი 208
5.6. დისტრიბუციული ტრანზაქციის მართვა 210
5.6.1. ქორეორგაფიაზე დაფუძნებული საგა 213
5.6.2. ორკესტრირებაზე დაფუძნებული საგა 214
5.6.3. დისტრიბუციული ტრანზაქციის კომპენსაცია (Eventual Consistency) 220
5.7. მეხუთე თავის დასკვნა 222
თავი 6. დისტრიბუციული სისტემის მიკროსერვისები და მათი შემადგენელი კომპონენტები 224
6.1. მიკროსერვისების ტექნიკური აღწერა 224
6.1.1. სუფთა არქიტექტურა (CA) 227
6.1.2. CAP თეორემა 230
6.1.3. Api Gateway სერვისი 231
6.1.4. მიკროსერვისის შემადგენელი კომპონენტები 233
6.2. პროდუქციის შესყიდვის პროცესში მონაწილე მიკროსერვისების აღწერა (მომხმარებლის პორტალი) 238
6.2.1. პროდუქტების მიკროსერვისი 238
6.2.2. შეკვეთების მიკროსერვისი 244
6.2.3. მომხმარებლების მართვის მიკროსერვისი 245
6.2.4. გადახდების მიკროსერვისი 248
6.2.5. ადგილზე მიტანის მომსახურების მიკროსერვისი 250
6.2.6. შეტყობინებათა მიკროსერვისი 252
6.2.7. მარაგების მართვის მიკროსერვისი 253
6.2.8. შეთავაზებების მიკროსერვისი 254

6.3. მეექვსე თავის დასკვნა 255

თავი 7. მიკროსერვისული არქიტექტურის მონაცემთა საცავის ჰორიზონტალური მასშტაბირება და სისტემის ადმინისტრირება 256

7.1. დეცენტრალიზებული მონაცემთა მართვა 256

7.2. კლიენტი კომპანიის მონაცემთა საცავის იდენტიფიცირების პროცესი 264

7.3. სერვისების ჰორიზონტალური მასშტაბირება მაღალი დატვირთულობის პირობებში 266

7.4. ადმინისტრირების მოდული 271

7.4.1. მიმწოდებლის ადმინისტრირების მოდული 272

7.4.1.1. კლიენტი კომპანიის რეგისტრაციის პროცესი 272

7.4.1.2. კლიენტი კომპანიის ადმინისტრირების მოდული... 277

7.4.2.1. მომხმარებელი კომპანიების ადმინისტრირების მოდულის არქიტექტურული მოდელი 278

7.4.2.2. რეპორტინგის ფუნქციონალი 279

7.5. მეშვიდე თავის დასკვნა 284

დასკვნა 286

გამოყენებული ლიტერატურა 288

გამოყენებული აბრევიატურები

AJAX – Asynchronous JavaScript and XML
API – Application Programming Interface
ASP – Active Server Pages
BFF – Backend For Frontend
COM – Component Object Model
CQRS – Command-Query Responsibility Segregation
CSS – Cascading Style Sheets
DDD – Domain Driven Design
DLL – Dynamic Link Library
DOM – Document Object Model
ERP – Enterprise Resource Planning
GUI – Graphical User Interface
HRMS – Human Resources Management System
HTTP – Hyper-Text Transfer Protocol
IDE – Integrated development environment
LINQ – Language Integrated Query
MVVM – Model View Viewmodel
MVC – Model View Controller
RIA – Rich Internet Application
OWASP – Open Worldwide Application Security Project
SDK – Software Development Kit
SQL – Structured Query Language
SSMS – SQL Server Management Studio
SSRS – SQL Server Reporting Services
T-SQL – Transact-Structured Query Language
URL – Uniform Resource Locator
VPN – Virtual Private Network
WCF – Windows Communication Foundation
WF – Work Flow
WPF – Windows Prezentation Foundation
XAML – Extensible Application Markup Language
XML – Extensible Markup Language

შესავალი

ორგანიზაციული მართვის (მენეჯმენტის) ბიზნეს-პროცესების მხარდაჭერა ახალი ციფრული ტექნოლოგიებით, დიდი მონაცემების ანალიზისა და მოდელირების სისტემებით ხელოვნური ინტელექტის მეთოდების საფუძველზე ინფორმაციული და კომუნიკაციური ტექნოლოგიების (ICT) სფეროს ძირითადი გამოწვევაა [1-5].

21-ე საუკუნის დასაწყისიდან განსაკუთრებით მოიმატა პროგრამული აპლიკაციების შექმნის და გამოყენების ტენდენციამ დესკტოპ- და ვებ- (ინტერნეტული) სისტემების ბაზაზე. კორპორაციული ვებ-აპლიკაციები ედება საფუძვლად „ელექტრონული მთავრობის“ და „ელექტრონული ბიზნესის“ სისტემებს. ამასთან ერთად, პროგრამული უზრუნველყოფის რეალიზაციისა და დანერგვის მიზნით შეინიშნება ვებ-აპლიკაციების გამოყენების მკვეთრი ზრდის ტენდენცია. ამ სფეროში დღეისათვის ერთ-ერთი მნიშვნელოვანი გამოწვევაა *სისტემების დაპროექტების სუფთა არქიტექტურის მეთოდოლოგიის კვლევა და მოქნილი მიკროსერვისული აპლიკაციების აგების ტექნოლოგიის შემუშავება.*

დესკტოპ-აპლიკაციებისგან განსხვავებით ვებ-აპლიკაციები მოსახერხებელია შემდეგი ფაქტორების გამო [6-8].

- არ საჭიროებს გადმოწერას და ინსტალირებას;
- არ საჭიროებს განახლებებს;
- იძლევა სიტემის გამოყენებადობის ანალიზის შედეგებს;
- ნაკლები ხარჯი.

ნაშრომის პრაქტიკულ ნაწილში წარმოდგენილია ადამიანური რესურსების მართვის ელეტრონული სისტემის იმპლემენტირება ვებ-გარემოში. ადამიანური რესურსების მართვის სისტემა არის პროგრამული უზრუნველყოფა, რომელიც მართავს ადამიანურ კაპიტალთან დაკავშირებულ ბიზნეს-პროცესებს. იგი ერთიანი, კომპლექსური სისტემაა, რომელსაც იყენებს HR მენეჯერი გადაწვეტილებების მიღე-ბისას.

HR-ის ძირითადი ნაწილი მოიცავს ორგანიზაციებში მომუშავე თანამშრომლების შესახებ საბაზისო ინფორმაციას. იგი აერთიანებს პერსონალურ მონაცემებს, როგორიცაა *თანამშრომლის მისამართი, დაბადების თარიღი, ოჯახური მდგომარეობა, საკონტაქტო ინფორმაცია; სახელფასო უწყისის მონაცემები; დანამატები, პრემია, ჯილდო, წახალისება; სამუშაოს აღწერილობა; ორგანიზაციული სტრუქტურა; თანამშრომლის პორტალი და self-service მოდული* და ა.შ.

ნაშრომის მიზანია მსხვილი ბიზნეს-პროცესების სამართავად სერვისებზე და მიკროსერვისებზე ორიენტი-რებული ვებ-აპლიკაციების შემუშავების მეთოდოლოგიის შემუშავება.

განხილულია პრგორამული დანართის არქიტექტურის ძირიადი პრინციპები და მიდგომები. გამოვლენილია Silverlight ვებ-აპლიკაციების შემუშავებისას ჩასატარებელი სამუშაოები [9-11].

დასმული მიზნის მისაღწევად (ადამიანური რესურ-სების მართვის ელექტრონული სისტემა HRMS) გადაწყვე-ტილია მსხვილი ბიზნეს აპლიკაციების იმპლემენტირება ვებ-

აპლიკაციის სახით Silverlight ტექნოლოგიის გამოყენებით. აუცილებელია შემდეგი ძირითადი ამოცანების გადაწყვეტა:

- სერვისებზე ორიენტირებული პროგრამული უზრუნველყოფის შემუშავებისათვის საჭირო დაპროგრამების პლატფორმის და ინსტრუმენტების გამოვლენა;

- Silverlight აპლიკაციების შემუშავების დაგეგმვა და RIA (Rich Internet Application) აპლიკაციებთან საუშაო ეფექტური არქიტექტურული სტანდარტების გამოვლენა [12,13];

- პროგრამული უზრუნველყოფის ლოგიკურ კომპონენტებად დაყოფა და ერთმანეთისგან გამიჯვნა დამოუკიდებელ ლოგიკურ შრეებში (Layer);

- ლოგიკური კომპონენტების ერთმანეთთან დაკავშირება და კომუნიკაციის აწყობა, ისე, რომ მათ შორის თანაკვეთის წერტილების მინიმალური რაოდენობა მივიღოთ. პროგრამული დანართის თითოეული კომპონენტი ან მოდული პასუხისმგებელი უნდა იყოს მხოლოდ ერთ კონკრეტულ ფუნქციონალზე;

- თეორიულად შემუშავებული არქიტექტურული მოდელების და მეთოდების პროგრამული რეალიზაცია Microsoft VisualStudio.NET Framework პროგრამულ გარემოში, C# ენაზე, SQL Server, WCF, Silverlight, SQL Server Reporting Services პროგრამული პაკეტების გამოყენებით [1,11,14]. ინტერაქტიული სამომხმარებლო გარემოს შემუშავება, რომელშიც ავტორიზებულ მომხმარებელს შეეძლება მუშაობა, ინფორმაციის შეტანა და ანგარიშგებების ამოღება;

- Silverlight ტექნოლოგიასთან სამუშაო გარემოს გამართვა და კომპანია Microsoft-ის მიერ შემოთავაზებული ინსტრუმენტების ინსტალაცია:

    ○ Silverlight Tools for Visual Studio – პაკეტით Visual Studio ხელსაწყოს გამდიდრება Silverlight პროექტების მართვისთვის საჭირო ფუნქციონალით [15];

    ○ Expression Blend 4: გრაფიკული ხელსაწყო XAML-ზე აგებული სამომხმარებლო ინტერფეისების ასაწყობად. Expression Blend-ი არ არის სავალდებულო ხელსაწყო, მაგრამ Visual Studio-სთან შედარებით უფრო მდიდარი სამომხმარებლო ინტერფეისის შემუშავებას უზრუნველყოფს [16];

    ○ Silverlight Toolkit: უფასო პროექტი, მოიცავს დამატებით Silverlight კონტროლებს [11].

- View და ViewModel დონეების ურთიერთკავშირის გამართვა. პროგრამის უკანა მხარეს მონაცემთა ცვლილების შემთხვევაში View ინტერფეისზე მონაცემების ავტომატური განახლება (ან პირიქით: აპლიკაციის წინა მხარეს მონაცემების ცვლილების შემთხვევაში - უკანა, პროგრამული ლოგიკის მხარეს შესაბამისი ცვლადების ავტომატური განახლება);

- ვებ-აპლიკაციაში სერვისების ასინქრონული გამოძახების მოდელის იმპლემენტირება და ხანგრძლივი გამოთვლითი პროცესების შემთხვევაში ბრაუზერის ბლოკირების თავიდან არიდება. მომხმარებლის ინფორმირება გამოთვლით სამუშაოების მსვლელობის შესახებ (ეკრანზე რაიმე

შეტყობინების ან ვიზუალიზაციის გამოტანა, მაგალითად, Busy Indicator კონტროლი);

- ბიზნეს წესების მართვა და ვალიდაციების მექანიზმების გამოყენება. ვალიდაციების იმპლემენტირება, როგორც კლიენტის, ასევე სერვერის მხარეს;

- უსაფრთხოების საკითხების გათვალისწინება, სახიფათო კოდებისა და ჰაკერული თავდასხმების არიდება, სენსიტიური ინფორმაციის დაცვა. პროგრამულ დანართში მომხმარებლის ქმედებების ლოგირების და აუდიტის ფუნქციონალის დაგეგმვა, ლოგირების ცხრილში შესატანი ინფორმაციის განსაზღვრა;

- მომხმარებლის აუთენტიფიკაციის გამართვა. მისი შესაძლო ვარიანტების გაანალიზება: მომხმარებლები ერთი და იმავე უფლებამოსილებით (Credentials) დაიშვებიან რამდენიმე პროგრამულ დანართში თუ არა, სერტიფიკატზე დაფუძნებული აუთენტიფიკაციის სისტემა გამოიყენება თუ არა და ა.შ.;

- პროგრამული აპლიკაციის განთავსება სერვერზე. ჩასატარებელი სამუშაოების დაგეგმვა, როდესაც ბრაუზერის Plug-In არ არის დაინსტალირებული. აპლიკაციის ხელახლა განთავსების შემთხვევის გათვალისწინება, მაშინ, როდესაც აპლიკაცია გაშვებულია კლიენტის კომპიუტერზე. აპლიკაციის ლოგიკურ მოდულებად დაყოფა და კომპონენტების ჩანაცვლება (მთელი დანართის შეცვლის გარეშე). ცალკეულ კომპონენტებზე ვერსიების განსაზღვრა (Versioning).

წიგნის *პირველი თავი* ეხება თანამედროვე ვებ-ტექნოლოგიების მიმოხილვას. გაანალიზებულია ASP.NET Web Forms, Silverlight, MVC ტექნოლოგიები, შედარებულია მათი ფუნქციონალური მახასიათებლები გამოვლენილია დადებითი და უარყოფითი მხარეები. განხილულია ASP.NET ვებ-ტექნოლოგიის განვითარების შკალა, მისი შემუშავება და .NET კლასების და ბიბლიოთეკების გამოყენება პროგრამული უზრუნველყოფის შექმნისას. განხილულია კომპანია Microsoft-ის შემდეგი თაობის პროდუქტი ASP.NET Web Forms, რომელმაც კიდევ უფრო განავრცო ვებ-აპლიკაციის მოვლენებით მართული მოდელი [17,18]. გამოვლენილია Web Forms ტექნოლოგიის დადებითი და უარყოფითი მხარეები. განხილულია ASP.NET MVC და Silverlight ტექნოლოგიების დანიშნულება, მათი საერთო მახასიათებ-ლები და განსხვავება. ილუსტრირებულია Silverlight ტექნოლოგიის საშუალებით მდიდარი და ინტერაქტიული გარემოს მიღწევა ვებ-გარემოში [12].

მეორე თავში წარმოდგენილია სერვის-ორიენტირე-ბული პროგრამული უზრუნველყოფის არქიტექტურა და მისი დაპროექტების მეთოდები. განხილულია პროგრამული დანართის ლოგიკურ დონეებზე გადანაწილებული არქიტექ-ტურა და ერთმანეთთან დაკავშირებული ფუნქციონალის ცალკეულ ლოგიკურ შრეებში (Layer) გაერთიანება. ილუსტრირებულია შრეების ერთი-მეორეზე ვერტიკალუ-რად განლაგება მათ შორის ინფორმაციის მიმოცვლა, კომუნიკაცია. დასაბუთებულია შრეების არქიტექტურის გამოყენების მიზეზი, რომელიც თავიდან გვარიდებს კონცე-ფციების მჭიდრო ურთიერთ-გადაჯაჭვულობას, უზრუნ-ველყოფს მოქნილობას და იოლ მართვას.

წარმოდგენილია Silverlight ტქექნოლოგიით დაწერილი RIA აპლიკაციის არქიტექტურა და კომპონენტების გადანაწილება ლოგიკურ შრეებში [13]. გამოვლენილია ძირითადი არქიტექტურული სტანდარტები, რომელთა გამოყენება მაღალ ხარისხობრივ მაჩვენებლებს იძლევა Silverlight აპლიკაციების შემუშავებისას. დემონსტრირებულია Command არქიტექტურის იმპლემენტაცია და Silverlight ვებ-აპლიკაციებში [19], სამომხმარებლო ინტერფეისის კონტროლი CanExecute მეთოდის საშუალებით: გამომძახებელი ობიექტის ჩართვა/ამორთვა.

ნაშრომში მოცემულია ასევე ანგარიშგებათა მოდულის ინტეგრაციის მეთოდები პროგრამულ უზრუნველყოფაში. მონაცემთა ანალიზისა და ანგარიშგებების გენერაციის მიზნით გამოყენებულია კომპანია Microsoft-ის ხელსაწყო SQL Server Reporting Services (SSRS).

გამოვლენილია Silverlight აპლიკაციების კიდევ ერთი უპირატესობა: სამომხმარებლო ინტერფეისის გამდიდრება ანიმაციებით და ტრანსფორმაციებით. პროგრამისტი აღარ საჭიროებს დამატებით სხვა ტექნოლოგიებთან მუშაობას (Adobe Flash). Expression Blend ხელსაწყოს გამოყენებით ნაჩვენებია რამდენიმე ანიმაციის შექმნის პროცესი [16]. ანიმაციის ყველა ტიპი Silverlight ტექნოლოგიაში მემკვიდრეობით მოდის Timeline კლასიდან, რომელიც განთავსებულია System.Windows.Media.Animation ბიბლიოთეკაში.

მესამე თავში განხილულია თეორიული საკითხების პრაქტიკული რეალიზაცია. მოცემულია ადამიანური რესურსების მართვის ელექტრონული სისტემა (eHRMS). დემონსტრირებულია საჯარო სამსახურის ბიუროს მიერ ჩატარებული კვლევის შედეგები, რაც თვალსაჩინოს ხდის HRMS სისტემის

საჭიროებასა და მოთხოვნადობას. გამოვლენილია სისტემის მიმართ წაყენებული ზოგადი მოთხოვნები.

HRMS პროგრამული უზრუნველყოფა შემუშავებულია .NET Framework 5.0 პროგრამულ გარემოში, C# ენაზე, SQL Server, WCF, Silverlight, SQL Server Reporting Services პროგ-რამული პაკეტების გამოყენებით. განხილულია სისტემის ლოგიკური კომპონენტები და თითოეული კომპონენტის მიერ შესასრულებელი ფუნქციონალი.

დასწრების კალენდრის მუშაობის მაგალითზე მოცე-მულია Silverlight გარემოში შემუშავებული სამომხმარებლო ინტერფეისის ეფექტურობა. დემონსტრირებულია ინფომა-ციის სხვადასხვა სახით ფორმატირება (რიცხვითი მნიშვნელობის ასახვა ფერში და პირიქით, ელემენტის გამოჩენა ან დამალვა და ა.შ.), Command არქიტექტურის იმპლემენტაცია და ვიზუალური ეფექტები.

მეოთხე თავში მოცემულია თანამედროვე არქიტექტუ-რული მიდგომები *მონოლითური და დისტრიბუციული* არქიტექტურის მქონე სისტემების ანალიზი [20,22]. დისტრიბუციული სისტემების არქიტექტურული მოდელი განხილულია „*მიკროსერვისული არქიტექტურის*“ მაგალი-თებზე [21]. წარმოდგენილია მათი სირთულეების და უპირატესობების მახასიათებლები, გამოკვეთილია გარკვეუ-ლი რეკომენდებული პრობლემების მინიმიზაციის მიზნით.

ნაშრომის მეხუთე თავში განხილულია მცირე და საშუალო ბიზნესის მართვის ციფრული პლატფორმის ბიზნეს პროცესები, ბიზნესის მოთხოვნების შესაბამისად [23,24]. წარმოდგენილია სისტემის დისტრიბუციული, Enterprise არქიტექტურა. განხილულია მისი შესაბამისი კომპონენ-ტების დიზაინის მაგალითები.

სისტემის სუფთა არქიტექტურის დისტრიბუციული გადაწყვეტა გვაძლევს საშუალებას, რომ სისტემა განვავითაროთ როგორც დამოუკიდებელი, მარტივად ინტეგრირებადი კომპონენტების ერთობლიობა და მივაღწიოთ მაღალ ხელმისაწვდომობას მომხმარებლისათვის კრიტიკული ფუნქციონალის ფარგლებში [25,26].

ანალიზის საფუძველზე, შემუშავებულია სისტემის უწყვეტი მიწოდების მოქნილი სტრატეგია - სისტემის მიწოდება სერვისის სახით.

მეექვსე თავში გადმოცემულია მიკროსერვისების ტექნიკური აღწერა და მათი შემადგენელი ძირითადი კომპონენტები. დეტალურად არის განხილული მცირე და საშუალო ბიზნესის მართვის ციფრული პლატფორმის არქიტექტურაში მონაწილე მიკროსერვისები და გაანალიზებულია მათი ფუნქციონალური ნაწილი.

ინფრასტრუქტურულად, სერვისები დანერგილია კონტეინერების სახით და გაზრდილი დატვირთულობის პირობებში, ავტომატურად ხდება სერვისების ჰორიზონტალური მასშტაბირება.

სისტემის უსაფრთხოების კუთხით, დეველოპმენტი ხორციელდება ,,OWASP Top Ten" (Open Worldwide Application Security Project – ღია მსოფლიო პროგრამის უსაფრთხოების პროექტი) მიდგომების სრული დაცვით და უსაფრთხო კოდის წერის სხვა გავრცელებული პრაქტიკების გათვალისწინებით [27].

მომხმარებლის პორტალისთვის შემუშავებულ იქნა პროდუქციის შესყიდვის პროცესში მონაწილე შემდეგი მიკროსერვისები: პროდუქტების, შეკვეთების, მომხმარებელთა მენეჯმენტის, გადახდების, პროდუქტის ადგილზე მიტანის სერვისის, შეტყობინებების, შეთავაზებების და

მარაგების მართვის. შესაძლებელია მათი სრულყოფისა და ახალი მიკროსერვისების ინტეგრაცია არსებულ სისტემაში.

მეშვიდე თავში წარმოდგენილია ჩვენი დაპროექტებული სისტემის ადმინისტრირების ფუნქციონალი და წარმოდგენილია შესაბამისი არქიტექტურული მოდელი. შემოთავაზებულია მონაცემთა საცავის აგების მეთოდოლოგია მიკროსერვისული არქიტექტურისათვის. განხილულია სისტემის დინამიკური, ჰორიზონტალური მასშტაბირების სტრატეგიები და ძირითადი მიდგომები, მაღალი დატვირთულობის პირობებში. ამ მიზნით გამოყენებული გვაქვს შეტყობინებათა პარალელური დამუშავების პრინციპი, რაც მნიშვნელოვნად ამცირებს სერვისების დატვირთულობას მოთხოვნების ასინქრონული დამუშავების ბაზაზე.

ამავე თავში გადმოცემულია კომპანიების ადმინისტრირების მოდულში რეპორტინგის ფუნქციონალის აქტიურად გამოიყენების საკითხი. იგი საშუალებას აძლევს კომპანიას ნებისმიერ დროს აწარმოოს საკუთარი ბიზნესის მონიტორინგი და სურვილისამებრ დააგენერიროს სხვადასხვა ტიპის/პერიოდის რეპორტი (ამონაწერი) სისტემიდან. ამასთანავე, სისტემა ავტომატურ რეჟიმში აგენერირებს რეპორტებს სხვადასხვა პერიოდის და ფუნქციონალის მიხედვით, ახორციელებს პროდუქციის რეალიზაციის და ბრუნვის სტატისტიკური მონაცემების ანალიზს და გენერირებული შედეგები სტატისტიკური დიაგრამების სახით ავტომატურ რეჟიმში ეგზავნება კომპანიის მენეჯმენტის რგოლის წარმომადგენლობას.

თავი 1
ბიზნესაპლიკაციები და Web-დაპროგრამების თანამედროვე ტექნოლოგიების კრიტიკული ანალიზი

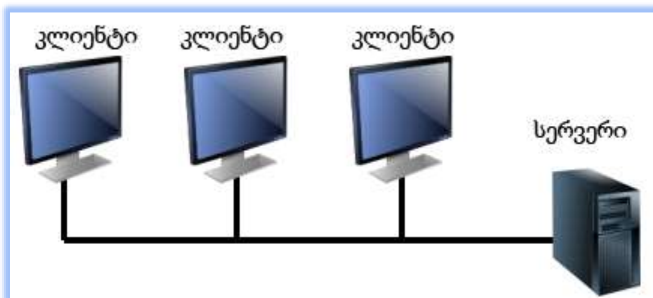
1.1. ბიზნეს აპლიკაციები, ადამიანური რესურსების მართვის სისტემა

ბიზნეს-აპლიკაცია ზოგადი ტერმინია და აღწერს ბიზნესის ან მწარმოებლის მიერ შემოთავაზებული პროდუქტების/სერვისების ერთობლიობას. ადამიანური რესურსების მართვის სისტემა წარმოადგენს პროგრამულ უზრუნველყოფას, რომელიც მართავს ადამიანურ კაპიტალთან დაკავშირებულ ბიზნეს-პროცესებს. HR-ის ძირითადი ნაწილი მოიცავს ორგანიზაციებში მომუშავე თანამშრომლების შესახებ საბაზისო ინფორმაციას. იგი აერთიანებს პერსონალურ მონაცემებს, როგორიცაა თანამშრომლის მისამართი, დაბადების თარიღი, ოჯახური მდგომარეობა, საკონტაქტო ინფორმაცია; სახელფასო უწყისის მონაცემები; დანამატები, პრემია, ჯილდო, წახალისება; სამუშაოს აღწერილობა; ორგანიზაციული სტრუქტურა; თანამშრომლის პორტალი და self-service მოდული და ა.შ. [4,5].

ადამიანური რესურსების მართვის პროგრამულ უზრუნველყოფის იმპლემენტირება გადაწყვეტილია ვებ-ტექნოლოგიების გამოყენებით. ეს განაპირობა მრავალი მომხმარებლის მიერ (როგორც დედაქალაქიდან, ასევე რეგიონებიდან) სისტემაში წვდომის აუცილებლობამ. მომხმარებელი პროგრამულ დანართთან მუშაობისთვის

საჭიროებს ვებ-ბრაუზერს და ინტერნეტ კავშირს. ეს მიდგომა უზრუნველყოფს ე.წ. „თხელი კლიენტების“ (კომპიუტერები, შეზღუდული აპარატურული შესაძლებლობებით) წვდომას ცენტრალიზირებული ინფრასტრუქტურის მქონე კომპლექ-სურ პროგრამულ უზრუნველყოფასთან. ამასთან ერთად, ვებ-ბრაუზერებისა და მათი მულტიმედიური შესაძლებლო-ბების გამოყენებამ პროგრამისტებს გაუადვილა მდიდარი, ინტერაქტიული სამომხმარებლო ინტერფეისის შექმნა.

ბიზნეს-ლოგიკის სერვერის მხარეს გატანა განაპირო-ბებს კლიენტის კომპიუტერზე შესასრულებელი სამუშაოების გამარტივებას და სიმსუბუქეს (ნახ.1). ამგვარი გადაწყვეტი-ლება მოსახერხებელია რეგიონებიდან მომხმარებლების მუშაობისთვის (რადგან ხშირად, რეგიონებში არსებული კომპიუტერული აპარატურა მოძველებული და სუსტია, გამოთვლითი პროცესები ნელა მიმდინარეობს) [6,57].



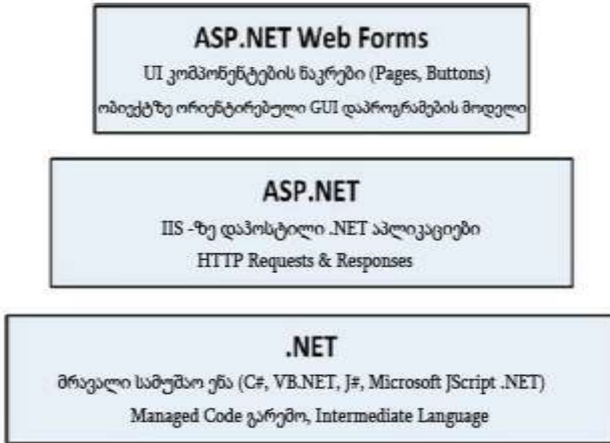
ნახ. 1.1. „თხელი კლიენტების" არქიტექტურა

1.2. ASP.NET ტექნოლოგიის განვითარების ისტორია

პროგრამული უზრუნველყოფის შექმნისას ერთ-ერთი მნიშვნელოვანი მომენტია დაპროგრამების ტექნოლოგიების სწორად შერჩევა. დაპროგრამების ტექნოლოგია გულისხმობს დაპროგრამების ხელსაწყოებისა და პროგრამული ენების კომბინაციას, რომელიც გამოიყენება პროგრამული უზრუნ-ველყოფის შესაქმენლად. პროგრამული დანართი ორი ძირითადი კომპონენტისგან შედგება: სერვერული ნაწილი და კლიენტის ნაწილი (Back End, Front End).

რომელი ტექნოლოგია ავირჩიოთ Web-ზე ორიენტირე-ბული პროგრამული დანართის შესაქმნელად? დღეისათვის ვებ-ტექნოლოგიების საკმაოდ დიდი არჩევანია. განვიხილოთ კომპანია Microsoft-ის თანამედროვე ვებ-ტექნოლოგიები და მათი გამოყენება ბიზნეს აპლიკაციების შემუშავებისას. გავაანალიზოთ ASP.NET Web Forms, Silverlight, MVC ტექნო-ლოგიები, მოვახდინოთ მათი მახასიათებლების შედარება, დადებითი და უარყოფითი მხარეების გამოვლენა (ნახ.2) [7].

საკითხის არსში უკეთ გასარკვევად გადავხედოთ web-ტექნოლოგიების განვითარების ისტორიას. 1960-იან წლებში ინტერნეტი ექსპერიმენტის დონეზე მუშავდებოდა და ხელ-მისაწვდომი იყო მომხმარებელთა შეზღუდული რაოდენო-ბისთვის (საგანმანათლებლო სისტემები, ინსტიტუტები, თავდაცვის სამსახურები). 1990 წელს *მოდემის* გამოგონებამ მკვეთრად დააჩქარა ინტერნეტზე მოთხოვნის ზრდა. 1993 წელს პირველი HTML ბრაუზერი შეიქმნა.



ნახ. 1.2. ASP.NET Web Forms ტექნოლოგიის განვითარების სკალა

ადრეული ვებ-საიტები ბროშურებს წააგავდა: შედგებოდა სტატიკური HTML გვერდებისგან. HTML 2 -ის შემუშავების შემდეგ გამოჩნდა HTML ფორმები, რომლებიც გარდა ფორმატირებული ტექსტის ასახვისა, კონტროლების (როგორიცაა ჩამოსაშლელი სია, ტექსტური ველი, ღილაკი) გამოყენების საშუალებასაც იძლეოდა.

ადრეულ ვებ-დეველოპმენტს ბევრი შეზღუდვა ჰქონდა: მაგალითად, მრავალი მომხმარებლის ერთდროულად შესვლისას შენელებული მუშაობა/წყვეტა და შედარებით მაღალი პროგრამული ლოგიკის გამოყენების (მომხმარებლის ავტორიზაცია, ინფორმაციის შენახვა, მონაცემთა ბაზიდან ჩანაწერების წამოღება/ასახვა) მწირი შესაძლებლობა.

ამ პრობლემების გადასაჭრელად კომპანია Microsoft-მა შეიმუშავა უფრო მაღალი დონის დეველოპმენტ პლატფორმები: ASP და ASP.NET. 2002 წელს კომპანია Microsoft-მა შეიმუშავა .NET Framework-ის 1.0 ვერსია, რომელსაც მოყვა Active Server Pages (ASP) ტექნოლოგიის შემუშავება.

ASP.NET აგებულია Common Language Runtime (CLR) გარემოზე, რაც საშუალებას აძლევს დეველოპერს ASP.NET კოდი შეიმუშაოს ნებისმიერ .NET ენაზე (C#, Visual Basic .Net).

ეს ტექნოლოგიები იძლეოდა დინამიკური, ინტერაქტიული, სერვერული მხარის ვებ-საიტების შემუშავების საშუალებას. ASP.Net მუშაობს HTTP პროტოკოლზე და მისი ბრძანებებით ახდენს ორმაგ კომუნიკაციას ბრაუზერსა და სერვერს შორის.

ASP.Net აპლიკაცია წარმოადგენს კომპილირებულ კოდს, რომელიც იყენებს .NET Framework-ში არსებულ კლასებს და იერარქიულ სტრუქტურებს [7].

ASP.NET Web Forms მოდელმა კიდევ უფრო განავრცო ვებ-აპლიკაციის ხდომილებებით მართული მოდელი - ბრაუზერი გადასცემს ვებ-ფორმის ინფორმაციას სერვერს და სერვერი საპასუხოდ აბრუნებს დამუშავებულ HTML გვერდს. Web Forms ტექნოლოგიით კომპანია Microsoft-ი შეეცადა დაემალა HTTP (თავისი დამახასიათებელი stateless თვისებით) და HTML (რომელიც იმ დროს დეველოპერთა უმრავლესობისთვის უცნობი იყო). ამის მისაღწევად შემუშავებულ იქნა სამომხმარებლო ინტერფეისის სპეციალური

კომპონენტები – სერვერული მხარის კონტროლები, რომლებიც:

- HTTP მოთხოვნებს შორის მდგომარეობის ამსახველ ინფორმაციას ინახავდა ViewState ობიექტში;
- რენდერდებოდა HTML ფორმატში და საჭიროების შემთხვევაში ავტომატურად აკავშირებდა კლიენტის მხარის მოვლენებს (მაგალითად, Button Click) შესაბამის სერვერულ პროგრამულ ლოგიკასთან (Event Handler).

საერთო ჯამში, შეიძლება ითქვას, რომ ASP.NET Web Forms ტექნოლოგია წარმოადგენს გიგანტურ აბსტრაქციის შრეს, რომელიც უზრუნველყოფს კლასიკურ, ხდომილებებით მართულ გრაფიკულ სამომხმარებლო ინტერფეისს (GUI) Web-ზე. ამ იდეოლოგიის ძირითადი დანიშნულება იყო ვებ-დაპროგრამების პროცესი დამგვანებოდა დესკტოპ-აპლიკაციების დაპროგრამების პროცესს. დეველოპერები აღარ საჭიროებდნენ ურთიერთ-დამოუკიდებელ HTTP Request-ებთან და Response-ბთან მუშაობას, მათ უკვე შეეძლოთ ეფიქრათ მდგომარეობის შენარჩუნების უნარიან სამომხმარებლო გარემოზე (Stateful UI). ამგვარად, Microsoft-მა მოახერხა დესკტოპ-აპლიკაციების დეველოპერების მთელი არმია გადმოეყვანა ვებ-აპლიკაციების სამყაროში.

1.3. რა ნაკლი აქვს ASP.NET Web Forms ტექნოლოგიას ?

ASP.NET Web Forms ტექნოლოგია, დადებით მხარეებთან ერთად გართულდა გარკვეული თვალსაზრისით შემდეგი მიზეზების გამო:

- ViewState ობიექტის წონა: მექანიზმი, რომელიც უზრუნველყოფდა მოთხოვნებს (Request) შორის მდგომარეობის შენარჩუნებას (View State) იწვევდა კლიენტსა და სერვერს შორის დიდი ზომის მონაცემთა ბლოკების ტრანსფერს (100-ობით კილობაიტს აღწევდა ყველაზე მოკრძალებულ ვებ-აპლიკაციაშიც კი) (ნახ.1.3).

ეს მონაცემთა ბლოკები წინ და უკან მოგზაურობდა ყოველი მოთხოვნის დროს, რაც იწვევდა დაყოვნებას (საპასუხო დროის მატების გამო) და სერვერის გამტარუნარიანობაზე მოთხოვნის ზრდას [8];

```
<body>
    <form method="post" action="Default.aspx" id="rsvpform">
<div class="aspNetHidden">
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="YWk++leAzekSRfrbNOvmVzeNbgdYN3fwgIxBsLyXRAEp0gX+mrmDP+/r+tuO5vHILH9y
</div>

<div class="aspNetHidden">

    <input type="hidden" name="__VIEWSTATEGENERATOR" id="__VIEWSTATEGENERATOR" value="CA0B0334" />
    <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION" value="hWzXo4wwVfU/3oHQTK+f...1QZ+HNlwphV+p2k/56IuZOpMTXEQ
7Rb25bnuyEd7KpEIT8GDY/rutHah6cnn0EAi6KIjo2C37KZHm0...fme/wa5CooM9T3iqg2nvt5l5K4ylo3c9ycDt" />
</div>
    <div>
        <h1>New Year's Eve at Jacqui's!</h1>
        <p>We're going to have an exciting party. And you're invited!</p>
    </div>
    <div id="validationSummary" data-valsummary="true" style="display:none;">
```

ნახ. 1.3. ViewState ობიექტი და ID ატრიბუტები

- გვერდის სასიცოცხლო ციკლი (Page Life Cycle): მექანიზმი, რომელიც აკავშირებს კლიენტის მხარის ხდომილებებსა (Event) და სერვერული მხარის ლოგიკას (Event Handler), წარმოადგენს გვერდის სასიცოცხლო ციკლის (Page Life Cycle) ნაწილს, რის გამოც მეტად რთული და დელიკატური საკითხი იყო ამ პროცესში გარკვევა და შეცდომების თავიდან არიდება. ხშირი იყო View State ობიექტის გამოისობით წარმოქმნილი შეცდომები, ზოგიერთი Event Handler კოდი დეველოპერისთვის გაურკვეველი მიზეზების გამო ვერ ეშვებოდა.
- მცდარი ილუზია განცალკევებული კონცეფციების შესახებ: ASP.NET Web-Forms ტექნოლოგიის code behind მოდელით აპლიკაციის კოდი HTML მარკირებიდან გატანილი იყო ცალკეულ code-behind კლასებში. ეს მეთოდი მოიწონეს, ვითომდა ლოგიკისა და პრეზენტაციის დონეების განცალკევების გამო, მაგრამ სინამდვილეში ეს რეალური არ იყო: დეველოპერი პირიქით, პრეზენტაციის შრეში წერდა აპლიკაციის ლოგიკის კოდსაც (მაგალითად code behind-ში შეიძლება შეგვხვედროდა სერვერული ხის იერარქიის სამართავად დაწერილი კოდიც და აპლიკაციის ლოგიკაც - მონაცემთა ბაზის ინფორმაციის დამუშავება). საბოლოო შედეგი ხშირად არასტაბილური და არასაიმედო იყო;
- HTML-ზე შეზღუდული კონტროლი: სერვერული კონტროლები რენდერდებოდნენ HTML-ის სახით. ASP.NET-ის ადრეულ ვერსიებში გამომავალი HTML მარკირება არ ემთხვეოდა ვებ-სტანდარტებს, ვერ ხერხდებოდა CSS

სტილების კარგი გამოყენება, სერვერული კონტროლები აგენერირებდნენ ID ატრიბუტის გაურკვეველ მნიშვნელო-ბებს, რომლებიც ართულებდა Javascript-ით მათზე წვდომას. მართალია, ეს პრობლემატური საკითხები თანამედროვე Web Forms ტექნოლოგიაში ბევრად გაუმჯობესებულია, მაგრამ, მიუხედავად ამისა, მაინც საკმაოდ რთულია მოსალოდნელი HTML-ის მიღება;

- ტესტირების ცუდი მხარდაჭერა: Web Forms ტექნოლოგიის შემუშავებისას მის არქიტექტორებს არ შეეძლოოთ მხედველობაში მიეღოთ ის ფაქტი, რომ სამომავლოდ ავტომატური ტესტირება პროგრამული უზრუნველყოფის შექმნის არსებით ნაწილი გახდებოდა. არაა გასაკვირი, რომ მჭიდროდ ურთიერთდაკავშირებული არქიტექტურა მოუხერხებელია Unit Testing-ისა და Integration Testing ტესტირების უზრუნველსაყოფად.

1.4. ASP.NET MVC ტექნოლოგია

Web Forms ტექნოლოგიის მიმართ აგორებული კრიტიკისა და სხვა არა-Microsoft ტექნოლოგიების (განსაკუთრებით - Ruby on Rails ტექნოლოგია, თავისი agile development მიდგომებით MVC არქიტექტურით, და HTTP პროტოკოლზე მუშაობის პრინციპით) მზარდი პოპულა-რობის საპასუხოდ, 2007 წლის ოქტომბერში Microsoft-მა განაცხადა ASP.NET პლატფორმაზე დაშენებული ახალი პროგრამული პლატფორმის გამოშვების შესახებ. ამ ახალ ტექნოლოგიაში კომბინირებულია Model-View-Controller (MVC) არქიტექტურის ეფექტიანობა, Agile Development

მიდგომის უახლესი იდეოლოგია და ASP.NET პლატფორმის საუკეთესო ნაწილები. იგი წარმოადგენს ტრადიციული ASP.NET Web Forms ტექნოლოგიის სრულყოფილ ალტერ-ნატივას, უპირატესია უმრავლეს შემთხვევაში, გარდა ტრივიალური ვებ საიტების პროექტირებისა [9].

1.4.1. MVC ტექნოლოგიის უპირატესობები

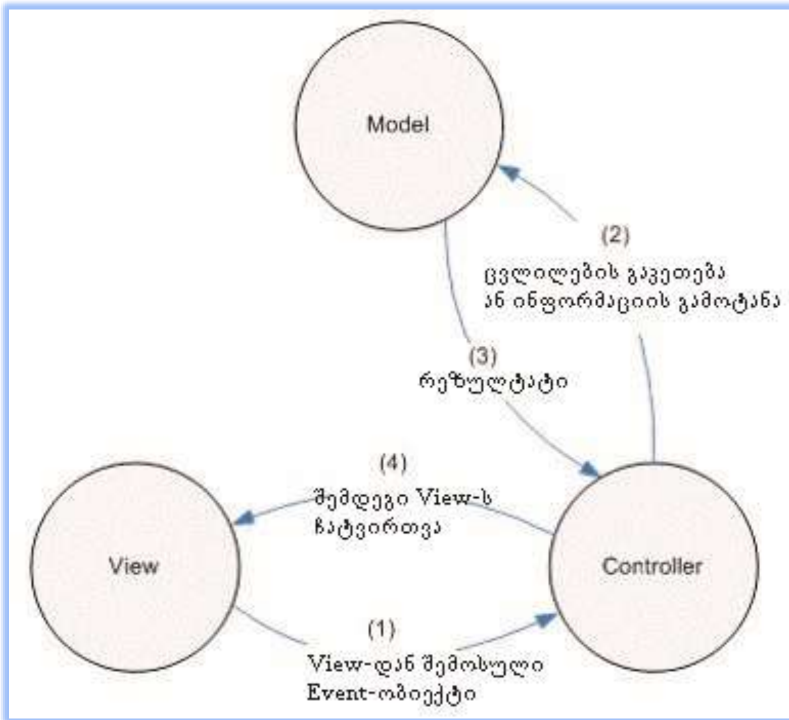
განვიხილოთ MVC ტექნოლოგიის გამოყენების უპირა-ტესობები, მისი ზოგადი მახასიათებლები და რეკომენდა-ციები.

- **MVC არქიტექტურა**

MVC არქიტექტურა ახალი არ არის - იგი სათავეს იღებს 1978 წლიდან (Smaltalk პროექტი, Xerox PARC კვლევითი ჯგუფის მიერ შემუშავებული) და დღეს დღეობით დიდი პოპულარობით სარგებლობს როგორც Web-აპლიკაციების ასაგებად გამოსაყენებელი არქიტექტურული მიდგომა. ამ ფართო აღიარების მიზეზებია:

- მომხმარებლის ურთიერთქმედება MVC არქიტექტუ-რით აგებულ აპლიკაციასთან ბუნებრივ ციკლს მიჰყვება: მომხმარებელი ასრულებს მოქმედებას, საპასუხოდ აპლიკა-ცია ცვლის Data Model ობიექტს და მომხმარებელს უბრუნებს შეცვლილ view ობიექტს, შემდეგ ეს ციკლი ისევ მეორდება. ეს მიდგომა კარგად არის მორგებული Web აპლიკაციების HTTP პროტოკოლზე HTTP Request და HTTP Response სერიებით მუშაობის პრინციპთან;
- ვებ-აპლიკაციები საჭიროებს რამდენიმე ტექნოლოგიის კომბინირებას (მონაცემთა ბაზა, HTML, მზა ბიბლიოთეკები

და სხვა), რომლებიც გადანაწილებულია ლოგიკურ შრეებში (Layer). ამ დაყოფის მიდგომა სრულ თანხმობაშია MVC არქიტექტურის კონცეფციასთან (ნახ. 1.4).



ნახ. 1.4. MVC არქიტექტურა

ASP.NET MVC Framework პროგრამულმა პლატფორმამ იმპლემენტაცია გაუკეთა MVC არქიტექტურას და ამით დიდი კონკურენცია გაუწია სხვა პროგრამულ პლატფორმებს, ხოლო MVC არქიტექტურა კი .NET სამყაროში წინა პლანზე წამოწია. სხვა, არა-Microsoft პლატფორმაზე მომუშავე დეველოპერების

მიერ წლობით დაგროვილი გამოცდილებისა და საუკეთესო პრაქტიკების გადაღებით გამდიდრებულმა ASP.NET MVC პლატფორმამ ზევრად წინ გაუსწრო კონკურენტებს [9,10].

- **განვრცობადობა (Extensibility)**

MVC პაკეტი აგებულია ურთიერთდამოუკიდებელი კომპონენტებისგან, რომლებიც აკმაყოფილებს .NET ინტერფეისს ან დაშენებულია აბსტრაქტულ მშობელ კლასზე (abstract base class). პროგრამისტს იოლად შეუძლია ჩაანაცვლოს არსებული კომპონენტები, (როგორიცაა მარშრუ-ტიზაციის სისტემა – routing system, view engine მექანიზმი, controller factory) საკუთარი იმპლემენტაციით.

MVC პროგრამული პლატფორმა დეველოპერს სთავა-ზობს თითოეული კომპონენტის გამოყენების 3 ვარიანტს:

- გამოიყენოს კომპონენტის default-იმპლემენტაცია (რაც აპლიკაციების უმრავლესობისთვის სავსებით საკმარისია);
- მემკვიდრეობის პრინციპით შექმნას ახალი კლასი (Subclass) და საკუთარი იმპლემენტაცია დააშენოს გაჩუმების პრინციპით არსებულ ობიექტზე;
- ახალი იმპლემენტაციით ჩაანაცვლოს კომპონენტი.

- **HTML -ზე და HTTP -ზე მაღალი ხარისხის კონტროლის მექანიზმები.**

ASP.NET MVC ტექნოლოგია სუფთა, სტანდარტებთან თავსებადი HTML მარკირების საშუალებას იძლევა. MVC ტექნოლოგიაში ჩაშენებული HTML helper მეთოდები იძლევა სტანდარტებთან თავსებად HTML კოდს.

მნიშვნელოვანი ცვლილება მოხდა ფილოსოფიურ მიდგომაში: Web Forms ტექნოლოგიის პრინციპებისგან გასხვავებით, აღარ არის მოწონებული კომპლექსური HTML-კონტროლების გამოყენებისა და დიდი HTML კოდის გენერაციის ტენდენცია. ამის ნაცვლად, MVC ტექნოლოგია მხარს უჭერს ელეგანტური, სადა მარკირების გამოყენებას CSS სტილებით. რა თქმა უნდა, თუ პროგრამისტს სურს მზა-კონტროლებისა და გაჯეტების გამოყენება სამომხმარებლო ინტერფეისის გასამდიდრებლად (კასკადური მენიუ, კალენ-დარის კონტროლი და სხვ.), ASP.NET MVC ტექნოლოგია თავისი ლოიალური მიდგომით მარკირების მიმართ, ადვილად იძლევა სხვადასხვა მზა ბიბლიოთეკების (jQuery UI, Bootstrap CSS library, Knokout) გამოყენების საშუალებას [7].

ASP.NET MVC ტექნოლოგიით გენერირებული გვერ-დები არ შეიცავს View State მდომარეობის ამსახველ მონაცემებს, ამგვარად მიღებული გვერდები უფრო მსუბუქია, ვიდრე ტიპური ASP.NET ვებ-ფორმები. მიუხედავად თანამედროვე სწრაფი ინტერნეტისა, ეს ეკონომია კვლავ დიდ უპირატესობას იძლევა და ამცირებს ვებ-აპლიკაციის გაშვების დროსა და საფასურს.

ASP.NET MVC მუშაობს HTTP პროტოკოლზე. პროგ-რამისტს სრული კონტროლი აქვს ვებ-ბრაუზერსა და სერვერს შორის მოთხოვნების მიმოცვლაზე.

AJAX ტექნოლოგია საკმაოდ იოლად არის შემუშა-ვებული და სამომხმარებლო ინტერფეისზე არ ასრულებს ავტომატურ postback მოქმედებებს [10].

- **ტესტირებადობა**

MVC არქიტექტურა მაღალი ხარისხის ტესტირების საშუალებას იძლევა. ეს შედეგი არქიტექტურული მიდგომის პრინციპებიდან გამომდინარეობს – ლოგიკური ნაწილები ერთმანეთისგან გამიჯნულია და სხვადასხვა შრეზეა გან-ლაგებული.

MVC პროგრამული პლატფორმის შექმნისას არქიტექ-ტორებმა MVC Framework-ის კომპონტენზე ორიენტირებული დიზაინის თითოეული ლოგიკური კომპონენტი საფუძვლი-ანად დააპროექტეს:

- Unit Testing და
- Mocking Tools

ინსტრუმენტებთან სრული შესაბამისობის მისაღწევად [7].

Visual Studio პროგრამაში Unit Test პროექტების შესაქ-მნელად დამატებულია ე.წ. *ოსტატები (Wizards)*, რაც შემდგომში ინტეგრირებადია სხვა ტესტირების ხელსაწ-ყოებთანაც, როგორიცაა NUnit, xUnit.

MVC-აპლიკაციაში იოლად რეალიზებადია ტესტები სხვადასხვა კომპონენტთან მიმართებაში (როგორიცაა, მაგა-ლითად, Controller, Action), სიმულაციები, სცენარები.

პროგრამისტს შეუძლია მომხმარებლის მოქმედების სიმულაციის გაკეთება ტესტირების სკრიპტებით.

მას არ ევალება იმის ცოდნა, თუ როგორია HTML ელემენტების და CSS კლასების სტრუქტურა, ან დაგე-ნერირებული ID მნიშვნელობები.

- **მძლავრი მარშრუტიზაციის სისტემა (Routing System)**

Web-ტექნოლოგიების განვითარებასთან ერთად, URL მისამართების სტრუქტურაც გაუმჯობესდა. ძველი URL მისამართი თუ ასე გამოიყურებოდა:

```
/App_v2/User/Page.aspx?action=show%20prop&prop_id=82742
```

ახალი URL მისამართი უფრო სუფთა და გასაგები ფორმატითაა წარმოდგენილი:

```
/to-rent/tbilisi/71-chavchavadze-street
```

URL მისამართის სტრუქტურაზე ყურადღების გამახვილებას შემდეგი საფუძვლიანი მიზეზები აქვს [10,11]:

- საძიებო მანქანები მნიშვნელობას ანიჭებს URL-ში ნაპოვნ საკვანძო სიტყვებს. მაგალითად ძიება კომბინაციით „rent in Tbilisi“ უფრო მეტი ალბათობით იპოვის სადა URL მისამართს;
- მომხმარებელი ვებ-ბრაუზერის URL სექციაში ხელით კრეფს მარტივ URL-მისამართს;
- მარტივ URL მისამართს მომხმარებელი ადვილად იმეორებს და იმახსოვრებს;
- არ ამჟღავნებს ტექნიკური იმპლემენტაციის დეტალებს (საქაღალდის ან ფაილის დასახელება, აპლიკაციის სტრუქტურა), ამიტომ ვებ-აპლიკაციის სტრუქტურის იმპლემენტაციის შეცვლა არ გამოიწვევს გარე ლინკების დარღვევას.

ადრეულ პროგრამულ პლატფორმებზე სადა URL მისამართების იმპლემენტაცია რთული გზებით მიიღწეოდა, მაგრამ ASP.NET MVC პლატფორმა იყენებს URL Routing

ფუნქციონალს და იძლევა ე.წ. „სუფთა“, აზრიან URL-გზავნილებს. ეს თვისება ამარტივებს თანამედროვე REST - სტილის URL-სქემების შემუშავებას.

➢ **ASP.NET პლატფორმის საუკეთესო პრაქტიკებზე აგებული**

კომპანია Microsoft-ის არსებული ASP.NET პლატფორმა ვებ-აპლიკაციების დეველოპმენტისთვის საჭირო მომწიფებულ და კარგად გამოცდილ კომპონენტთა სიმრავლეს იძლევა. ASP.NET MVC დაშენებულია .NET პლატფირმაზე - პროგრამული კოდის წერა ნებისმიერ .NET ენაზე არის შესაძლებელი, ხელმისაწვდომია .NET ბიბლიოთეკები და ASP.NET პლატფორმის მზა ფუნქციონალი, როგორიცაა: Authentication, Membership, Roles, Profiles, Internationalization. ეს ფუნქციონალი იგივე წარმატებით მუშაობს MVC-აპლიკაციებში, როგორითაც Web Forms აპლიკაციებში. MVC პროგრამული პლატფორმის საფუძველად აღებული ASP.NET პლატფორმა ხელსაწყოთა ფართო არჩევანს იძლევა MVC framework-ზე სამუშაოდ [7,9].

MVC პლატფორმამ მშობელი ASP.NET პლატფორმიდან მემკვიდრეობით მიიღო პროგრამირების ბოლო ტენდენციები და ინოვაციები, როგორიცაა: await keyword, lamda expressions, extension methods, anonymous types, dynamic types, LINQ queries.

➢ **ღია კოდი (Open Source)**

MVC პროგრამული პლატფორმა გახსნილია და ე.წ. „ღია კოდს“ წარმოადგენს - მისი პროგრამული კოდის გადმოწერა უფასოა და ჩამოტვირთულ პროექტში შესაძლებელია

საკუთარი ექსპერიმენტების ჩატარებაც კი. ეს განსაკუთრებით მოსახერხებელია საკუთარი რთული კომპონენტების შემუშავებისას ან როდესაც Debug-პროცესში შეცდომის კვალს მივყავართ სისტემურ კომპონენტთან და საჭიროა მის კოდში შესვლა. MVC კოდის გადმოწერა შესაძლებელია შემდეგი URL მისამართიდან: http://aspnetwebstack.codeplex.com

1.4.2. MVC ტექნოლოგიის შეზღუდვები

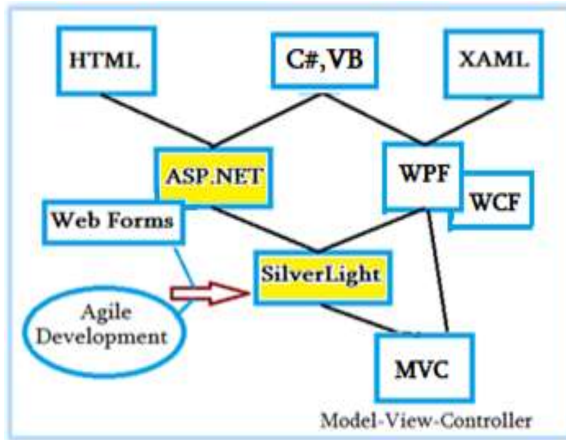
➢ ფუნქციონალის ურთიერთგამიჯვნის გამო აპლიკაციის ლოგიკა გადანაწილებულია სხვადასხვა ფაილში და არ იყრის თავს მხოლოდ ერთ რომელიმე ფიზიკურ ფაილში (როგორც ეს ხდება Web-Forms ტექნოლოგიის შემთხვევაში);

➢ არ არის სერვერული მხარის მდიდარი სამომხარებლო ინტერფეისის კონტროლები (Rich Server Side User Interface Controls). ამის გამო რთული ფუნქციონალის და ინტერაქტიულობის მისაღწევად HTML სტანდარტულ კონტროლებზე პროგრამისტს უწევს ლოგიკის წერა თითქმის თავიდან;

➢ სხვადასხვა კონტროლის მდგომარეობის შენახვა-შენარჩუნება Web-Forms პლიკაციებთან შედარებით რთული მისაღწევია, რადგან იქ სერვერული მხარის კონტროლების გამოყენება იძლეოდა ამის საშუალებას, - ინარჩუნებდა რა თავის მდგომარეობას postback-ებს შორის (ViewState ობიექტში).

1.5. ვებ-დაპროგრამების ინსტრუმენტი Silverlight

თანამედროვე ვებ-დაპროგრამების ტექნოლოგიებში დიდი პოპულარობა მოიპოვა Microsoft-ის *კროსპლატფორ-მულმა Silverlight ტექნოლოგიამ,* რომელიც უზრუნველყოფს მდიდარ სამომხმარებლო ინტერფეისს (Rich Interactive Applications – RIA) და ინტერაქტიული აპლიკაციების შემუშავების საშუალებას იძლევა (ნახ. 1.5-ა). განვიხილოთ Silverlight-აპლიკაციების შესაქმნელად საჭირო ინსტრუმენ-ტები და მათი გამოყენების უპირატესობები [9-11, 44,45].



ნახ. 1.5-ა. SilverLight-ის ადგილი

პროგრამულ პაკეტებში სამომხმარებლო ინტერფეისი განუწყვეტლივ ვითარდება და უმჯობესდება. ერთ-ერთი ყველაზე თვალსაჩინო მაგალითია Windows 7-ისა და მის წინამორბედ Windows-ის ვერსიებს შორის მკვეთრი სხვაობა, რაც გამოიხატება სამომხმარებლო ინტერფეისის მნიშვნელო-ვან გაუმჯობესებაში.

მაგალითად, ახალი *დავალებების პანელი* (TaskBar), სადაც ადრინდელი ტექსტური აღწერები დიდი Icon-ებით არის შეცვლილი. როდესაც მომხმარებელი კურსორს მიიტანს icon-ზე, შედეგად გამოჩნდება Window thumbnail (ნახ. 5-ბ).

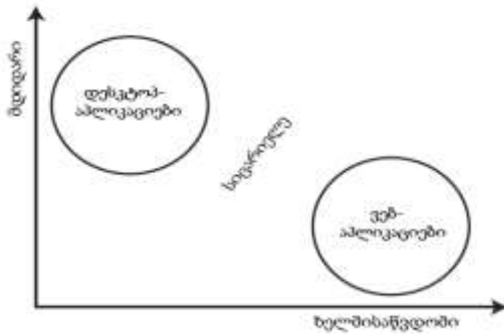
ნახ. 1.5-ბ. Windows-ის გაუმჯობესებული TaskBar

სამომხმარებლო ინტერფეისის სხვა გაუმჯობესების მაგალითია Aero Snap-თვისება Windows-ში. ეს თვისება მომხმარებელს საშუალებას აძლევს იოლად შეცვალოს ფანჯრის ზომები (ნახ. 1.6).

ნახ. 1.6. ფანჯრის ზომის შეცვლა (Aero Snap ფუნქცია)

თანამედროვე ბაზარზე გაიზარდა მოთხოვნილებები აპლიკაციების მიმართ - პროგრამული უზრუნველყოფა უნდა

აკმაყოფილებდეს არა მხოლოდ ფუნქციონალურ მოთხოვნილებებს, არამედ აგრეთვე უნდა ფლობდეს მდიდარ სამომხმარებლო ინტერფეისსაც. თუმცა მდიდარი სამომხმარებლო ინტერფეისი უშუალოდ პროგრამისტისთვის არ არის კრიტიკული მნიშვნელობის მახასიათებელი. ბოლო დროს აქტუალური ამოცანა გახდა ოქროს შუალედის პოვნა „მდიდარსა“ და „ხელმისაწვდომს“ შორის (ნახ. 1.7) [9].



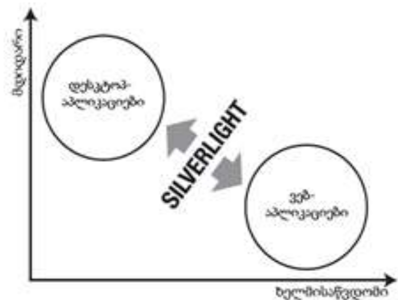
ნახ. 1.7. „მდიდარი" და „ხელმისაწვდომი" აპლიკაციების შედარება

თუ განვიხილავთ სტანდარტულ დესკტოპ-აპლიკაციებს – მათი ინსტალაცია ხდება ინდივიდუალური კლიენტის მანქანაზე. დესკტოპ-აპლიკაცია იძლევა მდიდარ და ინტერაქტიულ სამომხმარებლო ინტერფეისს, მაგრამ აპლიკაციის ფუნქციონალური მახასიათებლები დამოკიდებული ხდება კლინტის მანქანაზე (სადაც იგი არის დაინსტალებული). აქედან თვალსაჩინოდ ჩანს, რომ დაბალია აპლიკაციის ხელმისაწვდომობის მახასიათებელი – ცალკეულ

კომპიუტერზე უნდა ხდებოდეს ინსტალაცია და აპლიკაციის სამომავლო მხარდაჭერა. თითოეულ პლატფორმისათვის აპლიკაციას უნდა ჰქონდეს ამ პლატფორმაზე მორგებული პროგრამული კოდი [11].

ამის საპირისპიროა *ვებ-აპლიკაციები*, HTML-ზე ორიენტირებული პროგრამები, გათვლილი სხვადასხვა პლატფორმასა და ბრაუზერში სამუშაოდ. Microsoft პლატფორმაზე მომუშავე დეველოპერისთვის ეს ნიშნავს ASP.NET პროგრამული პაკეტის გამოყენებას და ვებ-სერვისების აგებას. აპლიკაციის ლოგიკის მხარის ძირითადი ნაწილი სრულდება სერვერზე, რაც აპლიკაციის სწრაფ მუშაობას უზრუნველყოფს. ამის საფასურია მწირი სამომხმარებლო გარემო – აპლიკაციას აქვს მაღალი ხარისხის ხელმისაწვდომობა, მაგრამ არც ისე „მდიდარი“ ინტერფეისი.

ტექნოლოგიების ამ ორ უკიდურესობას შორის *სიცარიელეა* წარმოქმნილი. ამ სიცარიელის ამოსავსებად პროგრამირების RIA მიდგომა იქნა შემუშა ვებული. მას აქვს ტრადიციული დესკტოპ-აპლიკაციების მსგავსი თვისებები და ფუნქციონალი. *RIA ტექნოლოგიის ერთ-ერთი სახეა Ms-Silverlight* (ნახ. 1.8).



ნახ. 1.8. RIA-ს ადგილი აპლიკაციებს შორის

1.6. RIA აპლიკაციებით ამოცანის გადაჭრა

RIA-ს კონცეპცია მანამდეც არსებობდა, მაგრამ ტერმინი „მდიდარი ინტერნეტ აპლიკაცია“ პირველად 2002 წელს იქნა გამოყენებული. დღეს დღეობით დასმული ამოცანის გადაჭრის ბევრნაირი მეთოდი არსებობს, რომლებიც აკამაყოფილებს RIA-ს განსაზღვრებას, მაგრამ ყველა მათგანს ახასიათებს საერთო თვისება: RIA ტექნოლოგია შეიცავს runtime პაკეტს, რომელიც ეშვება კლიენტის მანქანაზე და არქიტექტურულად ზის მომხმარებელსა და სერვერს შორის [9,10, 44,45].

ბოლო წლებში RIA-აპლიკაციებში ხშირად გამოიყენებოდა Flash ტექნოლოგია. Flash-ის შემოსვლამ ვებ-აპლიკაციების სამომხმარებლო ინტერფეისი გაამდიდრა, თუმცაღა საჭირო ხელსაწყების ნაკლებობის გამო Microsoft .NET დეველოპერებს რთულად უხდებოდათ Flash-ის ინტეგრაცია ვებ-აპლიკაციებში და ამ შემთხვევაშიც მხოლოდ მარტივი, არაფუნქციონალური დატვირთვის მატარებელი ეფექტების დასამატებლად თუ გამოიყენებოდა.

მნიშვნელოვანი სიახლე მოხდა როდესაც კომპანია Adobe-მ გამოუშვა Macromedia, შედეგად Flash-მა რეალიზაცია ჰპოვა Adobe-ს მიერ შემოთავაზებულ დეველოპმენტის ინსტრუმენტებში.

საპასუხოდ Microsoft-მა წარმოადგინა Silverlight ტექნოლოგია (ფორმალურად ცნობილი როგორც Windows Presentation Foundation Everywhere (WPF/E).

რას წარმოადგენს Silverlight ტექნოლოგია ? რა უპირატესობა აქვს მის გამოყენებას .NET-დეველოპერებისთვის ?

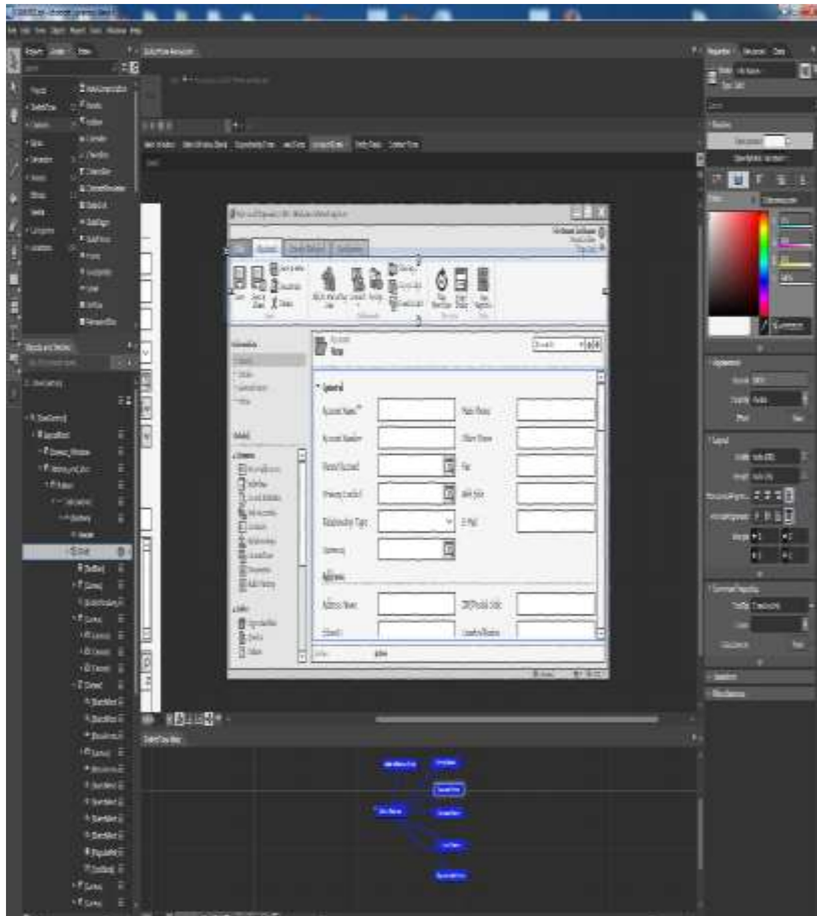
როგორც ზემოთ ითქვა, ყველა RIA-ტექნოლოგიას ახასიათებს საერთო თვისება: Client Runtime, რომელიც მომხმარებელსა და სერვერს შორის არის განთავსებული. Microsoft-ის RIA ტექანოლოგიაში Silverlight არის Client Runtimes.

Silverlight-ტექნოლოგია კროს-პლატფორმული, კროს-ბროუზერ დანართია (plug-in), რომელშიც სამომხმარებლო ინტერფეისი და გრაფიკული ელემენტები რენდერდება სპეციალურ არეზე/შრეზე (canvas), რაც შემდგომში ჩაშენებადია HTML-გვერდში. ენა, რომელიც გამოიყენება Silverlight-შრის (canvas) განსაზღვრის მიზნით, ცნობილია როგორც Extensible Application Markup Language (XAML, მიღებული XML-ის გაფართოებით) (ნახ. 1.9).

ამგვარად, XAML არის XML-ის ბაზაზე დაფუძნებული ენა, გარკვეულ ასპექტებში წააგავს HTML-ს. თუმცაღა XAML ენა გაცდა სამომხმარებლო ინტერფეისზე ელემენტების ურთიერთგანლაგების მარტივ აღწერას.

XAML-ის გამოყენებით შეგვიძლია განვსაზღვროთ დროის ინტერვალში ობიექტის ანიმაცია, ტრანსფორმაცია და სხვადასხვა მოვლენები (timelines, transformations, animations, events).

Silverlight-აპლიკაციების XAML-ის შესამუშავებლად მოსახერხებელია Microsoft-ის გრაფიკული ინსტრუმენტის – Expression Blend-ის გამოყენება [1,16].



ნახ. 1.8. XAML canvas (Microsoft Expression Blend)

1.7. Silverlight ტექნოლოგიის უპირატესობები

Silverlight ტექნოლოგია შეიცავს სხვა RIA ტექნოლოგიებისთვის დამახასიათებელ თვისებებს, მაგრამ აგრეთვე აქვს დამატებითი უნიკალური თვისებები, რითაც მნიშვნელოვანი უპირატესობა მოიპოვა ანალოგიურ ტექნოლოგიებთან მიმართებაში:

- თავსებადობა სხვადასხვა პლატფორმასა და ბრაუზერზე (cross-platform/cross-browser);
- შედგება .NET Framework-ის კროს-პლატფორმული ვერსიისგან;
- XAML-ი წარმოადგენს ტექსტზე დაფუძნებულ მარკირების ენას (text-based markup language);
- Silverlight-ს აქვს Out of Browser და Full Trust ოფციები;
- Silverlight runtime-ის იოლი ინსტალაცია კლიენტის მანქანაზე.

➢ კროს-პლატფორმულობა და ბრაუზერებთან თავსებადობა

როდესაც ASP.NET ტექნოლოგია გამოვიდა, მის ერთ-ერთ უპირატესობად ნათქვამი იყო მრავალ ბრაუზერთან თავსებადობა (cross-browser support). დეველოპერს პროგრამული უზრუნველყოფის კოდის მხოლოდ ერთი ვერსია უნდა შეემუშავებინა და ეს კოდი წარმატებით გაეშვებოდა ყველა თანამედროვე ვებ-ბრაუზერში. მაგრამ რეალურად, ASP.NET კონტროლების სრული ფუნქციონალი

მხოლოდ Internet Explorer-ის უახლეს ვერსიაში იყო ხელმისაწვდომი, ხოლო ნებისმიერ სხვა ვებ-ბრაუზერსა და წინა ვერსიებში ვებ საიტი აისახებოდა შედარებით დაბალი ხარისხით. ასე რომ, მიუხედავად ASP.NET-ის cross-browser თვისებისა, მომხმარებელი სხვადასხვა შედეგს იღებდა იმისდა მიხედვით - თუ რა ვებ-ბოუზერს იყენებდა.

Silverlight-ის გამოსვლამ ეს ხარვეზი გამოასწორა. იგი არა მხოლოდ cross-browser თვისების მატარებელია, არამედ აგრეთვე არის კროს-პლატფორმული (cross-platform) ტექნოლოგია. Silverlight აპლიკაცია თანაბარი წარმატებით მუშაობს სხვადასხვა ოპერაციულ სისტემასა (Windows, Mac OS, თუმცა არ არის გათვლილი Linux-ზე) და ვებ-ბრაუზერში (Internet Explorer, Firefox, Safari, Google Chrome).

- **ტექნოლოგიების მსგავსება**

Silverlight აპლიკაციების შემუშავებისას მირითადი როლი აკისრია Visual Studio ხელსაწყოს, რაც Silverlight ტექნოლოგიის კიდევ ერთი უპირატესობაა, რადგან პროგრამისტს არ უწევს ახალი IDE გარემოს შესწავლა (ნახ. 1.10).

გარდა Visual Studio ხელსაწყოსა, Microsoft-ი მომხმარებელს დამატებით სთავაზობს ხელსაწყოთა სხვა ნაკრებსაც, რომელიც Expression Studio სახელითაა ცნობილი. ერთ-ერთი ყველაზე ფართოდ აღიარებული ხელსაწყო ამ ნაკრებიდან არის Microsoft Expression Blend, რომელიც გამოიყენება Silverlight აპლიკაციების XAML კოდის შესამუშავებლად.

ნახ. 1.10. Silverlight runtime-ის გამაფრთხილებელი ლოგო

მიუხედავად იმისა, რომ Expression Blend დიზაინის მხრივ მკვეთრად განსხვავდება Visual Studio-სგან, მათ გააჩნია ბევრი საერთო ელემენტი. გარდა ამისა, Expression Blend-ით შეიძლება პროექტის გახსნა და მასზე მუშაობა Visual Studio-ს პარალელურად (რედაქტირებადი ფაილი მოითხოვს განახლებას, როდესაც პროგრამისტი გახსნის მას მეორე რედაქტორ ხელსაწყოში) [9].

- **მცირე ზომის Runtime პაკეტი და მარტივი ინსტალაცია**

Silverlight ტექნოლოგია საჭიროებს კლიენტის მანქანაზე Client Runtime პაკეტის ინსტალაციას, ამიტომ გადამწყვეტი მნიშვნელობა აქვს გადმოსაწერი საინსტალაციო ფაილის მინიმალურ ზომას. Silverlight 4-ის შემთხვევაში პაკეტის ზომა 6MB-ს არ აღემატება (შეიცავს .NET Framework-ის ქვესიმრა-ვლესა და ბიბლიოთეკებს). გარდა კომპაქტური ზომისა, Microsoft-მა კლიენტი უზრუნველყო მარტივი დიაგნოსტიკის მექანიზმით: თუ მანქანაზე არ არის დაინსტალებული

შესაბამისი Silverlight runtime პაკეტი, ეკრანზე გამოჩნდება ლოგო გამაფრთხლებელი შეტყობინებით (ნახ.1.10).

ამ შეტყობინებაზე დაკლიკვის შედეგად დაიწყება Silverlight runtime პაკეტის გადმოწერა. ინსტალაციის დასრულების შემდეგ Silverlight აპლიკაცია ხელმისაწვდომი ხდება მომხმარებლისთვის.

➢ Silverlight სამუშაო გარემო

Silverlight-ის გამოშვებამდე, Microsoft-ის პროგრამული პაკეტების უახლეს ვერსიებში სამუშაო გარემოს გამართვა საკმაოდ სწორხაზოვანი იყო, რაც გულისხმობდა Visual Studio-ს ბოლო ვერსიის დაყენებას და შესაბამისი software development kit პაკეტის ინსტალაციას.

Silverlight-ის შემთხვევაში Microsoft-ი ბევრ ახალ ხელსაწყოს სთავაზობს მომხმარებელს და სამუშაო გარემოს გამართვა გულისხმობს შემდეგი ინსტრუმენტების ინსტალაციას [9,44]:

1) Visual Studio-ს უახლესი ვერსია (რომლის ინსტალაცია ავტომატურად გულისხმობს .NET Framework-ის უახლესი ვერსიის ინსტალირებასაც);

2) Silverlight Tools for Visual Studio – პაკეტი Visual Studio-ს გაამდიდრებს Silverlight პროექტების მართვისთვის საჭირო ფუნქციონალით. იგი შედგება შემდეგი ხელსაწყოებისგან:

- Silverlight Runtime: სავალდებულოა ნებისმიერი კომპიუტერისთვის, რომელზეც უნდა გაეშვას Silverlight-აპლიკაცია;

- Silverlight Software Development Kit – ეს ე.წ. SDK წარმოადგენს Silverlight-კონტროლების, დოკუმენტაციის და სასწავლო მაგალითების კოლექციას;
- Silverlight Project Templates for Visual Studio – Silverlight აპლიკაციის შესაქმნელად წინასწარ გამზადებული შაბლონების ნაკრები.

3) Expression Blend 4: გრაფიკული ხელსაწყო XAML-ზე აგებული სამომხმარებლო ინტერფეისების ასაწყობად. Expression Blend-ი არ არის სავალდებულო ხელსაწყო, მაგრამ Visual Studio-სთან შედარებით უფრო მდიდარი სამომხმარებლო ინტერფეისის შემუშავებას უზრუნველყოფს;

4) Silverlight Toolkit: უფასო პროექტი, მოიცავს დამატებით Silverlight კონტროლებს.

1.8. MVVM არქიტექტურული სტანდარტი Silverlight აპლიკაციებში

Silverlight ტექნოლოგიის, როგორც პროგრამული უზრუნველყოფის შესამუშავებელი პლატფორმის პოპულარობის ზრდასთან ერთად წინა პლანზე წამოიწია საკითხი იმის შესახებ, თუ რომელი არქიტექტურული სტანდარტის (design pattern) გამოყენება იქნებოდა ყველაზე ხელსაყრელი Silverlight აპლიკაციებში. ამ ტექნოლოგიაზე მუშაობის მანძილზე დაგროვილმა პრაქტიკამ გამოავლინა, რომ კარგ შედეგს იძლევა MVVM (Model-View-ViewModel) არქიტექტურული სტანდარტის გამოყენება.

Silverlight-ის შემოსვლის ადრეულ ეტაპზე დეველოპერები პროგრამული ლოგიკის კოდს .XAML ფაილის back-end

მხარეს (code-behind ნაწილში) წერდნენ. თუმცა ეს მიდგომაც მუშაობს, მაგრამ MVVM არქიტექტურული სტანდარტით შემუშავებულ Silverlight აპლიკაცია გაცილებით უფრო სრულყოფილია და Silverlight ტექნოლოგიით შემოთავაზე-ბული შესაძლებლობების სრულად რეალიზაციის საშუალე-ბას იძლევა, როგორიცაა მაგალითად: ერთიდაიმავე კოდის სხვადასხვა ადგილას გამოყენება (code re-use), პროქეტის გამარტივებული მხარდაჭერა, მეტი მოდულარულობა პროგრამულ კოდში, ტესტირებასთან თავსებადობა [10,45].

MVVM მოდელი შედგება სამი ძირითადი ლოგიკური დონისგან (ნახ. 1.11):

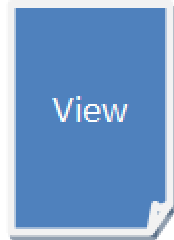
- Model: ბიზნესდომენი (ბიზნეს მოთხოვნები, მონაცემთა წვდომის ლოგიკა, model-კლასები);
- View: სამომხმარებლო ინტერფეისი (Silverlight screens);
- ViewModel: შუალედური დონე View-სა და Model-დონეებს შორის.

ნახ. 1.9. MVVM არქიტექტურული სტანდარტი

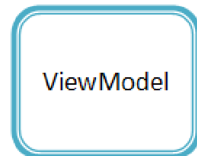
Model – ესაა ბიზნეს-დომეინი, რომელიც შეიცავს აპლიკაციაში გამოყენებულ მონაცემთა კლასებს (მაგალითად, Customer, Order და სხვ.), მონაცემებზე წვდომის კოდს (Data Access Code)

და ბიზნეს მოთხოვნებს (Business Rules). ზოგადად, მოდელი შეიძლება წარმოვიდგინოთ, როგორც ლოგიკური შრე, რომელიც ასახავს აპლიკაციაში არსებულ Entity-ერთეულებს და იმ ობიექტებს, რომლებიც ურთიერთქმედებს აპლიკაციის მონაცემთა საცავთან და ავსებენ entity-ს მონაცემებით. ზოგიერთი სპეციალისტის მოსაზრებით Model-შრე შედგება მხოლოდ აპლიკაციაში გამოყენებული model-კლასებისგან (როგორიცაა Customer, Order და სხვ.), თუმცა შეგვიძლია უფრო გავაფართოვოთ ეს ცნება და Model-შრის დონეზე გავაერთიანოთ მონაცემთა წვდომის კოდი (Data Access Code) და მოთხოვნების ბიზნეს წესები (Business Rules).

View - არის Silverlight აპლიკაციის ვიზუალიზაცია, რასაც ეკრანზე ხედავს საბოლოო მომხმარებელი. View ლოგიკურ დონეზე გაერთიანებულია XAML-ფაილები, მათთან ასოცირებული xaml.cs-ფაილები (code-behind). View დონე პასუხის-მგებელია მონაცემთა ვიზუალურად გამოტა-ნასა და მომხმარებლის მიერ პასუხად დაბრუნებული მონაცემების წამოღებაზე. View დონის ფუნქციებში არ შედის ინფორ-მაციის წამოღება მონაცემთა საცავიდან, ბიზნეს ლოგიკის წესების შესრულება და ვალიდაციების ჩატარება.

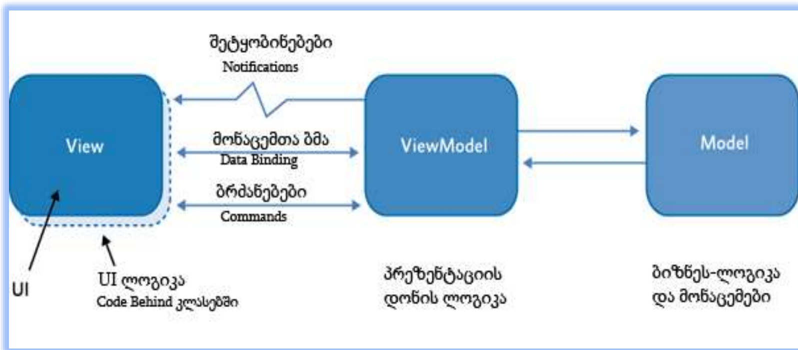


ViewModel - შრე ასრულებს შუამავლის როლს View და Model დონეებს შორის. ამ შრის ფუნქცია View-დონეზე გამოსატანი მონაცემების შენახვა და დამუშავებაა.



მაგალითად, ViewModel ობიექტი შეიძლება შეიცავდეს List<State> და List<Person> ტიპის ცვლადებს, რომლებიც View დონეზე გამოიყენება ComboBox-ის შესავსებად მონაცემებით.

1.12 ნახაზზე ნაჩვენებია MVVM არქიტექტურის დონეები.



ნახ. 1.10. MVVM არქიტექტურა

გარდა ამ 3 ძირითადი დონისა, კოდის ლოგიკური დონეების უკეთ გამიჯვნის მიზნით, Model-View-ViewModel კომბინაციას შეიძლება დაემატოს Service Agent კლასი, რომლის ფუნქციაში შედის Silverlight-იდან Remote Service მეთოდების გამოძახება და დაბრუნებული მონაცემების გადაწოდება ViewModel კლასისთვის (ნახ. 1.13).

ნახ. 1.11. MVVM არქიტექტურულ მოდელში ჩაშენებული Service Agent კლასი

შედეგად, ViewModel კლასს შეუძლია მონაცემთა წამოღების ფუნქცია გადაბაროს Service Agent კლასს. ერთი და იგივე Service Agent კლასი შეიძლება გამოყენებულ იქნას სხვადასხვა ViewModel კლასის მიერ (code re-use).

1.9. View და ViewModel დონეების ურთიერთკავშირი

Silverlight აპლიკაციების ერთ-ერთი მნიშვნელოვანი თვისებაა სამომხმარებლო ინტერფეისის ინტერაქიულობის მაღალი ხარისხი:

პროგრამის უკანა მხარეს მონაცემთა ცვლილების შემთხვევაში მოხდება View ინტერფეისზე მონაცემების ავტომატური განახლება (ან პირიქით: აპლიკაციის წინა მხარეს მონაცემების ცვლილების შემთხვევაში – უკანა, პროგრამული ლოგიკის მხარეს შესაბამისი ცვლადების ავტომატური განახლება).

ამ ფუნქციის მისაღწევად ViewModel კლასი იმპლე-მენტაციას უკეთებს Silverlight-ის INotifyPropertyChanged ინტერფეისს, რომელშიც აღწერილია Event ობიექტი, სახელწოდებით PropertyChanged.

მიღებულია, რომ INotifyPropertyChanged ინტერფეისის იმპლემენტაცია და OnNotifyPropertyChanged() მეთოდის აღწერა გაკეთდეს base-დონის ViewModel კლასში.

აპლიკაციის ყველა სხვა ViewModel კლასის აღწერაში კი მშობელ კლასად მითითებული იქნება ეს base კლასი. ამგვარი

მიდგომა გამორიცხავს ზედმეტი კოდის წერას. პროგრამული კოდი C# ენაზე ასე გამოიყურება [9,44]:

```
public class ViewModelBase : INotifyPropertyChanged
{
    protected void OnNotifyPropertyChanged(string p)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(p));
        }
    }

    public bool IsDesignTime
    {
        get
        {
            return (Application.Current == null)
                        || (Application.Current.GetType() ==
                typeof(Application));
        }
    }

    #region INotifyPropertyChanged Members

    public event PropertyChangedEventHandler PropertyChanged;

    #endregion
}
```

1.10. შედარებები პოპულარულ პლატფორმებს შორის (ASP.NET Web Forms, Silverlight, ASP.NET MVC)

1.10.1. Silverlight ტექნოლოიის უპირატესობები

ბიზნეს-აპლიკაციების შექმნა სხვადასხვა პლატფორმაზე შეიძლება, მაგრამ გარკვეული თვისებების და ფუნქციონალის გამო ბიზნეს აპლიკაციების დეველოპმენტისას უპირატესობა შეიძლება მიენიჭოს Silverlight ტექნოლოგიას [9,11,12]:

- დაპროგრამების მოდელი მარტივია – კლიენტს აპლიკაციაზე წვდომა აქვს ვებ-ბრაუზერის საშუალებით;
- არ არის აუცილებლობა .NET Framework-ის სრული პაკეტის დაყენებისა, მხოლოდ მცირე ზომის runtime-პაკეტის ინსტალაცია არის საჭირო;
- Silverlight-აპლიკაციის წერა შეიძლება ნებისმიერ .Net ენაზე (C#, Visual Basic და ა.შ.);
- Silverlight-აპლიკაცია თავსებადია სხვადასხვა პლატპორმასთან (Windows, Mac);
- HTML-აპლიკაციებთან შედარებით უფრო გაიოლებული დეველოპმენტის პროცესი.

1.10.2. როდის აჯობებს Silverlight ტექნოლოგიის გამოყენება

ტექნოლოგიის არჩევისას უნდა ვიხელმძღვანელოთ ობიექტურად და არა მხოლოდ იმ მოტივით, რომ ესა თუ ის ტექნოლოგია „მოდურია" და ბევრი პროგრამისტი იყენებს. უნდა გავითვალისწინოთ - არჩეული ტექნოლოგია რამდენად

სრულყოფილად დააკმაყოფილებს დასმულ ამოცანას, ბიზნეს-მოთხოვნებს. სხვა ტექნოლოგიების მსგავსად, Silverlight ტექნოლოგიასაც აქვს თავისი ძლიერი და სუსტი მხარეები, რომელიც დეველოპმენტის პროცესის დაწყებამდე წინასწარ უნდა იქნეს განხილულ-შეფასებული [9-12, 44,45].

1.10.3. ASP.NET-ტექნოლოგიასთან მიმართებაში შედარება

საერთო ფუნქციონალი:

ორივე ტექნოლოგია: ASP.NET და Silverlight ეშვება ვებ-ბრაუზერში;

- ASP.NET ტექნოლოგიის უპრატესობა Silverlight-თან მიმართებაში:
  - ASP.NET არ საჭიროებს კლიენტის მანქანაზე plugin-ის ინსტალაციას;
  - სრული .NET Framework პაკეტი (ოღონდ server-ის მხარეს მხოლოდ);
  - უფრო ხანგრძლივი გამოყენების პრაქტიკა (.NET Framework-ის შემოსვლიდან);
  - ეშვება ინტერნეტთან დაკავშირებულ თითქმის ყველა მოწყობილობაზე;
- ASP.NET ტექნოლოგიის უარყოფით მხარეები Silverlight-ტექნოლოგიასთან მიმართებაში:
  - HTML-ით შექმნილი აპლიკაციები სხვადასხვა ვებ-ბრაუზერში და ოს-ზე სხვადასხვაგვარად რენდერდება.

Silverlight-აპლიკაცია კი ყოველთვის ერთნაირად აისახება. ეს დიდ დროს ზოგავს, რაც გამოწვეულია cross-browser ტესტი-რების აუცილებლობით;

- რადგან Silverlight კლიენტის მხარეს სრულდება, მომხმარებელს არ უწევს წარამარა ლოდინი Server-ის postback-ებზე. ამის გამო Silverlight-აპლიკაცია უფრო ინტერაქტიულია;
- CLR და .NET Framework-ის ქვესიმრავლე გაშვებულია კლიენტის კომპიუტერზე, ამიტომ Silverlight-აპლიკაცია აღარ საჭიროებს დამატებით JavaScript კოდის წერას;
- სამომხმარებლო ინტერფეისის დეველოპმენტისთვის არსებობს საკმაოდ ეფექტური ხელსაწყოები (Visual Studio, Expression Blend – ერთერთი ყველაზე საუკეთესო და გავრცე-ლებული, თავსებადი ყველა პლატფორმასთან);
- აპლიკაციების გაშვება შესაძლებელია offline-რეჟიმში;

1.10.4. Windows Forms ტექნოლოგიასთან მიმართებაში შედარება

საერთო ფუნქციონალი:

- ორივე გადმოწერადია ვებ-ბრაუზერიდან (თუმცა Windows Forms-აპლიკაციები Silverlight-აპლიკაციებისგან განსხვავებით ვებ-ბრაუზერში ვერ გაეშვება) და თვით-განახლებადია ავტომატურ რეჟიმში, როგორც კი სერვერზე ახალი ვერსია გახდება ხელმისაწვდომი;

➢ Windows Forms ტექნოლოგიის უპრატესობანი Silverlight-ტექნოლოგიასთან მიმართებაში:

- მინიმალური შეზღუდვები (არ ხდება Sendboxing);
- სრული .NET Framework პაკეტი;
- სამომხმარებლო ინტერფეისის გამარტივებული დეველოპმენტი - არ საჭიროებს XAML-ის საფუძვლიან ცოდნას;
- უფრო ხანგრძლივი გამოყენების პრაქტიკა (მოყოლებული .NET Framework-ის შემოსვლიდან);

➢ Windows Forms ტექნოლოგიის უარყოფით მხარეები Silverlight-ტექნოლოგიასთან მიმართებაში:

- Windows Forms ტექნოლოგია საჭიროებს სრული .NET Framework პაკეტის ინსტალაციას;
- შეზღუდულია პლატფორმის მხრივ (Windows Forms-აპლიკაციები ეშვება მხოლოდ Windows ოპერაციულ სისტემაზე);
- Silverlight ტექნოლოგია უფრო მდიდარია data binding ფუნქციონალით, ვექტორული გრაფიკებით, ანიმაციებით;
- Silverlight ტექნოლოგია Windows Forms ტექნოლოგიასთან შედარებით უზრუნველყოფს უფრო მოქნილ სამომხმარებლო ინტერფეისს და styling-ფუნქციონალს;

1.10.5. WPF ტექნოლოგიასთან მიმართებაში შედარება

საერთო ფუნქციონალი:

- ორივე ტექნოლოგიაზე სამომხმარებლო ინტერფეისის შესაქმნელად გამოიყენება XAML-ი (თუმცა Silverlight-ზე მხოლოდ WPF-ის ქვესიმრავლე);
- ორივე ტექნოლოგიაზე დაწერილი აპლიკაციის deployment-ი შესაძლებელია ვებ-ბრაუზერის საშუალებით sandboxed application-ის სახით;

Windows Forms ტექნოლოგიის უპრატესობა Silverlight-თან მიმართებაში:

- მინიმალური შეზღუდვები (არ ხდება Sendboxing, გამონაკლისია მხოლოდ იმ შემთხვევაში, როდესაც ვებ-ბრაუზერიდან ხდება deployment-ის გაკეთება და "full trust" ოფცია არ არის ჩართული);
- WPF-ს აქვს სრული .NET Framework პაკეტი;

➢ WPF ტექნოლოგიის უარყოფით მხარეები Silverlight-ტექნოლოგიასთან მიმართებაში:

- WPF-ტექნოლოგია საჭიროებს სრული .NET Framework პაკეტის ინსტალაციას;
- შეზღუდულია პლატფორმის მხრივ (Windows Forms-აპლიკაციები ეშვება მხოლოდ Windows ოს-ში).

1.11. ამოცანის დასმა: HRMS with SOA

ჩვენი ნაშრომის ერთ-ერთი ძირითადი ამოცანაა Microsoft-კომპანიის თანამედროვე ტექნოლოგიების გამოყენებით, სერვის-ორიენტირებული არქიტექტურის საფუძველზე ავაგოთ პროგრამული უზრუნველყოფა, კერძოდ – *ადამიანური რესურსების მართვის სისტემა.*

ადამიანური რესურსების მართვის სისტემა (Human Resources Management System – HRMS) კომპიუტერული პროგრამაა და მოიცავს თანამშრომლებთან სამუშაო პროცესებს. იგი აღწერს HRM დისციპლინას, ორგანიზაციულ სტრუქტურას, თანამდებობების იერარქიულ ხეს და ძირითად ოპერაციებს ადამიანურ რესურსებზე. მთლიანობაში, HRMS სისტემის ზოგადი დანიშნულებაა სხვადასხვა პროგრამული უზრუნველყოფიდან თავი მოუყაროს ინფორმაციას და ერთიან, უნივერსალურ მონაცემთა საცავში გააერთიანოს. ფინანსური და ადამიანური რესურსების ერთიან მონაცემთა ბაზაში თავმოყრა განსაკუთრებით მნიშვნელოვანია ბიზნეს-პროცესების ანალიზის ჩასატარებლად, ორგანიზაციების შემდგომი განვითარებისა და სამუშაო პროცესების დასაგეგმად [4,5].

ქვემოთ მოყვანილ სექციებში წარმოდგენილია HRMS სისტემის მიმართ დასმული ამოცანები: ზოგადი მოთხოვნები, ფუნქციონალური მოთხოვნები, მომხმარებელთა ჯგუფების მართვა, სისტემის გრაფიკული ინტერფეისის მიმართ ზოგადი მოთხოვნები [13-15].

1.11.1. ზოგადი მოთხოვნები და კრიტერიუმები სისტემის მიმართ

- მოქნილი და დინამიური სისტემა – სისტემა ისე უნდა იყოს აგებული, რომ დანერგვის შემდეგ მასში მიმდინარე ცვლილების შეტანა არ იწვევდეს სისტემის ძირეულ გარდაქმნას.
- სამომავლოდ ახალი მოდულის ან ფუნქციონალობის დამატების შესაძლებლობა (Modular Design).
- სისტემის ვერსიების კონტროლი და განახლების ავტომატური ფუნქცია.
- სისტემას უნდა შეეძლოს ინფორმაციის გაცვლა სახელმწიფო ხაზინის საბუღალტრო სისტემასთან.
- ინფორმაციის უსაფრთხოების მაღალი ხარისხი.
- სისტემაში ნებისმიერი ქმედების დაფიქსირება (ლოგი-რება);

1.11.2. ფუნქციონალური მოთხოვნები

- კადრების ფურცლის, პერსონალის ბარათის, ორგანი-ზაციის სტრუქტურის, ასევე ერთჯერადი საშვების ბეჭდვის ფუნქცია;
- შაბლონების და საგამოცდო (შეფასების) კითხვარების გენერირების დინამიური ხელსაწყო; ასევე დაგენერირებული კითხვარების პასუხების კითხვის და შედეგების დათვლის შესაძლებლობა;
- ანგარიშების ამოღების მოქნილი სისტემა;
- შეტყობინებათა სისტემა – SMS-ით ან/და ელ. ფოსტით;
- დოკუმენტების საცავი (არქივი).

1.11.3. მომხმარებელთა და ჯგუფების მართვა

- სისტემას უნდა ჰქონდეს მომხმარებლის დამატების, ჯგუფებში გაერთიანებისა და ჯგუფების შექმნის საშუალება, ასევე მოდულებზე წვდომის და ქმედებათა განხორციელების უფლებების განაწილების ხელსაწყო (Permition Managment System) [20].
- სისტემაში უნდა არსებობდეს მინიმუმ სამი კატეგორიის მომხმარებელი:
  - *ადმინისტრატორი* (Super User - განუსაზღვრელი უფლების მქონე მომხმარებელი, რომელიც ანაწილებს სხვებზე უფლებებს, ქმნის ორგანიზაციის საბაზისო სტრუქტურის მახასიათებლებს და ა.შ.)
  - *მენეჯერი* (გადაწყვეტილებების მიმღები და ბიზნეს პროცესის აღმძრავი, ასევე ვიზირებისა და ბლოკირების უფლებით);
  - *ოპერატორი,* მონაცემების შემყვანი, ფორმების შემავსებელი;
  - ჩვეულებრივი *მომხმარებელი*, ანუ თანამშრომელი.
- მომხმარებელთა რეგისტრაცია, მათი ჯგუფებში გაწევრიანება არ უნდა გამორიცხავდეს ინდივიდუალურ გამონაკლისებს.
- ერთიანად უნდა განისაზღვროს უსაფრთხოების პოლიტიკა და მისი გათვალისწინებით მოხდეს სისტემის ადმინისტრირება.

1.11.4. სისტემის გრაფიკული ინტერფეისის (GUI) ზოგადი მოთხოვნები

სისტემაში რეგისტრირებულ ყველა მომხმარებელს უნდა ჰქონდეს ინდივიდუალური გარემო, სადაც თავისი კომპეტენციის და უფლებების ფარგლებში განახორციელებს მოქმედებებს (ე.წ. Self-service). რეგისტრაციის შემდეგ სამუშაო დაფაზე სასურველია დაფიქსირდეს ახალი დავალე-ბები, როგორც პირადი ასევე ჯგუფური. ასევე მიმდინარე და შესრულებული დავალებები [14, 15].

- სისტემის მომხმარებლისათვის უნდა შეიქმნას მაქსი-მალურად კომფორტული და მარტივი სამუშაო გარემო;
- აუცილებელი და პრიორიტეტული ენა უნდა იყოს ქართული.

1.12. პირველი თავის დასკვნა

– დაპროგრამების პლატფორმად გადაწყვეტილია Microsoft-ის კომპანიის .NET Framework 4.5 გამოყენება. კლიენტის მხარეს ინტერაქტიული და ეფექტური გარე-მო იმპლემენტირებულია Silverlight_5 ტექნოლოგიით. პროგრამული უზრუნველყოფის მონაცემთა სათავსოდ შემოთავაზებულია რელაციური MS SQL Server;

– MVVM არქიტექტურით შემუშავებული პროგრა-მული უზრუნველყოფა ლოგიკურად ყოფს კომპონენ-ტებს და ტესტირების პროცესს აიოლებს. სისტემა შედგება ურთიერთდამოუკიდებელი მოდულებისგან.

მათ შორის მჭიდრო ურთიერთკავშირი არ მიიღება, რაც ზრდის სისტემის გამოყენებადობას;

– პრეზენტაციის დონის RIA იმპლემენტაცია სამომხმარებლო გარემოს უზრუნველყოფს მდიდარი ანიმაციებითა და ინტერაქტიულობით, რაც Silverlight ტენოლოგიის უპირატესობაა Web Forms და MVC ტექნოლოგიებთან მიმართებაში.

თავი 2
პროგრამული აპლიკაციის არქიტექტურის დაპროექტების და აგების მეთოდოლოგია

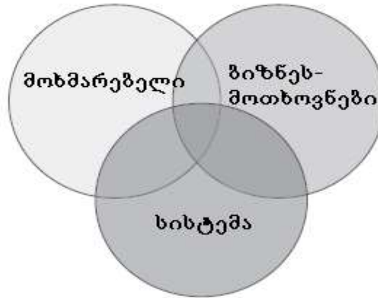
2.1. პროგრამული აპლიკაციის არქიტექტურა

პროგრამული აპლიკაციის (დანართის) არქიტექტურა სტრუქტურიზებული გადაწყვეტილების მიღების პროცესის მხარდამჭერი სისტემაა. იგი აკმაყოფილებს აპლიკაციის მიმართ წაყენებულ ტექნიკურ და ფუნქციონალურ მოთხოვ-ნილებებს. ამასთანავე, მიღწეულია მაღალი ხარისხობრივი მახასიათებლები, როგორიცაა უსაფრთხოება, მართვადობა და სწრაფქმედება. ყოველ მიღებულ გადაწყვეტილებას განაპირობებს მრავალი გარემო-ფაქტორი და თითოეულ მათგანს ზეგავლენა აქვს პროგრამული დანართის წარმა-ტებაზე [28].

ნებისმიერი სხვა კომპლექსური სტრუქტურის მსგავ-სად, პროგრამული აპლიკაცია უნდა აიგოს მყარ, საფუძვლიან საძირკველზე. თუ მხედველობიდან გამოგვრჩება ძირითადი სცენარები, არ გავითვალისწინებთ გავრცელებულ პრობლე-მებს – ამით პროგრამულ დანართს რისკის ქვეშ ვაყენებთ.

თანამედროვე ხელსაწყოები და პლატფორმები ამარტი-ვებს აპლიკაციის შემუავების პროცესს, მაგრამ ისინი ვერ გამორიცხავს დაგეგმვითი სამუშაოების ჩატარების აუცილებ-ლობას. ცუდი არქიტექტურით განპირობებული რისკ-ფაქტორები მოიცავს პროგრამული დანართის არასტაბი-ლურობას, რეალურ სამუშაო გარემოში (Production) გადატა-ნისა და მართვის სირთულეს [28-31].

პროგრამული აპლიკაცია (დანართი) უნდა დაიგეგმოს მომხმარებელის, სისტემის (IT ინფრასტრუქტურის) და ბიზნეს-მოთხოვნილებებთან თანხვედრაში (ნახ.2.1).



ნახ. 2.12. მომხმარებლის-, ბიზნესის- და სისტემის- მოთხოვნილებები

თითოეულ არეალში უნდა გამოვყოთ ძირითადი სცენარები, დავაიდენტიფიციროთ მნიშვნელოვანი ხარისხობრივი მახასიათებლები (მაგალითად, საიმედოობა, განვრცობადობა) და კმაყოფილებისა თუ უკმაყოფილების გამომწვევი ძირითადი ფაქტორები [43].

რეკომენდებულია შევიმუშაოთ საზომი ერთეულები და ყოველი მნიშვნელოვანი არეალისთვის გავზომოთ წარმატების კოეფიციენტი.

რეკომენდებულია ამ თანამაკვეთ არეალებს შორის ვიპოვოთ ოპტიმალური ბალანსი. მაგალითად, მომხმარებლის ინტერესების დაკმაყოფილება ეხება ბიზნეს- და IT-ინფრასტრუქტურას. ერთ-ერთ მათგანში გაკეთებული ცვლილება ავტომატურად აისახება მომხმარებელზე.

ანალოგიურად, მომხმარებლის მოთხოვნილებების ცვლილება ავტომატურად იქონიებს გავლენას ბიზნეს-პროცესებსა და IT ინფრასტრუქტურაზე. პროგრამული აპლიკაციის წარმადობა ძირითადი მიზანია მომხმარებლის-თვისაც და ბიზნესის კუთხითაც, მაგრამ ამ მომენტისთვის შესაძლოა სისტემურ ადმინისტრატორს არ შეეძლოს მიზნის 100%-იანი მაჩვენებლით უზრუნველყოფა არსებული კომპი-უტერული ტექნიკით. ამიტომ, ბალანსის პრინციპი გულისხმობს, რომ დროის ნებისმიერ მომენტში, მიზნის 80% იყოს მიღწეული.

არქიტექტურა ფოკუსირდება აპლიკაციაში ძირითადი ელემენტებისა და კომპონენტების გამოყენებაზე და მათ ურთიერთკავშირზე. პროგრამული დანართის დაგეგმვისას გათვალისწინებულ უნდა იქნას შემდეგი საკითხები:

- როგორ აპირებენ მომხმარებლები მოცემული პროგრამული დანართის გამოყენებას?
- როგორ უნდა დაინერგოს და იმართოს პროგრამული დანართი რეალურ სერვერზე (Producion)?
- რა არის აპლიკაციის ძირითადი ხარისხობრივი მოთხოვნები (უსაფრთხოება, სისწრაფე, ინტერნალიზაცია, პარალელიზმი);
- როგორ უნდა დაიგეგმოს აპლიკაცია, რომ მოქნილი იყოს და იოლად აიტანოს სამომავალო ცვლილებები;
- რა არქიტექტურული ტენდენციებია დღეს, რამაც შეიძლება ზეგავლენა იქონიოს პროგრამულ დანართზე მომავალში?

2.2. პროგრამული აპლიკაციის არქიტექტურის პრინციპები და დაგეგმვის პროცესი

➢ *არქიტექტურის ზოგადი პრინციპები*

არქიტექტურაზე ფიქრი გულისხმობს იმ ფაქტის გაცნობიერებას, რომ აპლიკაცია დროის მანძილზე უნდა განვითარდეს და არქიტექტურულმა დიზაინმა გარკვეულ ცვლილებები უნდა განიცადოს. წინასწარ ყველა მოთხოვნა არ გვეცოდინება, რომ თავიდანვე დავაპროექტოთ პროგრამული დანართი და ყველა საჭიროება გავითვალისწინოთ. ამიტომ, არქიტექტურა იმ პრინციპით უნდა შევიმუშაოთ, რომ ადაპტირებადი იყოს სამომავლო მოთხოვნილებებთანაც (რომლებიც ამ მომენტისთვის მკაფიოდ ჩამოყალიბებული არ არის). პროგრამული დანართის დაპროექტებისას უნდა გავითვალისწინოთ შემდეგი ძირითადი პრინციპები [28,29]:

– *ავაგოთ, რათა შეიცვალოს,* ნაცვლად იმისა, *რომ ავაგოთ, რათა გაძლოს.* ეს პრინციპი გულისხმობს, რომ აპლიკაცია უნდა იყოს მოქნილი და ცვლილებებისთვის მზად, რადგან სამომავლო მოდიფიკაციები გარდაუვალია;

– *მოდელირება, ანალიზი და რისკის შემცირება.* ინსტრუმენტების და მოდელირების სისტემების გამოყენება, როგორიცაა უნიფიცირებული მოდელირების ენა (UML); ვიზუალიზაციების შექმნა, რაც დაგვეხმარება მოთხოვნილებების გამოვლენაში, არქიტექტურული გადაწყვეტილებების მიღებასა და მათი ზეგავლენის გაანალიზებაში [32];

– *მოდელირებისა და ვიზუალიზაციების გამოყენებით დამკვეთთან კომუნიკაცია.* სისტემის არქიტექტურისთვის

კრიტიკულ მნიშვნელობას ატარებს დამკვეთთან ეფექტური კომუნიკაცია, მიღებული არქიტექტურული გადაწყვეტილე-ბათა და ცვლილებათა განხილვა, მოდელებისა და ვიზუალიზაციების ჩვენება;

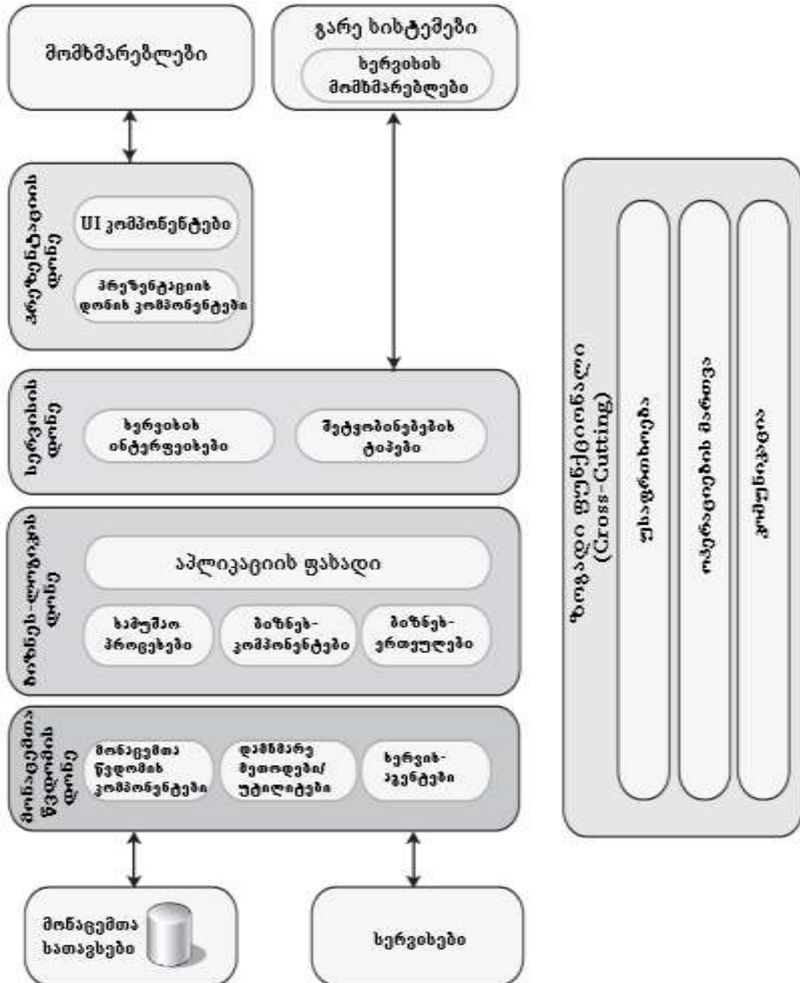
– *ძირითადი გადაწყვეტილებების იდენტიფიცირება.* წინასწარ დადგინდეს მნიშვნელოვანი არეალები, სადაც დიდია შეცდომის დაშვების ალბათობა. თავიდანვე ჩაიდოს დროითი ინვესტიცია ამ კრიტიკული გადაწყვეტილებების მისაღებად, რათა შემუშავებული არქიტექტურული დიზაინი უფრო მოქნილი და ცვლილებების მიმართ მედეგი იყოს.

➢ *არქიტექტურის ძირითადი პრინციპები*

პროგრამული დანართის არქიტექტურას ხშირად განმარტავენ, როგორც სისტემის ორგანიზების პროცესს, სადაც სისტემა სპეციფიკური ფუნქციონალის შემცველი კომპონენტების კოლექციაა. სხვაგვარად რომ ვთქვათ, არქიტექტურა ფოკუსირდება კომპონენტების ორგანიზებაზე, რათა მათ იმპლემენტაცია გაუკეთონ სპეციფიკურ ლოგიკურ ფუნქციონალს [28-31].

ფუნქციონალის ორგანიზებას უწოდებენ კომპონენ-ტების დაჯგუფებას ინტერესის არეალში. 2.2 ნახაზზე ნაჩვე-ნებია აპლიკაციის ზოგადი არქიტექტურა, სადაც კომპონენ-ტები გადანაწილებულია სხვადასხვა ლოგიკურ არეალში. გარდა კომპონენტების დაჯგუფებისა, გამოყოფილია „ზოგა-დი ფუნქციონალის არეალი“, რომელიც ფოკუსირდება კომპონენტების ურთიერთკავშირზე და ერთობლივ მუშაო-ბაზე. იმისათვის, რომ მივაღწიოთ მინიმალურ დანახარჯს,

მაღალი ხარისხის გამოყენებადობასა და ფუნქციონალის განვრცობადობას, უნდა გავითვალისწინოთ ლოგიკური დონეების დიზაინის ძირითადი პრინციპები [27,28].



ნახ. 2.2. პროგრამული დანართის ზოგადი არქიტექტურა

– *კონცეფციების ურთიერთგამიჯვნა.* პროგრამულ დანართში ლოგიკა გადავანაწილოთ ცალკეულ ლოგიკურ დონეებზე, ისე, რომ თანაკვეთის წერტილების მინიმალური რაოდენობა მივიღოთ. ამრიგად, გვექნება ლოგიკური კომპონენტების მინიმალური გადაჯაჭვულობა და დაწყვილებულობის დაბალი კოეფიციენტი;

– *ერთ დავალებაზე პასუხისმგებლობის პრინციპი.* პროგრამული დანართის თითოეული კომპონენტი (მოდული) პასუხისმგებელია მხოლოდ 1-კონკრეტულ ფუნქციონალზე;

– *მინიმალური ცოდნის პრინციპი* (აგრეთვე ცნობილია დემეტრეს კანონის სახელწოდებით). კომპონენტმა ან ობიექტმა არ უნდა იცოდეს სხვა კომპონენტების და ობიექტების შიგა სამუშაოების შესახებ;

– *გამეორების გამორიცხვა* (Don’t Repeat Yourself - DRY). აპლიკაციის დიზაინის თვალსაზრისით, ყოველი ფუნქციონალი იმპლემენტირებული უნდა იყოს მხოლოდ ერთ კონკრეტულ ადგილზე და სხვა რომელიმე კომპონენტში მისი დუბლირება არ უნდა ხდებოდეს.

– *არქიტექტურის წინასწარი დაგეგმარების მინიმიზაცია.* დავგეგმოთ მხოლოდ ის ფუნქციონალი, რაც ამ მომენტისთვის არის საჭირო. რისკების მაღალი ალბათობის შემთხვევაში, შესაძლოა დაგვჭირდეს წინასწარი დაგეგმარება და ტესტირება, მაგრამ სხვა შემთხვევებში, განსაკუთრებით კი Agile Development მიდგომით სარგებლობისას, რეკომენდებულია მხოლოდ აუცილებელი სამუშაოების დაგეგმვა და არა-მკაფიო წინასწარი სამუშაოების დაგეგმვის მინიმიზაცია.

➢ *პროგრამული აპლიკაციის ლოგიკურ დონეებზე გადანაწილება*

აქ იგულისხმება ერთმანეთთან დაკავშირებული ფუნქციონალის ცალკეულ ლოგიკურ შრეებში (Layer) გაერთიანება.

შრეები ერთი-მეორეზე ვერტიკალურად არის დალაგებული. თითოეული შრის ფუნქციონალი ატარებს რაიმე ერთ კონკრეტულ დანიშნულებას და შრეებს შორის მიმდინარეობს *ინფორმაციის მიმოცვლა და კომუნიკაცია.* პროგრამული აპლიკაციის გადანაწილება ლოგიკურ შრეებში თავიდან გვარიდებს კონცეფციების მჭიდრო ურთიერთგადაჯაჭვულობას, უზრუნველყოფს მოქნილობას და მარტივად მართვას [30].

ლოგიკური შრეების არქიტექტურა მუშაობის პრინციპით წააგავს ამობრუნებულ პირამიდას: ყოველი შრე უკავშირდება და იყენებს მის უშუალოდ ერთი დონით ქვემოთ მდგომი შრის ფუნქციონალს. შრეების უფრო თავისუფალ არქიტექტურაში დაშვებულია ნებისმიერ ქვედა დონის შრესთან ურთიერთქმედება.

აპლიკაციის ლოგიკური შრეები შეიძლება იმყოფებოდეს ერთსადაიმავე ფიზიკურ კომპიუტერზე (საერთო დონე, Tier) ან გადანაწილებული იყოს რამდენიმე კომპიუტერზე (N-Tier). N-Tier განაწილების შემთხვევაში შრეებს შორის კომუნიკაცია ხორციელდება მკაფიოდ განსაზღვრული ინტერფეისებით. მაგალითად, ტიპური ვებ-აპლიკაციის არქიტექტურული დიზაინი შედგება: პრეზენტაციის (Presentation Layer),

ბიზნეს-ლოგიკის (Business Layer) და მონაცემთა სათავსოსთან წვდომის (Data Layer) დონეებისგან.

პროგრამული აპლიკაციის ლოგიკურ დონეებზე გადანაწილების არქიტექტურული სტილი გულისხმობს შემდეგი ზოგადი პრინციპების დაცვას:

- *აბსტრაქცია (Abstraction).* შრეების არქიტექტურა აბსტრაქციას უკეთებს მთელ სისტემას და წარმოაჩენს ერთ მთლიანობად, ამავდროულად, საკმაო დეტალიზაციას იძლევა და თვალსაჩინოს ხდის თითოეული შრის როლსა და დანიშნულებას, ასევე, შრეებს შორის ურთიერთ-კავშირს;
- *ინკაფსულაცია (Encapsulation).* არქიტექტურის დიზაინის პროცესში არანაირი ვარაუდი არ კეთდება მონაცემთა ტიპების, მეთოდებისა და property-წევრების შესახებ - ეს შიდა ინფორმაცია ლოგიკური შრის (Layer) საზღვრებს გარეთ არ გადის;
- *მკაფიოდ ჩამოყალიბებული ფუნქციონალი.* ყოველი შრის ფუნქციონალი ერთიმეორისგან ურთიერთგამიჯნულია. ზედა დონის შრე (მაგალითად, პრეზენტაციის დონე) ქვედა დონის შრეებს (მაგალითად, ბიზნეს-ლოგიკის ან მონაცემებთან წვდომის დონე) უგზავნის ბრძანებებს და რეაგირებს ამ შრეებში წარმოქმნილ მოვლენებზე (Events). ეს კი შრეებს შორის მონაცემების ზედა და ქვედა მიმართულებით მოძრაობის საშუალებას იძლევა;
- *ერთიანი, მჭიდრო ლოგიკა შრის შიგნით (High Cohesion).* ყოველი შრისთვის განსაზღვრულია მკაფიოდ ჩამოყალიბებული ფუნქციონალი, რაც გარანტიას იძლევა,

რომ თითოეული შრე შედგება მხოლოდ მისთვის განსაზღვრული დავალებების იმპლემენტირებისთვის საჭირო ფუნქციონალისგან;

- *ხელმეორედ გამოყენება (Reusability).* ქვედა დონის შრეები ზედა დონის შრეებისგან დამოუკიდებლია, რაც მათი სხვა სცენარებში გამოყენების პოტენციალს იძლევა;
- *სუსტი კავშირი (Loose Coupling).* შრეებს შორის კომუნიკაცია ეფუძნება აბსტრაქციას და მოვლენებს, რაც თავიდან გვარიდებს შრეებს შორის მჭიდრო კავშირის წარმოქმნას.

ლოგიკური შრეების არქიტექტურის ნიმუშებია სტანდარტული ბიზნეს აპლიკაციები (Line-Of-Business - LOB), საბუღალტრო პროგრამები, ვებ-საიტები, ოგანიზაციების საწარმოო ვებ-აპლიკაციები და ა.შ.

ფუნქციონალის ურთიერთ გამიჯვნა პარალელური მუშაობის საშუალებას იძლევა: სამომხმარებლო გრაფიკულ ინტერფეისს ქმნიან ვებ-დიზაინერები და ამავდროულად, პროგრამისტები წერენ პროგრამულ ლოგიკას.

გარდა ამისა, იოლია ცალკეული კომპონენტების ტესტირება და სამომავლო მხარდაჭერა [31].

2.3. 3- და N- დონიანი არქიტექტურული სტილები

3-დონიანი და N-დონიანი (3-Tier, N-Tier) არქიტექ-ტურული სტილები გულისხმობს პროგრამული აპლიკაციის ფუნქციონალის სეგმენტებად განაწილებას და შრეების არქიტექტურული სტილის (Layered Style) ანალოგიურია. განსხვავება მდგომარეობს იმ ფაქტში, რომ თითოეული Tier-სეგმენტი (იარუსი) ფიზიკურად მოთავსებულია განცალკევე-ბულ კომპიუტერზე. ეს მიდგომა განვითარდა *კომპონენტზე-ორიენტირებული არქიტექტურიდან* და საკომუნიკაციოდ იყენებს პლატფორმისთვის დამახასიათებელ მეთოდებს (ნაც-ვლად შეტყობინებაზე დაფუძნებული მიდგომისა) [33-35].

N-დონიან არქიტექტურას ახასიათებს აპლიკაციის ფუნქციონალური დეკომპოზიცია, სერვისზე ორიენტირებუ-ლი კომპონენტები და მათი განაწილებული განთავსება ფიზიკურ სერვერებზე, რაც უზრუნველყოფს მაღალი ხარის-ხის მართვადობას, ხელმისაწვდომობას და მაშტაბურობას.

ყოველი დონე (Tier) დანარჩენი დონეებისგან სრულიად დამოუკიდებელია, გარდა მხოლოდ მის უშუალოდ ერთი საფეხურით ზემოთ და ქვემოთ მდგომი დონეებისა.

N პოზიციაზე მყოფ დონეს ევალება იცოდეს, როგორ მოემსახუროს N+1 პოზიციაზე მყოფი დონიდან მოსულ მოთხოვნას, როგორ გადაუგზავნოს მოთხოვნა N-1 პოზიცი-აზე მყოფ დონეს (თუკი ასეთი დონე არსებობს) და როგორ დაამუშაოს მოთხოვნიდან დაბრუნებული შედეგები.

წარმადობის ამაღლების მიზნით დონეებს შორის კომუნიკაცია ასინქრონულად ხორციელდება.

N-დონიანი არქიტექტურა მინიმუმ 3 ლოგიკური ნაწილის არსებობას გულისხმობს, რომელთაგან თითოეული ნაწილი განთავსებულია განცალკევებულ ფიზიკურ სერვერ-ზე. ყოველი ნაწილი პასუხისმგებელია სპეციფიკურ ფუნქცი-ონალზე. შრეების არქიტექტურის (Layered Design) გამო-ყენებისას, შრე განთავსდება Tier-დონეზე, თუკი ამ შრის მიერ აღწერილ ფუნქციონალზე დამოკიდებულია ერთზე მეტი სერვისი ან აპლიკაცია [28].

N-დონიანი/3-დონიანი არქიტექტურული სტილის მაგალითია ფინანსური ვებ-აპლიკაცია, სადაც უსაფრთხოება გადამწყვეტ როლს თამაშობს. ბიზნეს-ლოგიკის შრე უნდა განთავსდეს Firewall დამცავი მექანიზმის უკან. ეს კი თავის მხრივ, გვაიძულებს პრეზენტაციის ლოგიკის შრე ცალკე დონეზე განვათავსოთ პერიმეტრალურ ქსელში. კიდევ ერთი მაგალითია ტიპური RIA აპლიკაცია (Rich Client Application), სადაც პრეზენტაციის დონე განთავსებულია კლიენტის მანქანაზე, ხოლო ბიზნეს-ლოგიკისა და მონაცემებთან წვდომის დონე განთავსებულია ერთ ან რამდენიმე სერვერზე.

N- და 3-დონიანი არქიტექტურული სტილის ძირითა-დი უპირატესობებია:

– *ცვლილებათ/განახლებათა შესაძლებლობა* (Maintainabi-lity). რადგან თითოეული დონე სხვა დონეებისგან დამოუკიდებელია, განახლებები და ცვლილებები შეიძლება გაკეთდეს ისე, რომ მთლიან აპლიკაციას არ შევეხოთ;

– *მასშტაბურობა* (Scalability). რადგან აპლიკაცია განაწილებულია ლოგიკურ დონეებზე, მისი განვრცობა საკმაოდ სწორხაზოვანია;

– *მოქნილობა* (Flexibility). რადგან ყოველი ლოგიკური დონე დამოუკიდებლად იმართება, აპლიკაციის მოქნილობის მაჩვენებელი საკმაოდ მაღალია;

– *ხელმისაწვდომობა* (Availability). აპლიკაცია იყენებს მოდულარულ არქიტექტურას, ეს კი სისტემას ანიჭებს კომპონენტების იოლად განვრცობის/გაფართოების შესაძლებლობას და ზრდის ხელმისაწვდომობას.

რეკომენდებულია ვიფიქროთ 3- და N-დონიანი არქიტექტურის გამოყენებაზე, თუკი გვაქვს შემდეგი სიტუაციები:

- ერთ რომელიმე დონეზე გამოთვლითი სამუშაოები სრულდება ნელა, რაც დროით დაყოვნებას იწვევს სხვა დონეებზეც;
- უსაფრთხოების მოთხოვნები სხვადასხვა დონეზე განსხვავდება; მაგალითად, პრეზენტაციის დონე არ უნდა შეიცავდეს სენსიტიურ მონაცემებს, ამ ტიპის ინფორმაციის ადგილი არის ბიზნესლოგიკისა და მონაცემებთან წვდომის დონეებზე. N- და 3-დონიანი არქიტექტურა მოსახერხებელია, როდესაც ბიზნესლოგიკის გაზიარება გვსურს სხვადასხვა აპლიკაციებში.

3-დონიანი არქიტექტურა რეკომენდებულია შიგა ინტრანეტის აპლიკაციებისთვის, სადაც უსაფრთხოების მოთხოვნები არ ზღუდავს ბიზნესლოგიკის განთავსებას ვებ-ზე ან

Application-სერვერზე. 3-ზე მეტი დონის არსებობა რეკომენ-დებულია ბიზნესლოგიკის მიმართ უსაფრთხოების მაღალი მოთხოვნილებების ან აპლიკაციის რესურსებზე დიდი დატვირთვის შემთხვევაში [35].

2.4. პროგრამული აპლიკაცია ლოგიკური დონეებით (Layers)

განვიხილოთ ცალკეულ ლოგიკურ კომპონენტში (Layer) გადანაწილებული პროგრამული აპლიკაციის ზოგადი სტრუ-ქტურა. დავაკვირდეთ, როგორ ამყარებს ეს კომპონენტები კომუნიკაციას ერთმანეთთან და გარე სისტემებთან/კლიენ-ეტებთან. Layer-შრეს ეხება აპლიკაციის ფუნქციონალის ლოგიკური დაყოფა და არა კომპონენტების ფიზიკურ მისამართებზე დანაწილება. ლოგიკური შრეები (Layer) შეიძლება განსხვავებულ ან საერთო ფიზიკურ დონეზე (Tier) იმყოფებოდნენ. საკმაოდ ხშირია შემთხვევები, როდესაც რამდენიმე Layer-შრე იმყოფება ერთსადაიმავე სერვერზე [28,31,32].

პროგრამული აპლიკაციის ტიპის მიუხედავად, იქნება ის სამომხმარებლო ინტერფეისით, თუ წარმოადგენს სერვის-აპლიკაციას და მხოლოდ სერვისებზე იძლევა წვდომას, ნებისმიერ შემთხვევაში დიზაინი გულისხმობს აპლიკაციის კომპონენტების დეკომპოზიციას ლოგიკურ ჯგუფებში. ამ ლოგიკურ ჯგუფებს უწოდებენ შრეებს. შრეები გვეხმარება განვასხვაოთ კომპონენტების მიერ უზრუნველყოფილი ფუნქციონალი და აიოლებს პროგრამული ლოგიკის ხელახლა გამოყენებას. თითოეული ლოგიკური შრე, თავის

მხრივ, შეიცავს ქვე-შრეებს, რომლებიც სპეციფიკურ დავალე-ბებს ასრულებს.

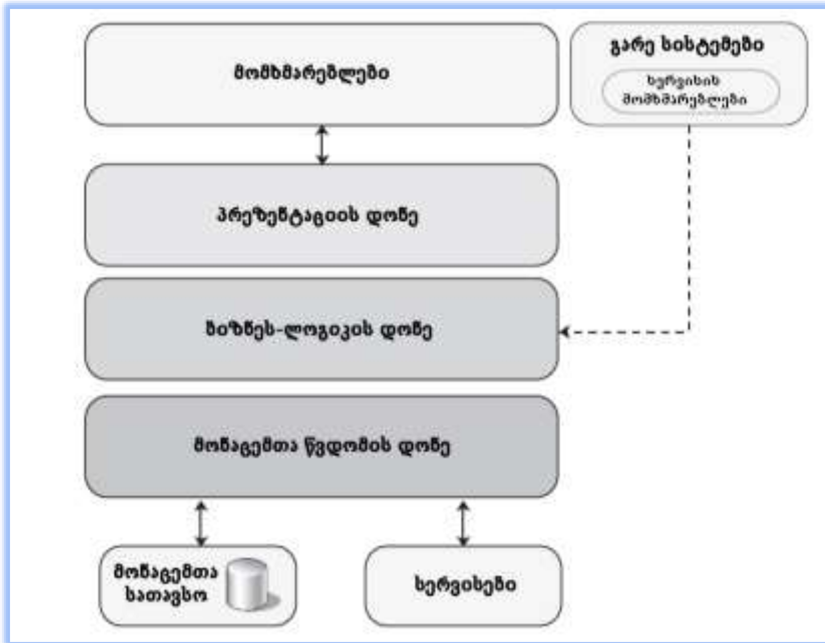
პროგრამული უზრუნველყოფის არქიტექტორის მიზა-ნია ზოგადად გაეცნოს ლოგიკური კომპონენტების ტიპებს, მთლიანობაში დაინახოს სისტემის არქიტექტურა, წარმოდ-გენა შეექმნას ურთიერთდაკავშირებული კომპონენტების ერთობლივ მუშაობაზე. ამის შემდეგ, ახალი პროგრამული დანართის კონსტრუირებას არქიტექტორი უკვე მზა კარკასიდან დაიწყებს.

- *პრეზენტაციის-, ბიზნეს- და მონაცემთა წვდომის შრეები*

აბსტრაქტული კუთხით რომ შევხედოთ, ნებისმიერი სისტემა არის ურთიერთთანამშრომლობაში მყოფი კომპონენ-ტები. 2.3 ნახაზზე ნაჩვენებია შრეების გამარტივებული წარმოდგენა და კავშირები: მომხმარებელთან, სხვა აპლიკაციებთან, რომლებიც იყენებს მოცემული აპლიკაციის ბიზნეს-ლოგიკის შრის სერვისებს, მონაცემთა სათავსოები (როგორიცაა რელაციური მონაცემთა ბაზები, ვებ-სერვისები, რომლებიც აპლიკაციას ამარაგებს მონაცემებით) [33-35].

ლოგიკური შრეები შესაძლოა იმყოფებოდეს ერთიდა-იმავე ან განცალკევებულ ფიზიკურ დონეებზე. თუ სხვადასხვა ფიზიკურ დონეზეა განთავსებული, მაშინ არქიტექტურაში საჭიროა ამ ფაქტორის გათვალისწინება.

როგორც სქემაზეა ნაჩვენები, პროგრამული აპლიკაცია შეიძლება შედგებოდეს მთელი რიგი შრეებისგან.



ნახ. 2.3. ლოგიკური შრეების შემცველი სისტემის არქიტექტურული ხედვა

გავრცელებული 3-დონიანი დიზიანი შედგება შემდეგი კომპონენტებისგან:

– *პრეზენტაციის დონე* (Presentation Layer). ეს დონე შედგება სამომხმარებლო ინტერფეისზე ორიენტირებული ფუნქციონალისგან, რომელიც პასუხისმგებელია მომხმარებელსა და სისტემას შორის ურთიერთქმედებაზე. პრეზენტაციის დონე ხიდის როლს თამაშობს და მომხმარებელს აწვდის ბიზნეს-ლოგიკის დონეზე აღწერილ ძირითად ფუნქციონალს;

– *ბიზნეს-ლოგიკის დონე* (Business Layer). ამ დონეზე იმპლემენტირებულია სისტემის ძირითადი ფუნქციონალი და ინკაფსულირებულია შესაბამისი ლოგიკა. ზოგადად, ბიზნესლოგიკის დონე შედგება კომპონენტებისგან, რომლებიც იძლევა სერვისის ინტერფეისებს, რათა სხვა გამომძახებელმა სისტემებმა შეძლოს მათ გამოყენება;

– *მონაცემთა წვდომის დონე* (Data Layer). ამ დონეზე უზრუნველყოფილია სისტემის ფარგლებში დაჰოსტილ მონაცმთა სათავსოებზე და სხვა სისტემების მიერ მოწოდებულ მონაცემებზე წვდომა (შესაძლოა ეს წვდომა სერვისების გავლით ხდებოდეს). მონაცემთა წვდომის დონე იძლევა Generic ტიპის ინტერფეისებს, რომლებსაც მოიხმარს ბიზნესლოგიკის დონის კომპონენტები.

➢ *სერვისები და შრეები*

ზედა დონის პერსპექტივიდან დანახული სერვისარქიტექტურით აგებული პროგრამული უზრუნველყოფა წარმოგვიდგება რამდენიმე სერვისის ერთობლიობად. ყოველი მათგანი სხვა სერვისებთან კომუნიკაციაშია შეტყობინებების (Message) მიმოცვლით. კონცეპტუალური თვალსაზრისით, *სერვისები* შეგვიძლია ჩავთვალოთ პროგრამული აპლიკაციის *კომპონენტებად.*

დეტალურად თუ ჩავუღრმავდებით და შიგნიდან შევხედავთ, სერვისები შედგება პროგრამული კომპონენტებისაგან, ისევე, როგორც ნებისმიერი სხვა აპლიკაცია და ეს კომპონენტები ლოგიკურად დაჯგუფებულია სხვადასხვა

დონეზე: *პრეზენტაციის, ბიზნეს-ლოგიკისა და მონაცემთა წვდომის დონეებზე.*

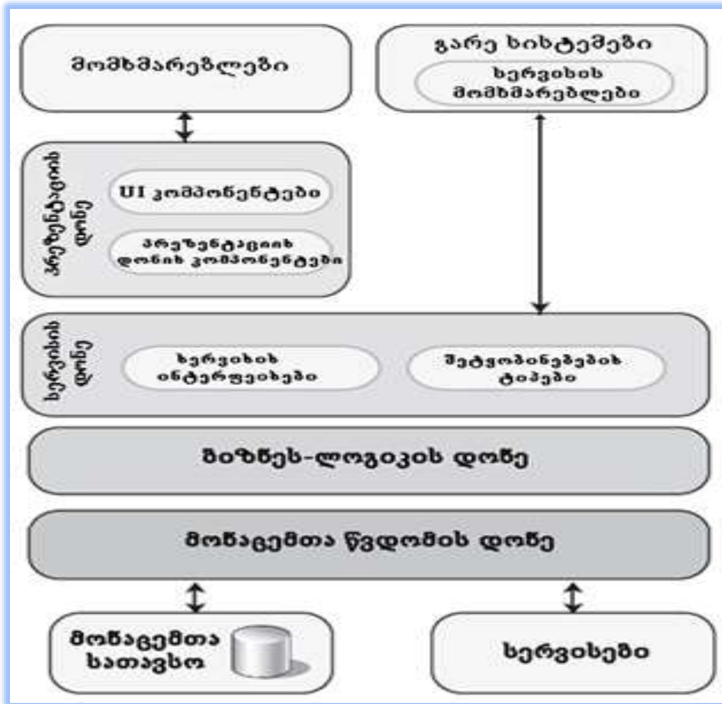
სხვა პროგრამული უზრუნველყოფები სერვისებს მოიხმარს, მაგრამ არ იცის მათი შიგა იმპლემენტაციის ლოგიკა. ამრიგად, შრეების არქიტექტურული დიზიანი თანაბრად ერგება სერვისზე-ორიენტირებულ პროგრამულ აპლიკაციებსაც [28,31,36].

როდესაც პროგრამულმა აპლიკაციამ სხვა პროგრამულ დანართებს უნდა მიაწოდოს სერვისები, და ამასთან ერთად კლიენტისათვის უშუალოდ ხელმისაწვდომი უნდა იყოს პრეზენტაციის დონიდანაც – ამ სიტუაციში გავრცელებული მიდგომაა *სერვისის დონის დამატება* (ნახ.2.4).

სერვისის დონე აპლიკაციის ბიზნეს-ლოგიკის ფუნქციონალს იძლევა ალტერნატიული გზით და კლიენტებს საშუალებას აძლევს განსხვავებული არხით გავიდნენ აპლიკაციის ფუნქციონალზე.

მოცემულ სცენარში მომხმარებელს შეუძლია პროგრამულ აპლიკაციასთან კავშირი დაამყაროს პრეზენტაციის დონიდან; გარდა ამისა, გათვალისწინებულია გარე მომხმარებლის/გარე სისტემების ფაქტორიც, რომელთაც ბიზნეს ლოგიკის დონესთან კომუნიკაცია შეუძლია აწარმოოს სერვისის დონიდან, სერვისის ინტერფეისის გავლით.

ზოგიერთ შემთხვევაში პრეზენტაციის დონე ბიზნეს-ლოგიკის დონესთან კომუნიკაციას ამყარებს სერვისის დონის გავლით. თუმცა, ეს სავალდებულო მოთხოვნას არ წარმოადგენს [28].



ნახ. 2.4. პროგრამულ უზრუნველყოფაში
სერვსის დონის დამატება

2.5. არქიტექტურის დაგეგმვის საფეხურები

პროგრამული აპლიკაციის დაგეგმარების საწყის ეტაპზე პირველი ამოცანაა *აბსტრაქციის ზედა დონეზე* არსებული ფუნქციონალების დაჯგუფება ლოგიკურ დონეებად. შემდეგ ეტაპზე ყოველი დონისთვის აღიწერება *ღია წვდომის მქონე ინტერფეისები* (Public Interface). შემდეგ კი უნდა გადაწყდეს თუ როგორ მოხდება Production-სერვერზე აპლიკაციის

განთავსება (Deployment პროცესი). ბოლოს ხდება აპლიკაციის Layer-დონეებსა და Tier-ფიზიკურ დონეებს შორის კომუნიკაციის პროტოკოლების არჩევა.

მიუხედავად იმისა, რომ პროგრამული აპლიკაციის სტრუქტურა და ინტერფეისები დროთა განმავლობაში შეიცვლება (განსაკუთრებით *Agile Development* მიდგომით სარგებლობისას), ქვემოთ ჩამოთვლილი საფეხურები გარანტიას იძლევა, რომ ძირითადი პუნქტები თავიდანვე გვექნება მხედველობაში მიღებული [28,31].

- *ბიჯი 1 – ლოგიკური დონეების სტრატეგიის არჩევა.*

ლოგიკურ დონეებად დაყოფა პროგრამულ დანართს მატებს მოქნილობას, აიოლებს შემდგომ მხარდაჭერას და ტესტირებას;

- *ბიჯი 2 – სავალდებულო დონეების განსაზღვრა.*

საერთო ფუნქციონალის დაჯგუფება ლოგიკურ დონეებზე; პრეზენტაციის, ბიზნესლოგიკის, სერვისებისა და მონაცემებთან წვდომის ფუნქციონალის გამიჯვნა. გარდა ძირითადი ლოგიკური დონეებისა, ზოგიერთ აპლიკაციაში შევხვდებით უფრო სპეციფიკურ ლოგიკურ დონეებს, მაგალითად, ანგარიშგებები, ინფრასტრუქტურა, კონფიგურაცია/მართვა და ა.შ.;

- *ბიჯი 3 – კომპონენტების გადანაწილება ლოგიკურ დონეებზე.* აუცილებლობის შემთხვევაში ლოგიკურ კომპონენტებს ცალკეულ ფიზიკურ სერვერებზე ათავსებენ. ამას გარკვეული მიზანი აქვს: უსაფრთხოება, ფიზიკური შეზღუდვები, საერთო ბიზნეს-ლოგიკა, მაშტაბურობა.

ამგვარი დანაწილება *სავალდებულო* არ არის და რიგ შემთხვევებში რამდენიმე ლოგიკური დონის ერთ ფიზიკურ სეგმენტზე განთავსებაც შესაძლებელია (მაგალითად, ვებ-აპლიკაციების შემთხვევაში ბიზნეს-ლოგიკის და პრეზენტა-ციის დონეები ერთ ფიზიკურ მისამართზე თავსდება);

- *ბიჯი 4 – ზედმეტი დონის ამოკლება.*

ზოგ შემთხვევაში სასურველია ლოგიკური დონეების სტრუქტურის შემსუბუქება და ზედმეტი დონის ამოკლება. მაგალითად, მწირი ბიზნესლოგიკის შემცველი აპლიკაცია არ საჭიროებს ბევრ შრეს და შეიძლება ბიზნესლოგიკისა და პრეზენტაციის დონეების გაერთიანება;

- *ბიჯი 5 – დონეებს შორის ურთიერთკავშირის წესების განსაზღვრა.* ამ პუნქტის მთავარი დანიშნულებაა წრიულ დამოკიდებულებათა აღმოფხვრა. ასეთი დამოკიდებულების მაგალითია, როდესაც ორი დონე შეიცავს დამოკიდებულებას მესამე დონის კომპონენტებზე. ზოგადი წესის თანახმად ლოგიკურ დონეებს შორის დაშვებულია მხოლოდ ცალ-მხრივი კავშირი;
- *ბიჯი 6 – ზოგადი ფუნქციონალის იდენტიფიცირება* (Cross-Cutting Concerns). ამ ტიპის ფუნქციონალი, როგორც წესი, რამდენიმე ლოგიკურ დონეზე ვრცელდება, მაგალი-თად: კეშირება, ვალიდაცია, გამონაკლისების (Exteptions) მართვა, აუთენტიფიკაცია. მნიშვნელოვანია Cross-Cutting ფუნქციონალის იდენტიფიცირება და ცალკე კომპონენტში გატანა. თავიდან უნდა ავირიდოთ საერთო ფუნქციონალის კოდის მოხვედრა სხვა დანარჩენ ლოგიკურ დონეებზე.

რეკომენდებულია, ერთი დონიდან, საჭიროების დროს, Cross-Cutting კომპონენტების გამოძახება;

- *ბიჯი 7 - დონეებს შორის ინტერფეისების აღწერა.*

ინტერფეისები თავიდან გვარიდებს კომპონენტების მჭიდრო გადაბმულობას. ღია წვდომის მქონე ინტერფეისის მიწოდებით არ ვამჟღავნებთ შიგა იმპლემენტაციის დეტა-ლებს. ასეთ მოვლენას *აბსტრაქციას* უწოდებენ. ამ დანიშ-ნულებას ემსახურება abstract base კლასების ან ინტერ-ფეისების აღწერა, Dependency Injection არქიტექტურული ნიმუშის გამოყენება, Message-Based კომუნიკაცია უშუალოდ მეთოდებსა და Property-წევრებთან;

- *ბიჯი 8 – აპლიკაციის ფიზიკურად განთავსების (Deploy-ment) სტრატეგიის არჩევა.* არსებობს ამ პროცესის წინასწარ განსაზღვრული სტანდარტები, რისი ცოდნაც დაგვეხმარება რეალურ სიტუაციებში გადაწყვეტილებათა მიღებისას;
- *ბიჯი 9 – კომუნიკაციის პროტოკოლების არჩევა.*

პროტოკოლები გამოიყენება Layer-დონეებს შორის კომუნიკაციის გასამართავად. ისინი გადამწყვეტ როლს თამაშობს პროგრამული აპლიკაციის წარმადობის, უსაფრ-თხოების და საიმედოობის თვალსაზრისით. როდესაც კომპონენტები ერთსადაიმავე ფიზიკურ დონეზე (Tier) იმყოფება, მათ შორის უშუალო კომუნიკაცია დაშვებულია და რისკს არ წარმოადგენს. თუმცა, სხვადასხვა ფიზიკურ სერვერებზე და კლიენტის მანქანებზე განაწილების შემთხვე-ვაში, აუცილებლად ყურადღებით უნდა მოვეკიდოთ კომუნიკაციის პროტოკოლებს და მათ საიმედოობას.

2.6. RIA აპლიკაციების დაპროექტება

ტერმინი RIA-აპლიკაცია ითარგმნება როგორც „მდიდარი“ ინტერნეტ აპლიკაცია („Rich Internet Application”). განვიხილოთ RIA-აპლიკაციების ძირითადი დანიშნულება, გავაანალიზოთ მისი კომპონენტები და RIA არქიტექტურის შემუშავების ძირითადი რეკომენდაციები [28,31,33-35].

RIA აპლიკაცია უზრუნველყოფს მდიდარ გრაფიკულ სამომხმარებლო ინტერფეისს და Streaming Media სერვისებს, ამავდროულად, აქვს ვებ-აპლიკაცისთვის დამახასიათებელი თვისებები: საწარმოო სერვერზე იოლად განთავსებადობა და შემდგომი მხარდაჭერის მოქნილი პროცესი.

RIA-აპლიკაციები ეშვება ბრაუზერის Plug-In გარემოში, რითაც განსხვავდება სხვა ვებ-ტექნოლოგიებისგან, რომლე-ბიც იყენებს ბრაუზერის კოდს (მაგ. AJAX ტექნოლოგია).

ტერმინი Plug-In აღწერს კომპიუტერულ პროგრამას, რომელიც ხელს უწყობს სხვა პროგრამული დანართების მუშაობის გაუმჯობესებას ან ფუნქციონალის გამდიდრებას.

ტიპური RIA-იმპლემენტაცია ვებ-ინფრასტრუქტურას იყენებს კლიენტის-მხარის აპლიკაციასთან ერთობლიობაში, რომელიც პრეზენტაციის დონეს უზრუნველყოფს.

ბრაუზერის Plug-In დანამატი იძლევა მდიდარ გრაფიკულ ინტერფეისთან სამუშაო ბიბლიოთეკებს და კონტეინერის როლს თამაშობს, რათა უსაფრთხოების დაცვის მიზნით, ლოკალურ რესურსებზე შეზღუდოს წვდომა.

RIA-აპლიკაციებს ახასიათებს უფრო კომპლექსური და მდიდარი კლიენტის მხარის (Client-Side) კოდის მხარდაჭერა,

ვიდრე შესაძლებელი იყო ტრადიციული ვებ-აპლიკაციებისთვის, რის შედეგადაც ვებ-სერვერზე დატვირთვა მსუბუქდება, ნაწილდება კლიენტის მანქანაზე. 2.5 ნახაზზე ნაჩვენებია ტიპური RIA-იმპლემენტაციის სტრუქტურა [30,31]:

ტიპური RIA-აპლიკაცია დანაწილებულია 3 ლოგიკურ დონეზე:

- პრეზენტაციის;
- ბიზნეს-ლოგიკის და
- მონეცემთა წვდომის დონეებზე.

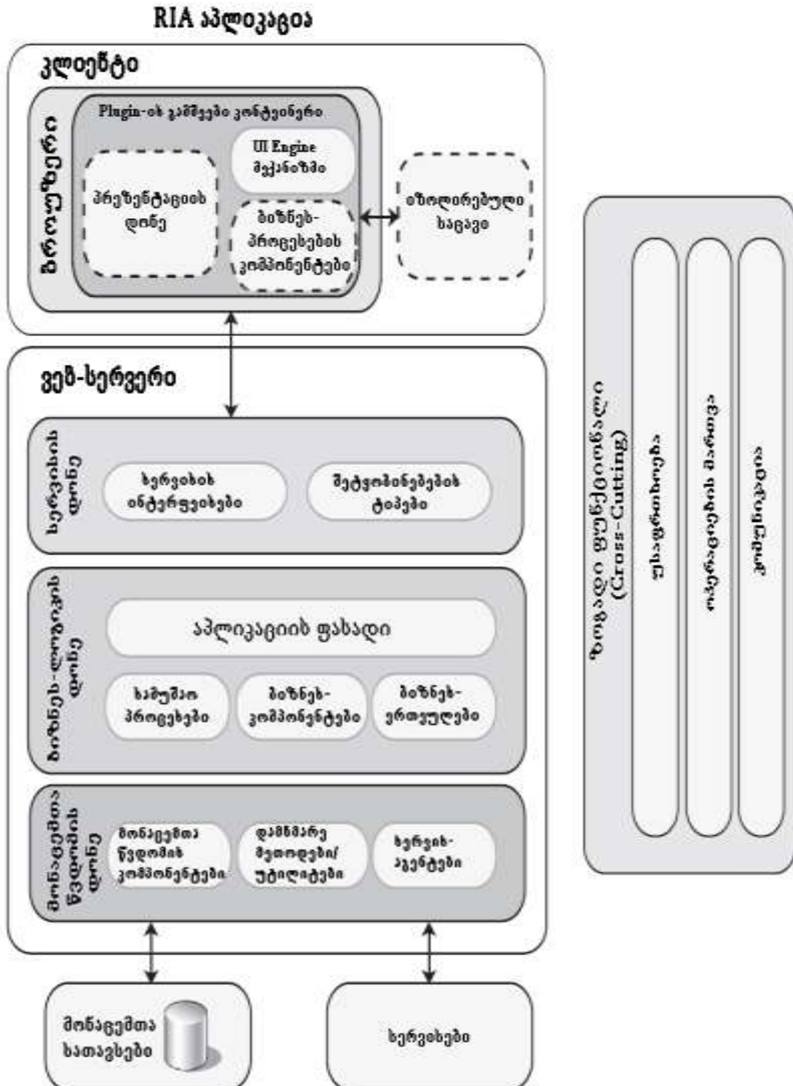
პრეზენტაციის დონე შეიცავს სამომხმარებლო ინტერფეისს (UI) და პრეზენტაციის ლოგიკის კომპონენტებს;

ბიზნეს-ლოგიკის დონე შეიცავს ბიზნეს-ლოგიკას, სამუშაო პროცესებს და ბიზნეს-ერთეულებს; *მონაცემთა წვდომის დონე* შეიცავს მონაცემთა წვდომის და სერვის-აგენტ კომპონენტებს.

***განსაზღვრება** (ინფრასტრუქტურა და* არქიტექტურა*):*

პროგრამული აპლკაციის *ინფრასტრუქტურა* აღწერს კომპონენტების რეალურ ნაკრებს, რომელიც ქმნის სისტემას, ხოლო *არქიტექტურა* აღწერს კომპონენტების დიზაინს და მათ ურთიერთობებს.

მოკლედ, სისტემა აგებულია ინფრასტრუქტურაზე, რომელსაც აქვს განსაკუთრებული არქიტექტურა.



ნახ. 2.5. RIA აპლიკაციის არქიტექტურა (წყვეტილი ხაზები აღნიშნავს არასავალდებულო კომპონენტებს)

შენიშვნა: RIA-სთვის დამახასიათებელია ზოგიერთი ბიზნეს-პროცესის გადატანა კლიენტის მხარეს (შესაძლოა, მონაცემთა წვდომის ლოგიკისაც კი). პროგრამული აპლიკაციის დანიშნულებიდან გამომდინარე, კლიენტი შეიძლება ნაწილობრივ ან მთლიანად შეიცავდეს ბიზნეს-ლოგიკისა და მონაცემთა წვდომის ფუნქციონალს. ზემოთ მოყვანილ სქემაზე ნაჩვენებია ბიზნეს-პროცესების ნაწილობ-რივი იმპლემენტაცია კლიენტის მხარეს [28,30].

RIA-აპლიკაციები შეიძლება იმპლემენტირებული იყოს კომპლექსური ბიზნეს-სერვისების (Back-End) შრეზე დადე-ბული „თხელი" ინტერფეისით ან წარმოადგენდეს რთულ აპლიკაციას, რომელიც პროცესების უმრავლესობას თავად ასრულებს და უკანა მხარის სერვისებთან კომუნიკაციას მხოლოდ ინფორმაციის წამოღების ან შენახვის მიზნით ამყარებს.

მიუხედავად არქიტექტურული სხვაობებისა, RIA აპლიკაციებზე ვრცელდება „კარგი დიზაინის" შემუშავების რეკომენდაციები.

მათი უმრავლესობა ეფუძნება არქიტექტურული ნიმუშების (Design Patterns) გამოყენებას, რაც ხელს უწყობს კომპონენტების ურთიერთგამიჯვნას, ფუნქციონალის ხელ-მეორედ გამოყენებას, ამცირებს მჭიდრო დამოკიდებუ-ლებებს, აიოლებს აპლიკაციის შემდგომ მხარდაჭერას და ავტომატურ ტესტირებას [37-39].

2.7. RIA დიზაინის ზოგადი რეკომენდაციები

RIA აპლიკაციების შემუშავებისას გათვალისწინებულ უნდა იქნას *რამდენიმე ასპექტი,* რომელიც დაგვეხმარება პროგრამული აპლიკაციის შესაბამისობის დარწმუნებაში მოთხოვნილებებთან და ეფექტურად მუშაობს RIA-სთვის დამახასიათებელ სცენარებში [28]:

- *RIA-ზე არჩევანის გაკეთება აუდიენციიდან გამომდინარე, მდიდარი ინტერფეისი და იოლი განთავსებადობა (Deployment).*

ვიფიქროთ RIA არქიტექტურაზე, თუკი ძირითადი აუდიენცია იყენებს RIA-სთან თავსებად ბრაუზერებს. იმ შემთხვევაში, თუ აუდიენციის ნაწილი სხვა ბრაუზერით სარგებლობს (არა-RIA თავსებადი), უნდა ავწონ-დავწონოთ, რამდენად გვიღირს ამ ტიპის აუდიენციის დაკარგვა და ვიფიქროთ პროგრამული დანართის ალტერნატიული ტიპის შეთავაზებაზე (მაგალითად, ვებ-აპლიკაცია AJAX ტექნოლოგიის გამოყენებით). RIA იმპლემენტაცია კარგად არის მორგებული ვებ-სცენარებზე: Deployment პროცესისა და მხარდაჭერის სიმარტივით არ ჩამოუვარდება სტანდარტული ვებ-აპლიკაციების ტექნოლოგიას, ხოლო ვიზუალიზაციის მხრივ ბევრად აჭარბებს სტანდარტული HTML-ის შესაძლებლობებს. სამომხმარებლო ინტერფეისი ბევრად უფრო მდგრადი და საიმედოა, სხვადასხვა ბრაუზერში ტესტირებას აღარ საჭიროებს [36];

- *დაპროექტება ვებ-ინფრასტრუქტურის სერვისების გამოყენების პერსპექტივით.*

RIA იმპლემენტაციები ვებ-აპლიკაციების მსგავს ინფრასტრუქტურას იყენებს. ტიპური RIA-აპლიკაცია გამოთვლითი პროცესების ნაწილს კლიენტის მანქანაზე ასრულებს, მაგრამ ასევე, კომუნიკაციას ამყარებს სხვა სერვისებთანაც, მაგალითად, ინფორმაციის შენახვა მონაცემთა სათავსოში;

- *დაპროექტება კლიენტის მხარეს შესრულებული გამოთვლითი პროცესების უპირატესობის გათვალისწინებით.*

RIA-აპლიკაციების უპირატესობა მდგომარეობს იმაში, რომ გამოთვლითი პროცესების ნაწილი კლიენტის მანქანაზე სრულდება, ამიტომ ამ რესურსის გამოყენებას მაქსიმალურად უნდა ვეცადოთ და ფუნქციონალის რაც შეიძლება დიდი ნაწილი გადავიტანოთ კლიენტის მხარეს. ამავე დროს, გასათვალისწინებელია, რომ კლიენტის კოდის „მოტყუება" საკმაოდ იოლია და საჭიროა სიფრთხილე სენსიტიურ მონაცემებთან – ბიზნეს წესები კვლავაც სერვერის მხარესაა;

- *დაპროექტება ბრაუზერის Sandbox გარმოში გაშვების გათვალისწინებით.*

სტანდარტულად, RIA-იმპლემენტაციებს აქვს უსაფრთხოების დაცვის მაღალი ხარისხი. ამიტომ, კლიენტის მანქანის რესურსებზე წვდომა შეზღუდულია. მაგალითად კამერები, ვიდეო-აქსელერატორები და კომპიუტერის სხვა აპარატურული საშუალებები. ლოკალურ ფაილურ სისტემასთან წვდომა შეზღუდულია, Local Storage სათავსოს მაქსიმალურ ზომაზე დაწესებულია ლიმიტი [44];

- *სამომხმარებლო გარემოს სირთულის განსაზღვრა.*

RIA იმპლემენტაციები ოპტიმალურად მუშაობს ერთიანი ოპერაციული გარემოს შემთხვევაში, ანუ სხვაგვარად რომ ვთქვათ, როდესაც ყველა ოპერაცია ერთ ფანჯარაში სრულდება (Single Screen). ოპერაციების რამდენიმე ფანჯარაში გადანაწილება დამატებით კოდირებას და ფანჯრების მონაცვლეობის პროცესის გათვლას საჭიროებს. მომხმარებელს იოლი ნავიგაციის საშუალება უნდა ჰქონდეს და დროის ნებისმიერ მომენტში სისტემის ხელახლა ჩატვირთვის გარეშე უნდა შეეძლოს სასურველ მისამართზე დაბრუნება.

მრავალგვერდიანი სამომხმარებლო ინტერფეისებისთვის გამოიყენება Deep Linking მეთოდები, ისტორიის სია და ბრაუზერის წინა და უკანა მიმართულებით გადაადგილების ღილაკები (რათა ეკრანებს შორის ნავიგაციისას მომხმარებელს თავიდან ავაცილოთ დაბნეულობა) [11,28];

- *პროგრამული აპლიკაციის გამოყენების სცენარების გათვალისწინებით ინტერაქტიულობის და სისწრაფის გაზრდა.*

განვიხილოთ პროგრამული აპლიკაციის გამოყენების სცენარები, რათა გადავწყვიტოთ მისი კომპონენტებად დაყოფა და დატვირთვის გადანაწილება. ასევე, მონაცემთა კეშირება და ბიზნეს-ლოგიკის პორციის გატანა კლიენტის მხარეს. გადმოწერისა და აპლიკაციის ჩატვირთვის დროის შესამცირებლად ფუნქციონალი გადავანაწილოთ ცალკეულ ჩამოტვირთვად კომპონენტებში;

- *დაპროექტება იმ შემთხვევების გათვალისწინებით, როდესაც ბრაუზერის Plug-In არ არის დაინსტალირებული.*

რადგან RIA-იმპლემენტაცია ბრაუზერის Plug-In გარემოს საჭიროებს, უნდა ვივარაუდოთ, რომ ზოგიერთ კლიენტს შესაძლოა არ ჰქონდეს უფლება, ან არ სურდეს მისი ინსტალაცია. რეკომენდებულია ამგვარი შემთხვევების გათვალისწინება და სამომხმარებლო გარემოში გამაფრთხილებელი შეტყობინების გამოტანა ან ალტერნატიული ვებ-ინტერფეისის მიწოდება.

2.8. RIA არქიტექტურის სპეციფიკა

RIA აპლიკაციის დაპროექტებისას გასათვალისწინებელია ზოგადი მიდგომები, რომელიც შემუშავებულია RIA კომპონენტებთან მიმართებაში.

მომდევნო სექციებში განვიხილოთ არქიტექტურული რეკომენდაციები თითოეული მნიშვნელოვანი არეალისთვის [28,31,37-39].

- *ბიზნეს-ლოგიკის დონე*

უმრავლეს სცენარებში RIA აპლიკაციები უკავშირდება მისთვის საჭირო მონაცემებს. მონაცემთა სათავსოები, როგორც წესი, აპლიკაციის გარეთ იმყოფება. მიუხედავად იმისა, რომ ინფორმაციის ნატურა განსხვავდება, მისი წამოღება ხშირად ბიზნეს-სისტემიდან ხორციელდება. ბიზნეს- და სერვისის-დონეებთან მუშაობისას გასათვალისწინებელია შემდეგი რეკომენდაციები [28]:

- დავადგინოთ პროგრამულ დანართში რა სახის ბიზნეს ლოგიკა და სერვისის ინტერფეისები გვექნება. კლიენტის მხარის ბიზნეს-ლოგიკის დონე სერვისის ინტერფეისს სერვის-აგენტის საშუალებით უნდა მიწვდეს. როგორც წესი, სერვის-აგენტები მიიღება სერვისის პროქსი კლასის (Proxy) გენერაციით [37];
- თუ ბიზნეს-ლოგიკა სენსიტიურ ინფორმაციას არ შეიცავს, შესაძლებელია კლიენტის მხარეს ბიზნეს-ლოგიკის ნაწილის გატანა, რაც გამოიწვევს პროგრამული დანართის მუშაობის სისწრაფის და ინტერაქტიულობის ზრდას. იმ შემთხვევაში, თუ ბიზნეს-ლოგიკა შეიცავს სენსიტიურ ინფორმაციას, მისი განთავსება მთლიანად Application-სერვერზე უნდა მოხდეს;
- თუ ბიზნეს-ლოგიკა კლიენტისა და სერვერის მხარეს დუბლირდება, სასურველია კოდის შემუშავებისას ორივე მხარეს ერთიდაიგივე პროგრამული ენა გამოვიყენოთ, რათა მივაღწიოთ ერთიან ლოგიკას. დომეინ-მოდელები, რომლებიც შესაძლებელია, კლიენტისა და სერვერის მხარეს პარალელურად არსებობდნენ, მაქსიმალურად მსგავსი უნდა იყოს;
- უსაფრთხოების მიზნით, არ უნდა დავუშვათ სენსიტიური ბიზნეს-ლოგიკის დაუშიფრავად განთავსება კლიენტის მხარე. XAP ფაილები გადმოწერადია და კოდის დეკომპილაციის პროცესი საკმაოდ იოლია. ამიტომ, რეკომენდებულია სენსიტიური ინფორმაციის სერვერის მხარეს იმპლემენტირება და მასზე წვდომის დაშვება ვებ-სერვისების გავლით.

➢ *კომუნიკაცია*

RIA იმპლემენტაციებში რეკომენდებულია სერვისებზე *ასინქრონული* გამოძახების მოდელის გამოყენება, რათა თავიდან ავიცილოთ ბრაუზერის პროცესებზე ბლოკის დადება. კროს-დომეინ, პროტოკოლისა და სერვისების ეფექტურად მუშაობის საკითხები არქიტექტურის ნაწილია და განსახილველ საკითხს წარმოადგენს. თუკი RIA-იმპლემენტაცია ამის საშუალებას იძლევა, რეკომენდებულია სხვადასხვა ოპერაციებისთვის განსხვავებული Thread გარემოს გამოყენება. კომუნიკაციის საკითხებზე მუშაობისას გასათვალისწინებელია შემდეგი რეკომენდაციები [36]:

- ოპერაციების შესრულების ხანგრძლივობის მატებისას, გასათვალისწინებელია მათი ფონად (Background Thread) ან ასინქრონულად გაშვება. ამით თავიდან ავიცილებთ სამომ-ხმარებლო ინტერფეისის (UI) დაბლოკვას;
- სენსიტიური ინფორმაციის დაცვის მიზნით რეკომენ-დებულია Internet Protocol Security (IPSec) ან Secure Socets Layer (SSL) გამოყენება, რათა დავიცვათ არხი და დავშიფროთ ინფორმაცია და ელექტრონული ხელმოწერების საშუალებით ავკრძალოთ ინფორმაციაზე არასასურველი პირების წვდომა;
- თუ RIA-კლიენტი უკავშირდება განსხვავებულ სერვერს (ანუ განსხვავდება სერვერისგან, საიდანაც RIA-კლიენტის გადმოწერა მოხდა), უნდა დავრწმუნდეთ, რომ ჩართულია კროს-დომეინ კონფიგურაციის მექანიზმი, რითაც წვდომა ჩართულია სხვა სერვერებზე/დომენებზე.

➢ *პრეზენტაციის დონე*

RIA აპლიკაციები ეშვება ბრაუზერში და ამრიგად, შემოიფარგლება ბრაუზერის შესაძლებლობებით. ამ მიზე-ზით, მათი მუშაობა ოპტიმალურია ერთიანი, საერთო ცენტრალური ინტერფეისის არსებობის პირობებში.

მრავლი გვერდის შემცველი პროგრამული აპლიკაცი-ები დამატებითი სამუშაოების ჩატარებას საჭიროებს, რათა დაიგეგმოს კავშირი გვერდებს შორის. პრეზენტაციის დონე-ზე მუშაობისას გასათვალისწინებელია რეკომენდაციები [28]:

- გამოვიყენოთ პრეზენტაციის დონის ლოგიკის განცალ-კევების სტანდარტი (Separated Presentation Pattern), რათა პროგრამული აპლიკაციის ვიზუალური წარმოდგენა გავმიჯ-ნოთ პრეზენტაციის ლოგიკისაგან;
- გამოვიყენოთ მონაცემთა ბმის ფუნქციონალის უპირა-ტესობები, რათა ინფორმაცია მაქსიმალურად ეფექტურად მივაწოდოთ მომხმარებელს, განსაკუთრებით კი ცხრილებისა და მრავალსტრიქონიანი სიების წარმოდგენების დროს. მონაცემთა ბმის მექანიზმი ამცირებს პროგრამული კოდის რაოდენობას, აიოლებს დაპროგრამების პროცესს და ამცირებს შეცდომების ალბათობას. გარდა ამისა, მონაცემთა ბმის მექანიზმი სხვადასხვა წარმოდგენასა თუ ფორმაზე არსებულ ინფორმაციას ავტომატურ სინქრონიზაციაში ამყოფებს. ორ-მხრივი ბმის გამოყენება რეკომენდებულია სიტუაციებში, როდესაც მომხმარებელს უნდა შეეძლოს ჩანაწერების რედაქტირება და მოდიფიკაცია;

• მრავალი გვერდის არსებობის პირობებში რეკომენდე-ბულია Deep-Linking ფუნქციონალის გამოყენება, რათა შესაძლებელი იყოს ინდივიდუალური გვერდების უნიკალუ-რად იდენტიფიცირება და ნავიგაცია.

➢ *ვალიდაცია*

ვალიდაცია შესაძლებელია კლიენტის მხარის კოდის ან სერვერზე განთავსებული სერვისების საშუალებით. თუ კლიენტის მხარეს ტრივიალურ ვალიდაციებზე მეტი ფუნქციონალია საჭირო, მაშინ რეკომენდებულია ვალიდა-ციის ლოგიკის იზოლირება დამოუკიდებელ, ჩამოტვირთვად კრებულში (Assembly). ეს აიოლებს ბიზნეს-წესების მართვას. ვალიდაციის საკითხებზე მუშაობისას გასათვალისწინე-ბელია შემდეგი რეკომენდაციები [9-11]:

• ეფექტურობის გაზრდის მიზნით, გამოვიყენოთ კლიენ-ტის მხარის ვალიდაციები, თუმცა უსაფრთხოების მიზნით, ყოველთვის გავამყაროთ სერვერის მხარის ვალიდაციებით. ზოგადად, მხედველობაში გვქონდეს, რომ ნებისმიერი კლინტის მიერ კონტროლირებადი ჩანაწერი სარისკოა და სერვერმა ხელახლა უნდა ჩაატაროს კლიენტიდან გადაცე-მული ინფორმაციის შემოწმება. დაიგეგმოს ნებისმიერი წყაროდან (Query String, Cookies და HTML-კონტროლები) მიღებული ინფორმაციის ვალიდაცია;

• დაიგეგმოს ვალიდაციის მექანიზმები, რომელსაც შეეძლება ჩანაწერების შეზღუდვა, უარყოფა და გაუვნებელ-ყოფა; შესრულდეს შეტანილი ინფორმაციის ვალიდაციები სიგრძეზე, დიაპაზონზე, ფორმატსა და ტიპზე; სერვერზე

დადგინდეს ნდობის არეალი რომლის ფარგლებს გარეთ მოხვედრილი ჩანაწერებისთვის ვალიდაცის ჩატარება სავალდებულო იქნება;

• განვიხილოთ იზოლირებული სათავსოს (Isolated Storage) გამოყენება კლინტისათვის სპეციფიკური ვალიდაციის წესების შესანახად. ხოლო იმ ტიპის წესებისთვის, რომლებიც სერვერის რესურსებზე წვდომას საჭიროებს, ვეცადოთ სერვისის ერთი გამოძახებით მოხდეს ვალიდაციების ჩატარება;

• კლიენტის ვალიდაციების დიდი რაოდენობის შემთხვევაში (განსაკუთრებით როდესაც ვალიდაციის ლოგიკა ცვალებადია) რეკომენდებულია მათი დამოუკიდებელ, ჩამოტვირთვად მოდულში გატანა. შედეგად, მოდულის ჩანაცვლება და განახლება იოლად მოხდება მთლიანი RIA აპლიკაციის ჩამოტვირთვის გარეშე.

➢ *უსაფრთხოების საკითხები*

უსაფრთხოება მოიცავს მთელ რიგ ფაქტორებს, რომლებიც კრიტიკულ დანიშნულებას ატარებს ნებისმიერი ტიპის პროგრამულ დანართში. „მდიდარი“ ინტერნეტ აპლიკაციები (RIA) უნდა დაპროექტდეს უსაფრთხოების საკითხების გათვალისწინებით, როგორიცაა *სენსიტიური ინფორმაციის დაცვა, მომხმარებლის აუტენტიფიკაცია და ავტორიზაცია, სახიფათო კოდებით შეტევისგან თავდაცვა, აუდიტი და მოვლენებისა (Event Logging) და მომხმარებლის ქმედებების (User Activity Logging) ლოგირება.* უსაფრთხოების საკითხებზე მუშაობისას გასათვალისწინებელია შემდეგი რეკომენდაციები [10]:

• შესაბამისი ტექნოლოგიისა და მომხმარებლის სისტემაში დაშვების დადგენა. გავიაზროთ: როდის და როგორ შევა მომხმარებელი სისტემაში, რა ტიპის მომხმარებელი დაიშვება სისტემაში, რა ტიპის უფლებების სიმრავლეებია განსაზღვრული (ადმინისტრატორის, მომხმარებლის და ა.შ.), როგორ მოხდება წარმატებული და წარუმატებელი ავტორიზაციის შემთხვევების შენახვა ლოგირების ცხრილში;

• გავაანალიზოთ აუთენტიფიკაციის შემთხვევები: თუ მომხმარებლები ერთიდაიმავე უფლებამოსილებით (Credentials) დაიშვებიან რამდენიმე პროგრამულ დანართში, რეკომენდებულია Windows Integrated Authentication აუთენტიფიკაციის გამოყენება. ამას გარდა, ალტერნატიულ გზებს წარმოადგენს სერტიფიკატზე დაფუძნებული აუთენტიფიკაციის სისტემის გამოყენება, რომელიც ორგანიზაციაზე მორგებულ გამოსავალს იძლევა;

• სისტემაში მოხვედრილი ინფორმაციის ვალიდაცია, რომელიც მომხმარებლის ან სერვისის მხრიდან შემოვიდა. შესაძლოა საკუთარი ვალიდაციის მექანიზმების შემუშავება ან თავად UI ტექნოლოგიის მიერ შემოთავაზებული ვალიდაციის ფუნქციონალით სარგებლობა;

• პროგრამული დანართის ლოგირების და აუდიტის ფუნქციონალის დაგეგმვა, განსაზღვრა, თუ რა ტიპის ინფორმაცია უნდა მოხვდეს ლოგირების ცხრილში. რეკომენდებულია ლოგირებისას სენსიტიური ინფორმაციის დაშიფრული სახით შენახვა. ხოლო განსაკუთრებით სენსიტიური ინფორმაციის შემთხვევაში, შესაძლებელია ციფრული ხელმოწერის გამოყენებაც.

- *პროგრამული დანართის სერვერზე განთასება (Deployment)*

RIA იმპლემენტაციები ტრადიციული ვებ-აპლიკაციების ანალოგიურ უპირატესობებს იძლევა სერვერზე განთავსებისა და პროგრამული დანართის შემდგომი მხარდაჭერის თვალსაზრისით.

რეკომენდებულია RIA-აპლიკაციის ცალკეულ, დამოუკიდებლად ჩამოტვირთვად მოდულებად დაპროექტება. ამგვარი მოდულების კეშირება და სხვა მოდულით ჩანაცვლება საკმაოდ იოლია და არ საჭიროებს პროგრამული უზრუნველყოფის მთლიანად შეცვლას.

RIA-აპლიკაციის სერვერზე განთავსების საკითხებზე მუშაობისას გასათვალისწინებელია შემდეგი რეკომენდაციები [16]:

- დაიგეგმოს ჩასატარებელი სამუშაოები, როდესაც ბრაუზერის Plug-In არ არის დაინსტალირებული;
- გავაანალიზოთ, როგორ შეიძლება აპლიკაციის ხელახლა განთავსება, მაშინ, როდესაც მისი ეგზემპლიარი გაშვებულია კლიენტის მანქანაზე;
- აპლიკაციის ლოგიკურ მოდულებად დაყოფა და კომპონენტების იოლი ჩანაცვლება (მთლიანად აპლიკაციის გამოცვლის გარეშე);
- ცალკეულ კომპონენტზე ვერსიების განსაზღვრა (Versioning).

➢ *განაწილებულ სერვერებზე განთავსება*

რადგან RIA აპლიკაციები პრეზენტაციის ლოგიკას აკოპირებს ან გადააქვს კლიენტის მანქანაზე, პროგრამული უზრუნველყოფის განთავსების დროს ყველაზე მორგებული სცენარია განაწილებული არქიტექტურა:

– განაწილებულ სერვერებზე განთავსების დროს პრეზენ-ტაციის ლოგიკა კლიენტის მანქანაზე იმყოფება;

– ბიზნესლოგიკა შესაძლოა იმყოფებოდეს კლიენტის მანქანაზე, სერვერზე ან გაზიარებული იყოს ორივეზე – სერვერზეც და კლიენტზეც;

– მონაცემთა წვდომის დონე იმყოფება ვებ-სერვერზე ან Application-სერვერზე.

ზოგადად, ბიზნესლოგიკის ნაწილი (და შესაძლოა მონაცემთა წვდომის ლოგიკის ნაწილიც კი) გადაგვაქვს კლიენტის მხარეს, რითაც პროგრამული დანართის წარმადობა და სისწრაფე იზრდება. ამ შემთხვევაში ბიზნეს-ლოგიკისა და მონაცემთა წვდომის დონეები გავრცობილია კლიენტის მანქანასა და Application-სერვერზე [10,44].

ეს შემთხვევა ილუსტრირებულია 2.6 ნახაზზე.

ნახ. 2.6. RIA აპლიკაციის განთავსება განაწილებულ სერვერებზე

RIA აპლიკაციების განთავსებისას შესაძლოა შეგვხვდეს შემთხვევა, როდესაც ბიზნეს-ლოგიკის რამდენიმე პროგრამულ დანართთან გაზიარება არის საჭირო. ასეთ სიტუაციებში რეკომენდებულია ბიზნეს-ლოგიკის კომპონენტების სერვისის სახით განთავსება სერვერზე, რომელზეც ყველა მომხმარებელ აპლიკაციას ექნება მიმართვის საშუალება.

2.9. RIA აპლიკაციების შესაბამისი არქიტექტურული სტანდარტები

დაპროექტების ძირითადი სტანდარტები (Design Pattern) ორგანიზებულია კატეგორიებად, სექციებში: შრეები (Layers), კომუნიკაცია (Communication), კომპოზიცია (Composition), პრეზენტაცია (Prezentation) (ცხრ.2.1).

ამ სტანდარტების გათვალისწინებით ვხელმძღვანელობთ არქიტექტურული გადაწყვეტილებების არჩევისას [16,23,25-28]:

RIA-აპლიკაციებში გამოყენებული არქიტექტურული სტანდარტები **ცხრ.2.1**

კატეგორია	შესაბამისი სტანდარტი(Design Pattern)
შრეები (Layers)	Service Layer. არქიტექტურული სტანდარტი, რომელშიც სერვისის ინტერფეისი და იმპლემენტაცია გაერთიანებულია ერთ ლოგიკურ შრეზე. სერვისი იმპლემენტირებულია WCF ტექნოლოგიით.
კომუნიკაცია Communication)	**Asynchronous Callback.** გრძელვადიანი სამუშაოების განცალკევებულ Background Thread ფონზე გაშვება და Call Back ფუნქციით

	შესრულებული სამუშაოს შედეგების ამოკითხვა. **Command.** მოთხოვნის დამუშავების ლოგიკის ინკაფსულაცია command ობიექტში, რომელიც იძლევა Common Execution Interface გარემოს.
კომპოზიცია (Composition)	**Composite View.** ინდივიდუალური View გვერდების გაერთიანება კომპოზიტურ View-ში. **Inversion of Control.** ობიექტების სხვა ობიექტებზე/კომპონენტებზე დამოკიდებულებების აღწერა, რომელთა იმპლემენტაცია სავალდებულოა, რათა მოცემული ობიექტის გამოყენება შესაძლებელი გახდეს პროგრამულ აპლიკაციაში.
პრეზენტაცია (Prezentation)	**Application Controller.** ობიექტი, რომელიც შეიცავს სრული სამუშაო პროცესის ლოგიკას და გამოიყენება სხვა კომპონენტების მიერ, რომლებიც მუშაობს Model ობიექტთან და ასახავს შესაბამის View პრეზენტაციას. **Supervising Presenter.** პრეზენტაციის დონის სტრუქტურის გადანაწილება სამ დამოუკიდებელ როლში: View პასუხისმგებელია მომხმარებლთან ურთიერთკავშირზე (User Input); მონაცემთა ბმის მექანიზმით გამოაქვს Model კომპონენტის ინფორმაცია, ხოლო Model კომპონენტში ინკაფსულირებულია ბიზნეს-ლოგიკა; Presenter ობიექტში იმპლემენტირებულია პრეზენტაციის ლოგიკა და

	კოორდინირებას უკეთებს View-სა და Model-ობიექტებს შორის ურთიერთქმედებებს. **Presentation Model.** წარმოადგენს Model-View-Presenter (MVP) არქიტექტურული სტანდარტის ვარიაციას და შექმნილია თანამედროვე სამომხმარებლო ინტერფეისების დეველოპმენტ- პლატფორმებისთვის, სადაც View გრაფიკული დიზაინერის დანიშნულებას უფრო ატარებს, ვიდრე პროგრამული ლოგიკის.

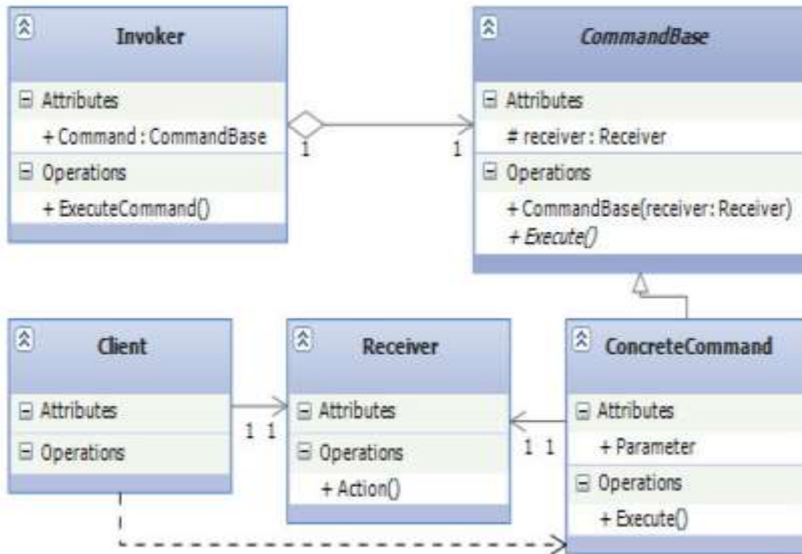
2.10. არქიტექტურული სტანდარტი Command

Command არქიტექტურული სტანდარტი შემომავალ მოთხოვნებზე საპასუხო ლოგიკას აერთიანებს ერთ ობიექტში. Command ობიექტის გამოძახება მოთხოვნის შემოსვლისთანავე ხდება. ამ არქიტექტურული მიდგომის მთავარი დამახასიათებელია ამა თუ იმ პროგრამული ლოგიკის შესასრულებლად საჭირო ინფორმაციის თავმოყრა ერთ ობიექტში. თავად ობიექტი არ ასრულებს რაიმე ოპერაციას, იგი მხოლოდ ინფორმაციას შეიცავს [10,11,44,45].

განვმარტოთ სამი მნიშვნელოვანი ცნება: კლიენტი (Client), გამომძახებელი (Invoker) და მიმღები (Receiver).

კლიენტი ქმნის Command ობიექტს. *გამომძახებელი* წყვეტს თუ როდის უნდა მოხდეს Command ობიექტში აღწერილი მეთოდის გამოძახება. *მიმღები* კი არის კლასის ეგზემპლარი, რომელიც შეიცავს ამ მეთოდის პროგრამულ კოდს.

2.6 ნახაზზე მოყვანილია UML-Class დიაგრამა, რომელიც აღწერს Command არქიტექტურული სტანდარტის იმპლემენტაციას.



ნახ. 2.6. Command არიტექტურული სტანდარტის იმპლემენტაცია

ეს დიაგრამა შედგება 5 კლასისგან:

- კლიენტი (Client): კლასი, რომელიც წარმოადგენს Command არქიტექტურის მომხმარებელს. იგი ქმნის Command ობიექტს და აკავშირებს მიმღებთან (Receiver);
- მიმღები (Receiver): კლასი, რომელმაც იცის თუ როგორ უნდა განახორციელოს შემომავალ მოთხოვნასთან დაკავშირებული ოპერაციები;

- CommandBase: აბსტრაქტული კლასი (ან ინტერფეისი) ყველა Command ობიექტისთვის. იგი შეიცავს ინფორმაციას თუ რომელი მიმღები (Receiver) არის პასუხისმგებელი Command ობიექტში ინკაპსულირებული ოპერაციების შესრულებაზე;
- კონკრეტული იმპლემენტაცია (ConcreteCommand): CommandBase აბსტრაქტული კლასის (ან ინტერფეისის) კონკ-რეტული იმპლემენტაცია;
- გამომძახებელი (Invoker): ობიექტი, რომელიც წყვეტს თუ როდის უნდა გაეშვას Command ობიექტი.

Command არქიტექტურული სტანდარტი Silverlight ვებ-აპლიკაციებში გვეხმარება მოვლენების შეზღუდვების დაძლევაში, რადგან გრაფიკულ სამომხმარებლო გარემოში (View) აღიძვრება Command ობიექტი და ViewModel შრეზე ხდება მისი მიღება, რომელმაც იცის რა სამუშაოები უნდა ჩაატაროს საპასუხოდ [10,11,44,45].

გარდა ამისა, Command ობიექტის რამდენიმე წერტი-ლიდან გამოძახების შესაძლებლობა უზრუნველყოფს პროგრამული ლოგიკის ერთ ადგილას იმპლემენტირებას და სხვადასხვა ადგილებში მის გამოყენებას კოდის გამეორების გარეშე. მაგალითად, წარმოვიდგინოთ, რომ პროგრამულ დანართში გვაქვს Exit ბრძანება (Command ობიექტი), რომე-ლიც პასუხისმგებელია პროგრამის დახურვაზე. Exit ბრძანე-ბის პროგრამული კოდი იმპლემენტირებულია მხოლოდ ერთ ადგილას, მაგრამ სამომხმარებლო გარემოში მისი გამოძახება სხვადასხვა ადგილიდან შეგვიძლია: ეკრანის ზედა მარჯვენა

კუთხეში არსებულ X სიმბოლოზე დაჭერით, File მენიუდან Exit ოპერაციის არჩევით ან უბრალოდ, კლავიატურაზე Alt+F4 კომბინაციის აკრეფით.

ყველა ჩამოთვლილი ქმედება არის სხვადასხვა გამომძახებელი (Invoker) და უკავშირდება ერთსა და იმავე Command ობიექტს.

Command სტანდარტის იმპლემენტაცია MVVM არქიტექტურაში

➢ ICommandSource ინტერფეისი

ბრძანების გამომძახებელი ობიექტი (Invoker) არის ICommandSource ინტერფეისის წევრი. ქვემოთ მოცემულია IcommandSource ინტერფეისის პროგრამული კოდი:

```
public interface ICommandSource
{
   ICommand Command { get; }
   object CommandParameter { get; }
   IInputElement CommandTarget { get; }
}
```

ამ არქიტექტურით ნათლად ჩანს, რომ ყოველი კლასი, რომელიც იმპლემენტაციას უკეთებს ICommandSource ინტერ-ფეისს, უნდა შეიცავდეს მხოლოდ ერთ Command ობიექტს.

თავად Command ობიექტი ICommand ინტერფეისის იმპლემენტაციაა. Silverlight პროგრამულ უზრუნველყოფაში კონტროლები Command ობიექტს ამჟღავნებენ Dependency-Property სახით, რათა მონაცემთა ბმით შევძლოთ მისი მართვა. CommandParameter არის ობიექტი, რომელიც

ინდივიდუალური Command იმპლემენტაციისთვის სახასიათო ინფორმაციას შეიცავს და ნებისმიერი ტიპის მნიშვნელობის მიღება შეუძლია.

➢ ICommand ინტერფეისი

MVVM არქიტექტურაში ნებისმიერი ბრძანება არის ICommand ინტერფეისის იმპლემენტაცია. იგი აღწერს კონტრაქტს, რომელიც Command ობიექტის თითოეულმა იმპლემენტაციამ უნდა დააკმაყოფილოს. ICommand ინტერფეისის აღწერა მოყვანილია ქვემოთ:

```
public interface ICommand
{
    event EventHandler CanExecuteChanged;
    bool CanExecute(object parameter);
    void Execute(object parameter);
}
```

მეთოდი Execute არის Command არქიტექტურის მთავარი ნაწილი. იგი ინკაპსულირებას უკეთებს Command ობიექტის მიერ შესასრულებელ სამუშაოს. Execute მეთოდს ინდივიდუალური კონტექსტიდან ნებისმიერი ტიპის პარამეტრი შეგვიძლია გადავცეთ (რადგან C#-ში ყველა ობიექტი მემკვიდრეობით მოდის object ტიპიდან) [11].

CanExecute მეთოდი გამომძახებელს ატყობინებს დაშვებულია თუ არა მოცემულ მომენტში მითითებული ბრძანების შესრულება. პრაქტიკულად იგი გამოიყენება სამომხმარებლო ინტერფეისზე გამომძახებელი ობიექტის (მაგ., ღილაკი, ნახ.2.7, 2.8) ჩართვისა და ამორთვის მიზნით (Enable, Disable).

ნახ. 2.7. ღილაკი „დამატება" ჩამქრალია, რადგან ფორმაზე არ არის შეტანილი სავალდებულო ინფორმაცია

განვიხილოთ მაგალითი: დავუშვათ, რომ მომხმარებელმა არასრულად შეავსო ვებ-ფორმა და რომელიმე სავალდებულო ინფორმაციული ველი ცარიელი დატოვა. ფორმის შენახვა არ უნდა მოხდეს (მონაცემთა სათავსოში სავალდებულო ველის შეზღუდვის გამო), ანუ შენახვის ღილაკზე

დაჭერას არავითარი ეფექტი არ ექნება. თუ მომხმარებელს რაიმე ვიზუალურ მანიშნებელს არ მივაწვდით, იგი გაღიზიანდება შენახვის ღილაკზე ამაოდ დაჭერისას.

ამ დროს გამოსავალია შენახვის შესაბამისი Command ობიქტის *გაუქმება* (Disable), რაც სამომხმარებლო გარემოში გამოჩნდება როგორც ჩამქრალი ღილაკი.

თანამშრომლები/თანმხლები პირები
თანამშრომელი
თანმხლები პირი
პირადი ნომერი*
გვარი* კიკილაძე
სახელი ნინო
არარეზიდენტი
ქვეყანა* საფრანგეთის რესპუბლიკ
დასაწყისი* 10.04.2016
ქალაქი* პარიზი
დასასრული* 10.04.2016
შეტყობინების გააქტიურება
დღით ადრე 0
დამატება
დახურვა

ნახ. 2.8. ღილაკი „დამატება" ჩართულია, რადგან ფორმაზე შევსებულია ყველა სავალდებულო ინფორმაციული ველი

Command არქიტექტურაში მნიშვნელოვანი როლი აკისრია CanExecute მეთოდს. იგი აბრუნებს Boolean ტიპის მნიშვნელობას. იმისდა მიხედვით CanExecuteChanged მოვლენაზე საპასუხოდ რა მნიშვნელობას დააბრუნებს CanExecute

მეთოდი (True ან False), გადაწყდება ICommandSource ობიექტი ჩართული იყოს თუ ამორთული (Enabled, Disabled).

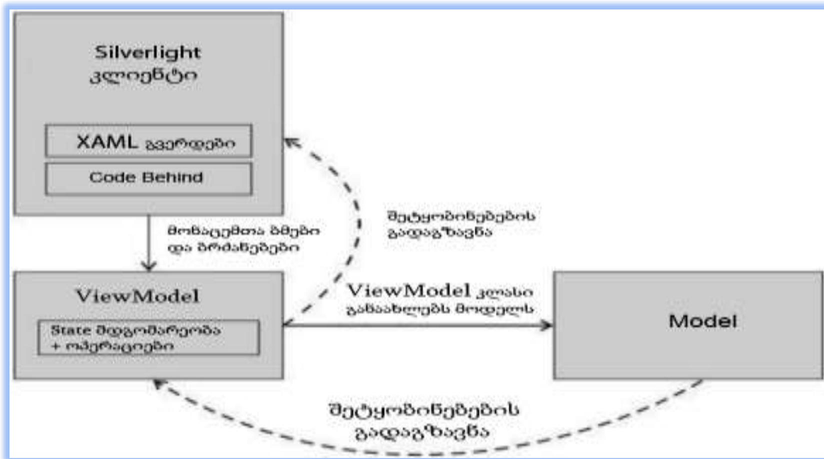
თავად CanExecute მეთოდის იმპლემენტაცია შეგვიძლია კონკრეტულ ViewModel კლასში გავიტანოთ. ქვემოთ მოყვანილია CommandBase კლასის პროგრამული კოდი:

```
public class CommandBase : ICommand
{
    public event EventHandler CanExecuteChanged;
    public Func<object, bool> CanExecuteCommnad;
    private Action<object> execute;
   public CommandBase()
    {
       execute = EX => { };
       CanExecuteCommnad = CE => true;
    }
   public CommandBase(Action<object> _Execute,
                     Func<object, bool> _CanExecute)
    {
       execute = _Execute;
       CanExecuteCommnad = _CanExecute;
    }
   public bool CanExecute(object parameter)
    {
       return CanExecuteCommnad == null
                     || CanExecuteCommnad(parameter);
    }
  public void Execute(object parameter)
```

```
    {
        Execute(parameter);
        if (CanExecuteChanged != null)
            CanExecuteChanged(this, new EventArgs());
    }
  protected virtual void OnCanExecuteChanged(EventArgs e)
    {
        var canExecuteChanged = CanExecuteChanged;
        if (canExecuteChanged != null)
            canExecuteChanged(this, e);
    }
  public void RaiseCanExecuteChanged()
    {
        OnCanExecuteChanged(EventArgs.Empty);
    }
}
```

2.11. პრეზენტაციის დონე, კომუნიკაცია View და ViewModel დონეებს შორის

პრეზენტაციის დონე აღწერს სამომხმარებლო ინტერფეისს. ყოველი გვერდი code-behind კლასი შეიცავს პროგრამული კოდის მინიმალურ რაოდენობას და ისიც, მხოლოდ იმ შემთხვევაში, თუ გრაფიკული ინტერფეისის მართვის გარკვეული როლი აქვს დაკისრებული. ძირითადი პროგრამული ლოგიკა იმპლემენტირებულია ViewModel კლასში, რომელიც ინახვას აპლიკაციის მდგომარეობას (State) და პრეზენტაციის დონეზე აგზავნის შეტყობინებებს. ეს დონეები ერთმანეთთან შემდეგი სქემით უქრთიერთქმედებს (ნახ. 2.9).



ნახ. 2.9. Silverlight კლიენტის, ViewModel და Model დონეებს შორის ურთიერთქმედება

View გვერდის კონტროლები მონაცემთა ბმით დაკავშირებულია ViewModel კლასის წევრებთან (Properties), როგერც წესი – დეკლარაციული გზით. მაგალითად, *ქალაქის* შემთხვევაში TextBox-ზე CityName მნიშვნელობა გამოგვაქვს შემდეგი XAML მარკირებით:

```
<TextBox
    Name="ctbCityName"
    Text="{Binding CityName, Mode=TwoWay}"/>
```

// ViewModel კლასში შენახულია CityName მნიშვნელობა.

ViewModel კლასში აღწერილია ბრძანებების სიმრავლე, როგორიცაა ცვლილებების შენახვა მონაცემთა სათავსოში. პრეზენტაციის დონე მათ კონკრეტულ კონტროლებთან აკავშირებს, რათა მომხმარებელს ქმედების განხორციელების საშუალება მიეცეს.

მაგალითად, მომხმარებელს შეუძლია Save ღილაკზე დაჭერა, რომელიც ViewModel კლასის SaveDataCommand ბრძანებაზე არის მიბმული [10].

ამას გარდა, ViewModel დონე ინფორმაციის მისაღებად მოთხოვნებს უგზავნის Model დონეს. მოდელის დონე იწყებს გამოთვლით პროცესებს და როდესაც დაასრულებს სამუშაოებს, ViewModel ობიექტს გადაუგზავნის შეტყობინებებს (ამ შემთხვევაში იგულისხმება, რომ მოდელი კლიენტის მხარეს იმყოფება და სერვერულ მხარეს მიმართავს ოპერაციების შესასრულებლად – ინფორმაციის წამოღება ან ცვლილება).

ViewModel ობიექტი, შედეგების მიღების შემდეგ, გადააგზავნის შეტყობინებებს View გვერდის მიმართულებით და მონაცემთა ბმის მექანიზმით განაახლებს სამომხმარებლო ინტერფეისს.

2.12. ანიმაციები Silverlight ტექნოლოგიაში

ვებ-აპლიკაციებში ანიმაცების დამატებით სამომხმარებლო ინტერფეისი მნიშვნელოვნად უმჯობესდება.

წარსულში ამ ტიპის ანიმაციის მიღწევა უმეტესწილად Adobe Flash ტექნოლოგიის გამოყენებით ხდებოდა. კარგი სიახლე მდგომარეობს იმაში, რომ Microsoft.NET პლატფორმაზე შესაძლებელი გახდა ანიმაციების გაკეთება – ეს Silveright ტექნოლოგიის ერთ-ერთი მნიშვნელოვანი ასპექტია. დეველოპერს აღარ ჭირდება Adobe Flash ტექნოლოგიის ცოდნა, ანიმაციების შესამუშავებლად [9,10,44,45].

ტერმინი *ანიმაცია* ასოცირდება ანიმირებულ სურათებთან (ძველი დისნეის მულტფილმები). მხატვარი ქმნის მცირე

განსხვავების მქონე ზევრ სურათს და როდესაც მათ სწრაფი თანმიმდევრობით აჩვენებს ერთმანეთის მონაცვლეობით, იქმნება თანაბარი მოძრაობის ეფექტი.

Silverlight ტექნოლოგიაშიც ანალოგიურად არის ანიმაცია იმპლემენტირებული: დროის მანძილზე ხდება ობიექტის რომელიმე ატრიბუტის მდგომარეობის მცირედი რიცხვითი მნიშვნელობით წანაცვლება, რის შედეგადაც მომხმარებელი ხედავს ერთი წერტილიდან მეორეში თანაბ-რად გადაადგილებად ობიექტს.

მაგალითად, შეიძლება სურათის იკონი გავზარდოთ, როდესაც მომხმარებელი მაუსის კურსორს მიიტანს მასთან, ხოლო როდესაც კურსორი გასცდება იკონს, იგი პირვანდელ ზომას დაუბრუნდება (ნახ.2.10).

ნახ.2.10. Silverlight ტექნოლოგიით შექმნილი ანიმირებული Application -პანელი

ზემოთ მოყვანილ მაგალითში ანიმაცია გაკეთებულია შემდეგი ლოგიკით: როდესაც მაუსის კურსორი ხვდება სურათის icon-ის არეში დროითი შკალის მომენტში Timestamp=0.00 სურათის ატრიბუტების საწყისი მნიშვნელობებია:

Width = 50 pixel, Height = 50 pixel

ხოლო Timestamp=0.25 მომენტში ეს მახასიათებლები იცვლება შემდეგი მნიშვნელობებით:

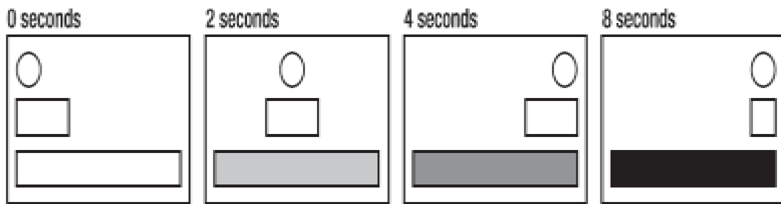
Width = 75 pixel, Height = 75 pixel

ეს გარდასახვა Timestamp = 0.00 -დან Timestamp = 0.25-ში ხდება საკმაოდ მდორედ, Silverlight ტექნოლოგიაში გათვალისწინებულია მდორე, თანაბარი გადადინება თავდაპირველ და საბოლოო მნიშვნელობებს შორის, რის შედეგადაც იქმნება უწყვეტი მოძრაობის ილუზია.

- *Storyboard პანელი*

ფილმებსა და ანიმაციებში ტერმინი Storyboard აღნიშნავს სურათების თანმიმდევრობას, რომელიც ასახავს მოქმედებას, მდგომარეობის ცვლილებას ფილმის მსვლელობის მანძილზე, ანუ Storyboard არის დროის შკალაზე (Timeline) გადატანილი სურათების ტრანსფორმაცია ერთი მდგომარეობიდან მეორეში.

მაგალითად, 2.11 ნახაზზე მოყვანილ Storyboard შკალაზე ნაჩვენებია სამი ობიექტის (წრე, პატარა და დიდი მართკუთხედი) ტრანსფორმაცია [9]:



ნახ. 2.11. Storyboard პანელზე ანიმაციის მაგალითი

საწყისი მდგომარეობის მომენტში სამივე ობიექტი მარცხენა კიდის მიმართ არის გასწორებული. 2 წამის გასვლის შემდეგ წრე და პატარა მართკუთხედი მოძრაობას იწყებს დოკუმენტის მარჯვენა კიდის მიმართულებით, ხოლო დიდი მართკუთხედის ფერი გადადის თეთრიდან შავში. 4 წამის გასვლის შემდეგ Timeline შკალაზე წრე და პატარა მართკუთხედი უკვე მარჯვენა კიდის მიმართაა გასწორე-ბული, და ამ მომენტიდან დაწყებული პატარა მართკუთხედი იწყებს გარდასახვას კვადრატად. 8 წამის გასვლის მომენტში პატარა მართკუთხედი კვადრატად არის გარდასახული, ხოლო დიდი მართკუთხედი ბოლომდე გაშავებულია.

თუ მოყვანილ storyboard-ს გადავიყვანთ Silverlight ანიმაციების ენაზე, გვექნება 4 ანიმაცია:

- ორი ანიმაცია, როდესაც წრე და პატარა კვადრატი მარცხნიდან მარჯვნივ გადაადგილდება.
- ერთი ანიმაცია, როდესაც დიდი მართკუთხედის ფერი იცვლება თეთრიდან შავზე.
- ერთი ანიმაცია, როდესაც პატარა მართკუთხედი გარდაიქმნება კვადრატად.

➢ *ანიმაციის ტიპები Silverlight ტექნოლოგიაში*

ანიმაციის ყველა ტიპი Silverlight ტექნოლოგიაში მემკვიდრეობით მოდის Timeline კლასიდან, რომელიც განთავსებულია System.Windows.Media.Animation ბიბლიოთეკაში. Silverlight ტექნოლოგიაში დაშვებულია შემდეგი ტიპის ანიმაციები:

- ColorAnimation
- ColorAnimationUsingKeyFrames
- DoubleAnimation
- DoubleAnimationUsingKeyFrames
- ObjectAnimationUsingKeyFrames
- PointAnimation
- PointAnimationUsingKeyFrames

თითოეული ანიმაცია სხვადასხვა ტიპის ობიექტს წარმოადგენს. მაგალითად, ColorAnimation ტიპის ანიმაცია Color ატრიბუტის ანიმაციას ახდენს (ფერის შეცვლა საწყისი ფერიდან და საბოლოო ფერამდე), DoubleAnimation ანიმირებას უკეთებს Double ატრიბუტს, ხოლო ObjectAnimation ანიმირებას უკეთებს Object ატრიბუტს და ა.შ. პროგრამისტი განსაზღვრავს რომელი ანიმაციის ტიპი უნდა გამოიყენოს იმისდა მიხედვით, თუ რისი ანიმირება არის საჭირო.

მაგალითისთვის განვიხილოთ უმარტივესი ანიმაცია, სადაც გარკვეული დროის განმავლობაში მართკუთხედის ზომა უნდა გაიზარდოს (ნახ.2.12).

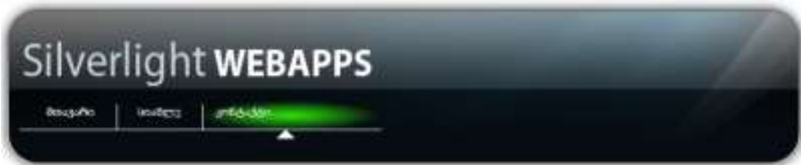
ნახ. 2.12. მართკუთხედის ზომის გაზრდის ანიმაცია

ამ ეფექტის მისაღწევად გამოიყენება DoubleAnimationUsingKeyFrames ტიპის ანიმაცია (რადგან ვცვლით Width და Height ატრიბუტების მნიშვნელობებს და ორივე მათგანის მნიშვნელობა არის double ტიპის). XAML ენაზე ამ ანიმაციის მისაღებად გვექნება შემდეგი კოდი:

```
<UserControl.Resources>
    <Storyboard x:Name="Storyboard1">
        <DoubleAnimationUsingKeyFrames
            BeginTime="00:00:00"
            Storyboard.TargetName="rectangle"
            Storyboard.TargetProperty="Width">
            <SplineDoubleKeyFrame KeyTime="00:00:02" Value="400"/>
        </DoubleAnimationUsingKeyFrames>
        <DoubleAnimationUsingKeyFrames
            BeginTime="00:00:00"
            Storyboard.TargetName="rectangle"
            Storyboard.TargetProperty="Height">
            <SplineDoubleKeyFrame KeyTime="00:00:02" Value="400"/>
        </DoubleAnimationUsingKeyFrames>
    </Storyboard>
</UserControl.Resources>

<Grid x:Name="LayoutRoot" Background="White">
    <Rectangle
        Height="120"
        Width="200"
        HorizontalAlignment="Left"
        VerticalAlignment="Top"
        Stroke="#FF000000"
        x:Name="rectangle"/>
</Grid>
```

ნახ. 2.13. XAML კოდი ანიმაციის მისაღებად

Silverlight ტექნოლოგიის ანიმაციის ეფექტური გამოყენების ნიმუშს წარმოადგენს საიტის ანიმირებული სანავიგაციო პანელი: როდესაც მომხმარებელი მაუსის კურსორს მიიტანს მენიუს შესაბამის ღილაკზე, ტექსტის ფონად ჩნდება მკვეთრი მწვანე გრადიენტი და მის ქვემოთ პატარა სამკუთხა სიმბოლო გადაადგილდება ჰორიზონტა-ლურ ღერძზე (ნახ.2.14).

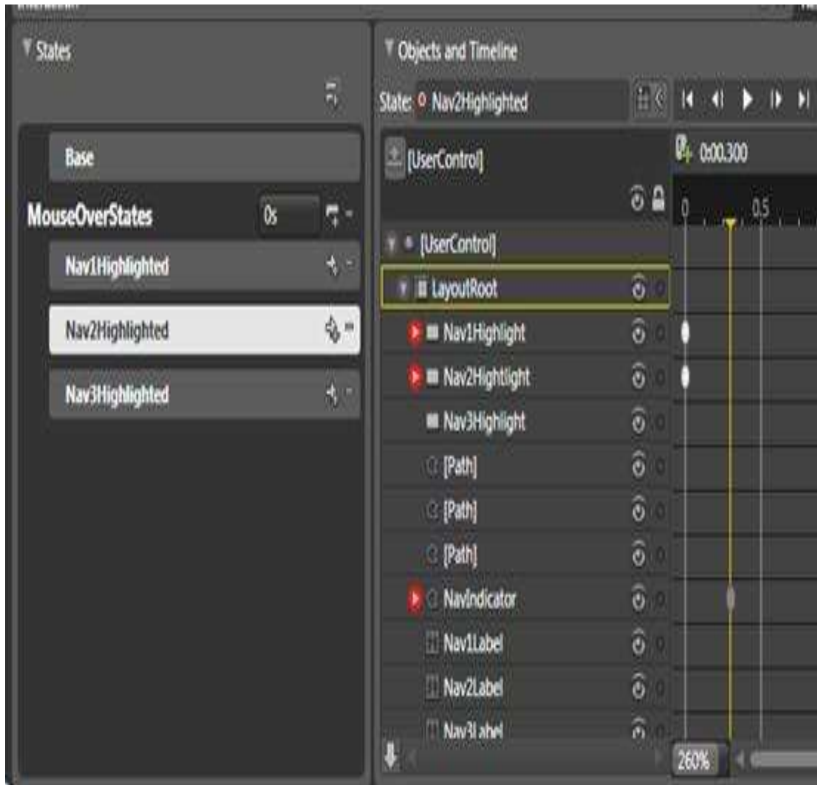


ნახ. 2.14. ვებ აპლიკაციის სანავიგაციო პანელი ანიმაციის ეფექტით

ამ ეფექტის მისაღებად გამოყენებულია Expression Blend ხელსაწყო. პირველ ეტაპზე მოხდა მენიუს პანელის დახატვა (გამოყენებულია სახატავი ინსტრუმენტები: line, box, pen path, gradient). შემდეგ ნახატზე განისაზღვრა რეგიონები, რომელიც წარმოადგენს მოხმარებელის სამოქმედო არეს (ანუ დადგინ-და არეები, რომელზეც პროგრამა ვალდებულია აღიქვას მაუსის კურსორის მიტანის ან დაკლიკვის ქმედებები).

ნახ. 2.15. სანავიგაციო პანელზე ანიმაციის ეფექტის დადება Blend ხელსაწყოში

შემდეგ ეტაპზე მოხდა მომხმარებლის ქმედებების დაკავშირება MouseOverStates მდგომარეობებთან და ანიმირების ეფექტების დადება VisualStateManager მდგომარეობის ცვლილებებით (ნახ.2.16).



ნახ. 2.16. ანიმაციის მართვა Blend ხელსაწყოში

2.13. Web - აპლიკაციაში რეპორტების ინტეგრაციის მეთოდები

თანამედროვე, მაღალკონკურენტულ ბიზნეს გარემოში ინფორმაციის ფლობას გადამწყვეტი მნიშვნელობა ენიჭება. ტექნოლოგიის განვითარებასთან ერთად ინფორმაციის შეგროვება გაიოლდა, თუმცა მისი გაანალიზება კვლავ რჩება რთულ და ფაქიზ თემად.

ბიზნეს ანალიზის და ანგარიშგებათა (რეპორტების) შემუშავებისთვის გადამწყვეტი ფაქტორია კარგი, მოქნილი სამუშაო ინსტრუმენტების (Business Intelligence Tools) არსებობა [47].

SQL Server Reporting Services (SSRS) არის კომპანია Microsoft-ის ინსტრუმენტი, რომელიც შემუშავებულია მონაცემთა ანალიზისა და ანგარიშგებების გენერაციის მიზნით. იგი ბიზნეს ანალიზის პლატფორმის (BI) ერთ-ერთი კომპონენტია. მთლიანობაში, კომპონენტები იძლევა მონაცემთა ანალიზის მოქნილ საშუალებებს. ეს კომპონენტებია [40, 48-51]:

- *SQL Server:* ტრადიციული მონაცემთა ბაზის მანქანა, რომელზეც ასევე ინახება SSRS რეპორტების კატალოგი;
- *SQL Server Analysis Services (SSAS):* კომპონენტი ასრულებს ისეთ ანალიზურ პროცესებს, როგორიცაა მონაცემთა აგრეგაცია და წარმოდგენა სხვადასხვა ჭრილში (მაგალითად, გეოგრაფიული მდებარეობა, დრო);

- *SQL Server Integration Services (SSIS):* კომპონენტი გამოიყენება მონაცემთა ამოსაღებად, ტრანსფორმირების და ჩატვირთვის მიზნით (Extract, Transform, Load - ETL);
- *SQL Server Reporting Services (SSRS):* სერვერზე დაფუძნებული, განვრცობადი პლატფორმაა, რომელშიც ხდება ინფორმაციის დამუშავება, ფორმატირება და მომხმარებლისთვის ტრადიციული თუ ინტერაქტიული ფორმატით მიწოდება. SSRS კონფიგურირებადია და ხასიათდება მრავალი ფუნქციით, როგორიცაა მაგალითად, მონაცემთა ვიზუალიზაცია დიაგრამებისა და გრაფიკების სახით, ანგარიშგებათა სხვადასხვა ფორმატით ექსპორტირება (HTML, Excel, PDF, მომხმარებლის ელ-ფოსტაზე გადაგზავნა, დაკონფიგურირება და SharePoint კორპორატიულ პორტალებში ჩაშენება [46].

➢ *ანგარიშგების მოდულის ინტეგრაცია Web-აპლიკაციაში*

Microsoft-ის კომპანიამ რეპორტინგის სერვისის (Reporting Services) დაპროექტებისას თავიდანვე გაითვალისწინა მისი განვრცობადობის აუცილებლობა და სერვისის ღია ინტერფეისით წვდომადი გახადა ვებ-აპლიკაციებისა და დაპროგრამების გარემოთა ფართო სპექტრისთვის.

განვიხილოთ პროგრამულ უზრუნველყოფაში ანგარიშგებათა ჩაშენების სამი შესაძლო ვარიანტი [48]:

- Reporting Server Web Service (ასევე ცნობილი როგორც Reporting Services SOAP API);
- ReportViewer კონტროლი;
- წვდომა URL გზავნილის საშუალებით.

➢ *Reporting Server ვებ-სერვისი*

Report Server Web Service რეპორტინგის ფუნქციასთან სამუშაო ბაზისური ინტერფეისია. სერვისი უზრუნველყოფს პროგრამისტს აპლიკაციაში რეპორტინგის ფუნქციის ჩაშენების ყველა საჭირო მეთოდითა და კონტროლით. მაგალითად Report Manager არის აპლიკაცია, რომელიც მოყვება რეპორტინგ სერვისებს და იყენებს Web-სერვისს რეპორტ სერვერის მონაცემთა ბაზის სამართავად.

რადგან SOAP API ვებ-სერვისია, იგი .NET Framework-ში წვდომადია, სხვა SOAP სერვისების მსგავსად. Proxy-კლასების გენერაციით შესაძლებელია Report Server ვებ-სერვისის მეთოდებზე წვდომა და მათი გამოყენება [49,50].

➢ *ReportViewer კონტროლი (Visual Studio)*

Report Viewer კონტროლი მოყვება Visual Studio ინსტრუმენტს და აპლიკაციიდან რეპორტის გაშვების დანიშნულებით ატარებს (ნახ.2.17). არსებობს კონტროლის ორი რეალიზაცია: ერთი – რომელიც მუშაობს Windows Forms აპლიკაციებთან, ხოლო მეორე – Web Forms აპლიკაციებთან.

ნახ. 2.17. ReportViewer კონტროლი ვებ-აპლიკაციაში

თითოეული კონტროლი უზრუნველყოფილია Report Server-ზე ატვირთული რეპორტების გაშვების და სხვადასხვა ფორმატში ექსპორტირების ფუნქციონალით (მოსახერხებელია მომხმარებლისთვის, რომლის კომპიუტერზე არაა დაინსტალირებული რეპორტ სერვერი) [51].

ReportViewer კონტროლს ახასიათებს ორი რეჟიმი:

- Remote Processing Mode – Report Server სერვერზე ატვირთული რეპორტების ჩვენების რეჟიმი. რეპორტის დამუშავება ხდება სერვერზე. მუშაობის პროცესში იგი იძლევა რამდენიმე სერვერის დაიტვირთვის ან შესასრულებელი გამოთვლითი სამუშაოების რამდენიმე პროცესორზე გადანაწილების შესაძლებლობას;
- Local Processing Mode – რეპორტების გაშვების ალტერნატიული მეთოდი, როდესაც Reporting Servises სერვისი არ არის დაყენებული სამუშაო კომპიუტერზე. Remote Processing Mode რეჟიმისგან განსხვავებით, კონტროლში მხოლოდ რეპორტ სერვერის ფუნქციის გარკვეული ქვესიმრავლეა ხელმისაწვდომი. ამ რეჟიმში რეპორტის მონაცემთა სიმრავლის დამუშავება (Data Processing) ReportViewer კონტოლით აღარ ხდება, არამედ თვითონ ვებ-აპლიკაციაშია რეალიზებული, თუმცა კი რეპორტის დამუშავებას (Report Processing) კვლავ კონტროლი ასრულებს.

- *URL მისამართი*

URL მისამართით რეპორტებზე წვდომა არის ვებ-აპლიკაციებიდან რეპორტების დათვალიერების კიდევ ერთი მეთოდი და გამოიყენება როდესაც ReportViewer კონტროლი არ არის ხელმისაწვდომი. URL მისამართით წვდომა მოსახერხებელია მომხმარებლისთვის ელ-ფოსტაზე რეპორტების გზავნილების გადასაცემად (ნახ.2.18).

```
<a href="http://server/reportserver?/EmployeeReports/EmployeeList
Drilldown&rs:Command=Render&rc:LinkTarget=main" target="main">
    თანამშრომლების შესახებ ინფორმაციის ნახვა
</a>
```

ნახ. 2.18. Web -აპლიკაციიდან რეპორტზე URL გზავნილით მიმართვა

2.14. სერვერული და ლოკალურად ჩაშენებული რეპორტების შედარება

რეპორტის ვიზუალური მხარის აწყობა შესაძლებელია Report Server პროექტში, რომელიც მოყვება SQL Server თანამედროვე ვერსიებს და ასევე ინტეგრირებულია Visual Studio პროგრამაში. დიზაინერი (Report Designer) რეპორტის ასაწყობი გრაფიკული გარემოა [46].

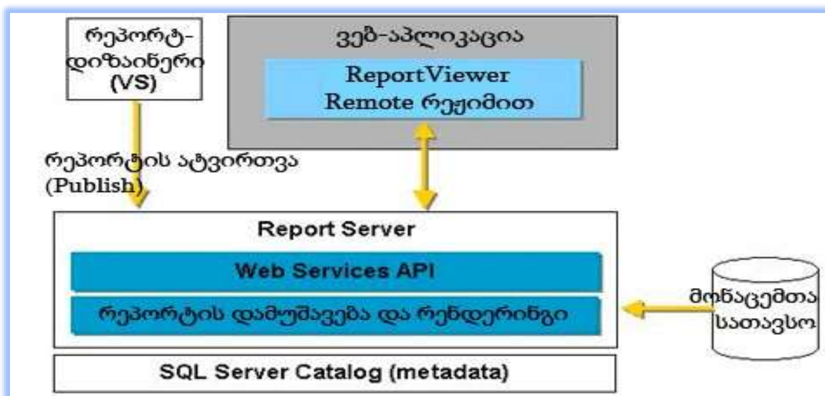
დიზაინის აწყობის შემდეგ მომხმარებლისთვის ხელმისაწვდომი ხდება რეპორტის ატვირთვა Report Server სერვერზე ან ვებ-აპლიკაციაში მისი ლოკალურად ჩაშენების შესაძლებლობა.

➢ *სერვერული მხარის რეპორტები*

Report Server – სერვერზე ატვირთული რეპორტი არის სტანდარტული (.rdl) გაფართოების ფაილი, რომელიც შეიცავს ინფორმაციას მონაცემთა ბაზასთან დაკავშირებისა და მონაცემთა ამოღების შესახებ.

სერვერულ რეჟიმში ინტეგრირებული რეპორტების შემთხვევაში ვებ-აპლიკაცია უკავშირდება Report Server-ზე ატვირთულ ანგარიშგებას, ხოლო რეპორტის მონაცემებით შევსებას, დამუშავებას და დარენდერებას ასრულებს სერვერი (ნახ.2.19) [48]. Report-სერვერი უზრუნველყოფილია სხვადა-სხვა სერვისით, როგორიცაა უსაფრთხოება, ისტორიის შენახვა, ანგარიშგებების ელ-ფოსტაზე გამოწერა და ა.შ.

როდესაც ვებ-აპლიკაციაში რეპორტი ინტეგრირებულია სერვერულ რეჟიმში, მაშინ SQL Server-ის Report Server ლიცენზია სავალდებულოა.



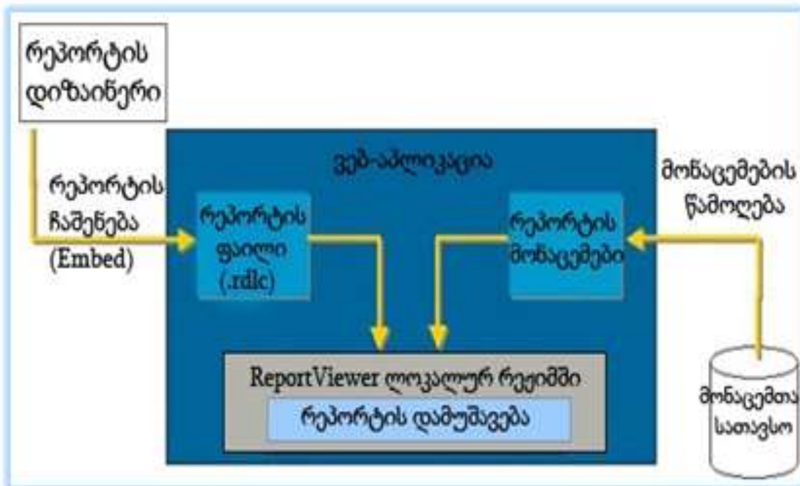
ნახ. 2.19. სერვერული მხარის რეპორტის დამუშავების პროცესი

➢ *ლოკალურად ჩაშენებული რეპორტები*

ვებ-აპლიკაციაში ლოკალურად ჩაშენებული რეპორტი .rdlc გაფართოების ფაილია, რომელიც მოიცავს DataSource ობიექტების შესახებ მეტა-მონაცემებს, მაგრამ არ შეიცავს SQL-სერვერთან კავშირის ან Query პროგრამული კოდის შესახებ ინფორმაციას [47-51]. ჩაშენებული რეპორტების ფუნქციის სარეალიზაციოდ Visual_Studio პროგრამას მოყვება Report-Viewer კონტროლი. მისთვის სავალდებულოა სამუშაო მანქანაზე იყოს .NET Framework 2.0 ან უფრო ახალი ვერსია.

ლოკალურ რეჟიმში ვებ-აპლიკაციის მხარეს ხდება რეპორტის მონაცემებით შევსება. სერვერული რეპორტე-ბისგან განსხვავებით SQL-ლიცენზია არ არის სავალდებულო.

რეპორტის დასამუშავებლად საჭირო ფუნქცია უზრუნ-ველყოფილია ReportViewer კონტროლში (ნახ. 2.20).



ნახ. 2.20. ლოკალური რეპორტის დამუშავების პროცესი

ლოკალურ რეჟიმში რეპორტის სამომხმარებლო ინტერ¬ფეისის უზრუნველყოფა და პარამეტრების გადაწოდება ევალება აპლიკაციის მხარეს. რეპორტის მონაცემთა წყარო შეიძლება იყოს ADO.NET-ის DataTable ობიექტები [52].

ლოკალურად ჩაშენებული რეპორტების შემთხვევაში უსაფრთხოების ფუნქციის უზრუნველყოფა Web-აპლიკა¬ციის მხარეს გადაინაცვლებს. რეპორტში ჩაშენებული კოდი არ არის უფლებამოსილი ფაილური სისტემის წვდომასა თუ ქსელურ გარემოსთან მუშაობაზე (თუ ცხადად არ აქვს მინიჭებული უფლებები - explicit permissions).

- *დასკვნა: რეჟიმის გამოყენების შესახებ*

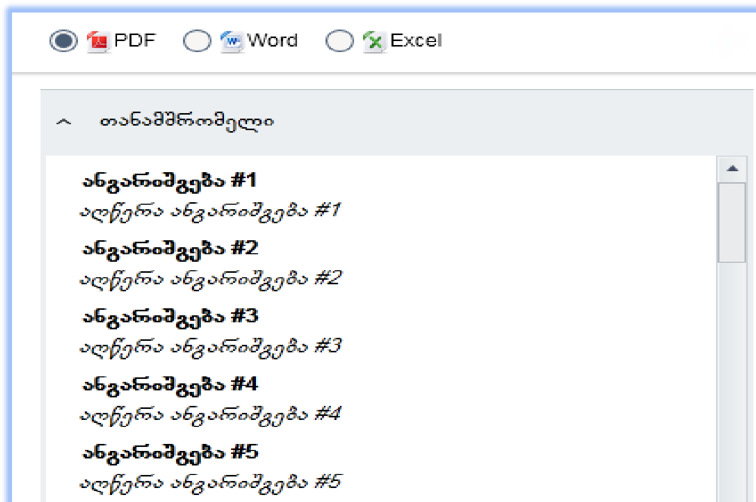
ლოკალური რჟიმი სერვერულ რეჟიმთან შედარებით ნაკლებად მოქნილია და განკუთვნილია მცირე ან საშუალო ზომის რეპორტების გენერაციისთვის (რადგან გამოთვლითი პროცესები და რეპორტის გენერაცია კლიენტის მანქანაზე სრულდება), stand-alone ტიპის აპლიკაციებში, რომლებიც არ საჭიროებს Report Server-ს.

სერვერული რეპორტების გამოყენება რეკომენდებუ-ლია მრავალი მომხმარებლის ერთდროული მუშაობის რეჟიმის შემთხვევაში, კომპლექსური Query სკრიპტებისგან შემდგარი დიდი მოცულობის მონაცემების ანგარიშგებათა გენერაციის დროს [47].

2.15. ანგარიშგებების გენერაცია მომხმარებლისთვის ადვილად გამოყენებადი ინტერფეისიდან

განვიხილოთ მაგალითი, სადაც ServerReport კლასის და ReportViewer კონტროლის გამოყენებით ხდება კავშირის დამყარება Reporting Services სერვერთან და Render() მეთოდის გამოყენებით ექსპორტირება სხვადასხვა ფორმატში.

ეს მეთოდი მოსახერხებელია, როდესაც არ გვსურს მომხმარებლის ინტერფეისში ReportViewer კონტროლის გამოტანა, მაგრამ გვჭირდება რეპორტის გენერაცია და ექსპორტირება სხვადასხვა ფორმატით. 2.21 ნახაზზე ნაჩვენებია მომხმარებლის ინტერფეისზე გამოტანილი რეპორტების სია.



ნახ. 2.21. ანგარიშგებები: სამომხმარებლო ინტერფეისის მხარე

ანგარიშგების დასახელება არის გზავნილი, რომლის არჩევითაც ხდება ანგარიშგების გენერაციის მეთოდის გამოძახება, ამორჩეულ ფორმატში (მაგალითად, PDF) გადაყვანა და დაბრუნებული ბაიტების მასივის ჩაწერა დროებით ფაილში. შემდეგ, მომხმარებელს შეუძლია ფაილი ჩამოტვირთოს საკუთარ კომპიუტერზე ან გახსნას ეკრანზე.

➢ *პრერეკვიზიტები:*

1) Microsoft Report Viewer 2012 (ან უფრო ახალი) Runtime პაკეტის გადმოწერა და ინსტალაცია. პაკეტის საინსტალაციო ფაილის დასახელება არის ReportViewer.msi და .NET Framework-ზე დაწერილი აპლიკაციებისთვის შეიცავს Microsoft Reporting ტექნოლოგიით შემუშავებულ ანგარიშგებათა გაშვების ფუნქციას;

2) Web-აპლიკაციის პროექტში გზავნილების (References) Folder-ში Microsoft.ReportViewer.WinForms.dll-ის დამატება.

შემდეგ ბიჯზე ვქმნით SaveReportToTempFile() მეთოდს, რომელიც აგენერირებს რეპორტს და აბრუნებს მას ბაიტების მასივის (byte[]) სახით.

მეთოდის პარამეტრებია:

- ანგარიშგების მისამართი Report Server-ზე (reportPath);
- ფორმატის დასახელება, რომელშიც უნდა მოხდეს ანგარიშგების ექსპორტირება (format);
- ანგარიშგების პარამეტრები (param1, param2, param3 და ა.შ.)

ანგარიშგების გენერაციისთვის სავალდებულოა Reporting Services Credentials-ის მანდატის განსაზღვრა. მომხმარებლის დასახელება, პაროლი და დომენის დასახელება დაკონფიგურირებულია web.config ფაილში, ისევე როგორც Reporting Server სერვერზე ატვირთული რეპორტების საქაღალდის მისამართი (ნახ. 2.22).

```
<add key="ReportingServerAddress"
     value="http://localhost/ReportServer"/>
<add key="ReportServiceUserName" value="ReportUser"/>
<add key="ReportServicePassword" value="123"/>
<add key="ReportServiceDomain" value="domain"/>
```

ნახ. 2.22. Web.config ფაილში Reporting Services Credentials კონფიგურაცია

2.23 ნახაზზე ნაჩვენებ ლისტინგში ასახულია SaveReportToTempFile() მეთოდში ServerReport ობიექტის ინიციალიზაციის მაგალითი Web.config კონფიგურაციის პარამეტრების გამოყენებით.

შემდეგ ბიჯზე ServerReport ობიექტს გადაეცემა ანგარიშგების პარამეტრები და Render() მეთოდის გამოძახებით მოხდება ექსპორტირების ფორმატში გადაყვანა (ნახ. 2.24).

```
var TmpServerReport = new ServerReport
{
    ReportServerCredentials = new ReportCredentials
        (
            ConfigurationManager.AppSettings["ReportServiceUserName"],
            ConfigurationManager.AppSettings["ReportServicePassword"],
            ConfigurationManager.AppSettings["ReportServiceDomain"]
        )
        ,
    ReportServerUrl = new Uri(
        ConfigurationManager.AppSettings["ReportingServerAddress"])


        ,
    ReportPath =
        ConfigurationManager.AppSettings["ReportingServerFolder"]
        + ReportingListLine_ReportPath
};
```

ნახ. 2.23. SaveReportToTempFile() მეთოდში
პარამეტრების გამოყენება

```
TmpServerReport.SetParameters(
    new ReportParameter(
        "StringParameter1", param1));

TmpServerReport.SetParameters(
    new ReportParameter(
        "DateParameter2", param2.ToString("yyyyMMdd")));

TmpServerReport.SetParameters(
    new ReportParameter(
        "DateParameter3", param3.ToString("yyyyMMdd")));

File.WriteAllBytes(
    TempPath, TmpServerReport.Render(ExportFormat));

DocumentStorageBO_SLData DS =
    new DocumentStorageBO_SLData()
    {
        DocumentStorage_FileName = TmpFileName,
        DocumentStorage_Extention = TmpExtension
    };

return DS;
```

ნახ. 2.24. ანგარიშგებისთვის პარამეტრების გადაწოდება და მითითებული ფორმატით ექსპორტირება bytes[] მასივში

დაბრუნებული bytes[] მასივი File.WriteAllBytes() მეთოდით იწერება დროებით ფაილში და შემდეგ, მომხმარებლის სურვილისამებრ, შესაძლებელია ამ ფაილის გადმოწერა ან ეკრანზე გახსნა.

2.16. მეორე თავის დასკვნა

RIA-არქიტექტურა წარმოადგენს ინტერაქტიული გრაფიკული სამომხმარებლო ინტერფეისის მქონე Web-აპლიკაციების შესამუშავებლად საჭირო პრინციპების ნაკრებს. იგი შემუშავებულია ვებ-საიტებისა და დესკტოპ-აპლიკაციების საუკეთესო პრაქტიკების საფუძველზე. RIA-აპლიკაციებში პროცესები სრულდება ასინქრონულად და მომხმარებელი არ ელოდება პასუხს ვებთან მუშაობისას.

გარდა ამისა, ბიზნეს-ლოგიკის ნაწილი სრულდება კლიენტის მხარეს, რაც ამსუბუქებს სერვერზე დატვირთვას და ზრდის პროგრამული დანართის შესრულების სისწრაფეს. RIA არქიტექტურის გამოყენება კარგ შედეგს იძლევა ცხრ.2.1 ზე ჩამოთვლილ არქიტექტურულ სტანდარტებთან ერთობ-ლიობაში. Command არქიტექტურული სტანდარტის გამოყენებით მოთხოვნებზე საპასუხო ლოგიკის კაფსული-რება ხდება ერთ, დამოუკიდებელ ობიექტში.

SQL Server Reporting-სერვისები ვებ-აპლიკაციაში ანგარიშგებათა ინტეგრაციის მრავალფეროვანი არჩევანის გაკეთების საშუალებას იძლევა. მეთოდოლოგიის არჩევისას ვხელმძღვანელობთ დასმული ამოცანიდან გამომდინარე: ლოკალური რეპორტები კარგად მუშაობს მცირე ან საშუალო სიდიდის ინფორმაციის გადამუშავების შემთხვევაში, ხოლო რთული გამოთვლითი სამუშაოების და დიდი ზომის ინფორმაციის გამოტანის დროს რეკომენდებულია სერვერუ-ლი რეპორტების გამოყენება.

თავი 3
ადამიანური რესურსების მართვის სისტემა

3.1. არქიტექტურა და იმპლემენტაცია

სახელმწიფო სტრუქტურებსა და კერძო სექტორში ადამიანური რესურსების მართვის სისტემების (Human Resource Management Systems – HRMS) შექმნა, სრულყოფა და განვითარება ყოველთვის იყო (და არის) მნიშვნელოვანი და აქტუალური მოვლენა. მაღალკვალიფიციური კადრების მოზიდვა, შენარჩუნება და განვითარება მოქალაქეებისათვის მაღალი ხარისხის მომსახურების მიწოდების მნიშვნელოვანი წინაპირობაა. ეფექტიანი და ფუნქციონალური ადამიანური რესურსების მართვის სტრატეგიის შესამუშავებლად საჭიროა არსებული სისტემის მახასიათებლების დადგენა და იმ ხარვეზების იდენტიფიკაცია, რომლებიც დაუყოვნებლივ აღმოფხვრას საჭიროებს. ამ მიზნით საჯარო სამსახურის ბიურომ არა-ერთი კვლევა ჩაატარა [28,29,53-55].

– *კვლევის მიზანი:* საქართველოს საჯარო სამსახურის დაწესებულებათა ადამიანური რესურსების მართვის ძირითადი საბაზო სისტემების გამოვლენა, კლასიფიკაცია და დარგობრივი სიტუაციური რუკის წარმოდგენა;

– *კვლევის მეთოდი:* ანალიზისთვის საჭირო მონაცემების შეგროვება მოხდა სტრუქტურირებული კითხვარის საფუძველზე, რომელიც შედგებოდა ღია და დახურული კითხვებისგან;

– *საკვლევი ნიმუში:* საკვლევ ნიმუშს სამთავრობო უწყებათა ადამიანური რესურსების მართვის მენეჯერები

წარმოადგენენ. ამგვარი სტრატეგია იმ ფაქტით არის განპირობებული, რომ პირდაპირი ინფორმაციის წყარო სწორედ ამ სფეროს პროფესიონალები არიან. ამავდროულად, ისინი ადამიანური რესურსების მართვის სისტემების დანერგვა/გამოყენებაზე აგებენ პასუხს. კვლევაში ცხრამეტი (19) სამინისტროს, ასევე საქართველოს პარლამენტის, მთავრობის კანცელარიისა და პრეზიდენტის ადმინისტრაციის HR მენეჯერები მონაწილეობდნენ [55].

– *მიგნებები:* კვლევის ზოგადი მიგნებები დაგვეხმარა საკვანძო ინფორმაციის დამუშავებაში და ადამიანური რესურსების მართვის სისტემების მიზნის, როლის და საჭიროების ანალიზში. ქვემოთ წარმოდგენილ პარაგრაფებში თავმოყრილია ადამიანური რესურსების მართვის სისტემის ფუნქციონალური მოთხოვნილებებისა და პროგრამული უზრუნველყოფის არქიტექტურის შესახებ ინფორმაცია.

3.2. კადრების მართვის ერთიანი ელექტრონული სისტემა საჯარო სტრუქტურებში

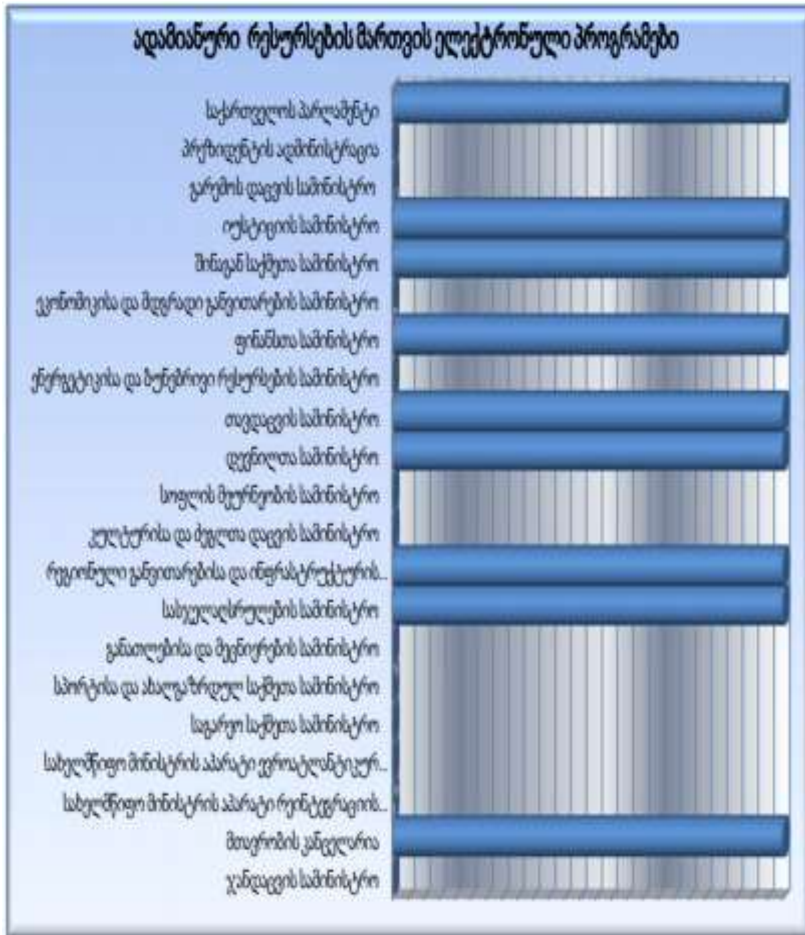
ადამიანური რესურსების ოპერაციული პროცესების გაუმჯობესების მიზნით კომპიუტერული ტექნოლოგიების გამოყენება კადრების მართვის სფეროში მსოფლიოში ფართოდ აპრობირებული პრაქტიკაა. ადამიანური რესურსების მართვის ელექტრონული სისტემების დანერგვა მნიშვნელოვანია, რათა მოხდეს ზოგადი მენეჯმენტის ხარისხის სრულყოფა საქართველოს საჯარო სამსახურის დაწესებულებებში. ხელისუფლების მიზანია ადამიანური რესურსების

მართვის პროგრამული უზრუნველყოფის პროექტების დანერგვა, რათა გამლიერდეს ადმინისტრაციული ეფექტიანობა, შემცირდეს ხარჯები და გაუმჯობესდეს მომსახურეობის სტანდარტები. ადამიანური რესურსების მართვის ელექტრონული ტექნოლოგიების დანერგვა სახელმწიფო სტრუქტურებში ხელს შეუწყობს ადამიანურ რესურსებთან დაკავშირებული მონაცემების, ინფორმაციის, სერვისის, მონაცემთა ბაზების, მეთოდებისა და გადაწყვეტების, გაზიარებასა და განვითარებას [53-55].

გასული ათწლეულის მდგომარეობით, კვლევაში მონაწილე უწყებებიდან მხოლოდ რამდენიმეს ჰქონდა დაახლოებით 47% ადამიანური რესურსების ელექტრონული პროგრამა (ნახ.3.1). ადამიანური რესურსების პროგრამული სისტემის არარსებობას რესპოდენტები ასეთი მიზეზებით ხსნიდნენ:

- არასაკმარისი საბიუჯეტო რესურსები;
- ცოდნისა და ექსპერტიზის არაადეკვატური ხარისხი;
- სერვერის მოცულობის სიმცირე;
- ამორტიზირებული/მოძველებული კომუნიკაციები.

შენიშვნა: დღევანდელი მდგომარეობით სიტუაცია მკვეთრად გაუმჯობესებულია. ადამიანური რესურსების მართვის ელექტრონული სისტემა დაინერგა თითქმის ყველა სახელმწიფო სტრუქტურაში, რის შედეგადაც ერთიან მონაცემთა სათავსოში მნიშვნელოვანმა ინფორმაციამ მოიყარა თავი.



ნახ. 3.1. HRMS სისტემების არსებობა საქართველოს სახელმწიფო უწყებებში (2016 წ.)

3.3. ელექტრონული მართვის სისტემის მოდულები: თანამშრომლის პირადი ბარათი

ელექტრონული მართვის სისტემაში ძირითადი რგოლი არის *თანამშრომელი*, რომლის შესახებ ხდება ინფორმაციის რეგისტრირება როგორც ხელით, ასევე ავტომატურად, მოდულებს შორის ინფორმაციის გაცვლის საშუალებით, შესაბამისი სერვისების საფუძველზე [53,55].

ყველა თანამშრომელი სისტემაში რეგისტრირდება ინდივიდუალურად და ხედავს თავის სამუშაო გარემოს, სისტემაზე დაშვებები უნდა იყოს განსაზღვრული და რეგულირდებოდეს შიგა მოხმარების ნორმატიული აქტებით.

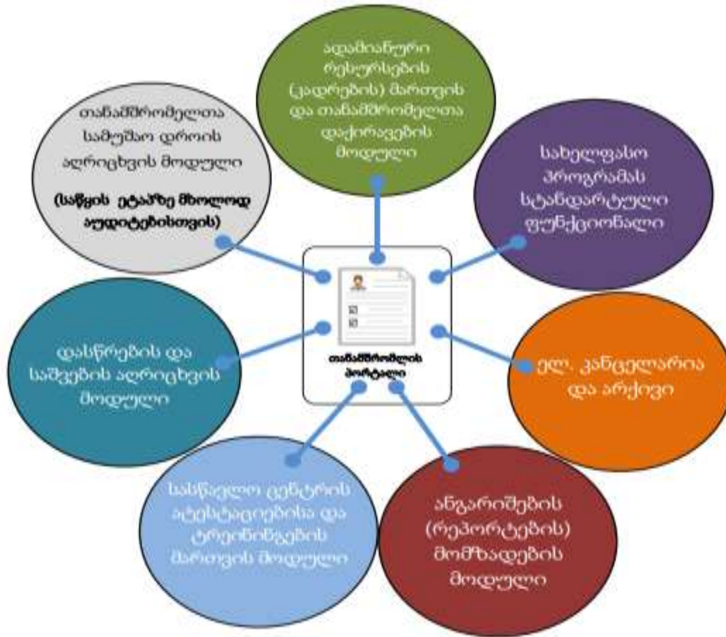
ელექტრონული მართვის სისტემა უნდა მოიცავდეს 3.1 ცხრილში აღწერილ მოდულებს. მოდულის საშუალებით თითოეულმა მომხმარებელმა, თანამდებობრივი ჯგუფის (დონის, სტატუსის) შესაბამისად უნდა მიიღოს ინფორმაცია და ასევე შეასრულოს მოქმედებები.

ელექტრონული კადრების მართვის სისტემის მოდულები **ცხრ.3.1**

N	მოდული	აღწერა
1	ა.) ორგანიზაციის სტრუქტურა. ბ.) თანამდებობების სტრუქტურა (დაქვემდებარების იერარქიული ხე)	ორგანიზაციაზე, სტრუქტურულ ერთეულზე ან თანამდებობაზე განხორციელებული მოქმედებების აღრიცხვა. ორგანიზაციის სტრუქტურის აგება, ცვლილება.
2	თანამშომელთა მართვა (პიროვნების ბარათი).	თანამშრომლის ბარათზე განხორციელებული მოქმედებების აღრიცხვა. თანამშრომლის დანიშვნა, გადანიშვნა, გათავისუფლება, თანამდებობების შეთავსება. თანამშრომლებზე ინფორმაციის მუდმივი განახლება.

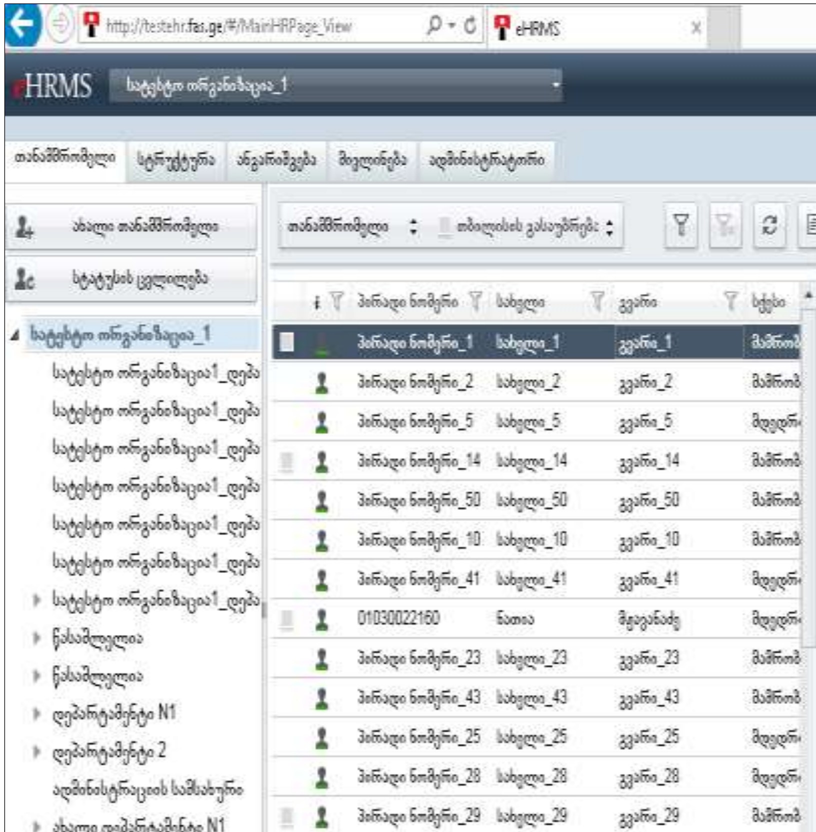
3	ახალი თანამშრომელის დაქირავება.	ვაკანსიის გამოცხადება, გადაწყვეტილების ასახვა. სისტემაში ახალი თანამშრომლის დარეგისტრირება და თანამდებობაზე დანიშვნა.
4	თანამშრომლების დასწრების და საშვების აღრიცხვა.	არსებული დაშვების სისტემასთან ინტეგრაცია და საშვების აღრიცხვა.
5	სასწავლო ცენტრის მართვის, ატესტაციებისა და ტრეინინგების მოდული.	შეხვედრების რეზერვირება და დასწრების და შედეგების ასახვა.
6	თანამშრომელთა მიერ დახარჯული სამუშაო დროის აღრიცხვის მოდული.	მხოლოდ აუდიტების მიერ შესავსები ფორმების შევსება.
7	სახელფასო პროგრამა, სტანდარტული მინიმ. ფუნქციონალით.	მხოლოდ სახელფასო სიის მომზადება (სტანდარტული ფუნქციონალი).
8	პერიოდული და მიმდი ნარე ანგარიშების მომ-ზადებისა და ბეჭდვის მოდული.	წინასწარ მომზადებული შაბლონების მიხედვით ანგარიშების მომზადება.
9	სისტემის ადმინისტრირების მოდული.	სისტემის მართვა, მომხმარებლების წვდომის ადმინისტრირების პანელი.
10	შემოსული, გასული და შიგა წერილების აღრიც ხვის (კანცელარიის) მოდული	შემოსული გასული წერილების რეესტრი და მარშუტის და რეზოლუციების ისტორია

ახლა განვიხილოთ HRMS სისტემის ზოგიერთი მნიშვნელოვანი მოდული (ნახ. 3.2). ძირითადი ინფორმაცია თანამშრომლის შესახებ განთავსებულია პირად ბარათზე და კლასიფიცირებულია ჩანართებში.



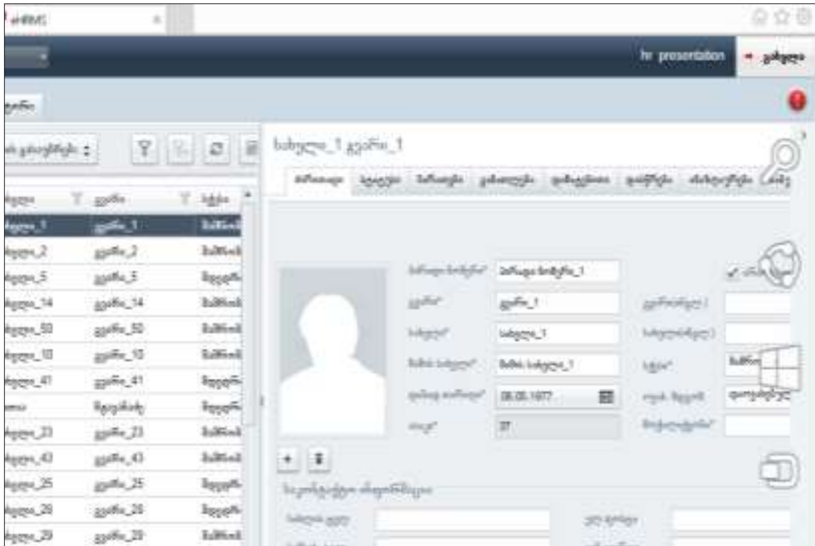
ნახ. 3.2. სისტემის მოდულების ზოგადი დიაგრამა

საწყის გვერდზე გამოტანილია სისტემაში დარეგისტ-რირებული თანამშრომლების სია (ნახ.3.3). თანამშრომლის ჩანაწერის მონიშვნის შემდეგ (ლურჯი ფერით გამოკვეთილი სტრიქონი) გააქტიურდება თანამშრომლის პირადი ბარათის ჩანართი (ნახ.3.4). პირად ბარათზე დატანილია თანამშრომ-ლის შესახებ ძირითადი ინფორმაცია (ფოტოსურათი, პირადი ნომერი, საკონტაქტო ინფორმაცია, ფაქტობრივი და იურიდი-ული მისამართი, მოქალაქეობა, ინფორმაცია ნასამართლე-ობისა და ჯანმრთელობის მდგომარეობის შესახებ და ა.შ.).



ნახ. 3.3. თანამშრომლების სია

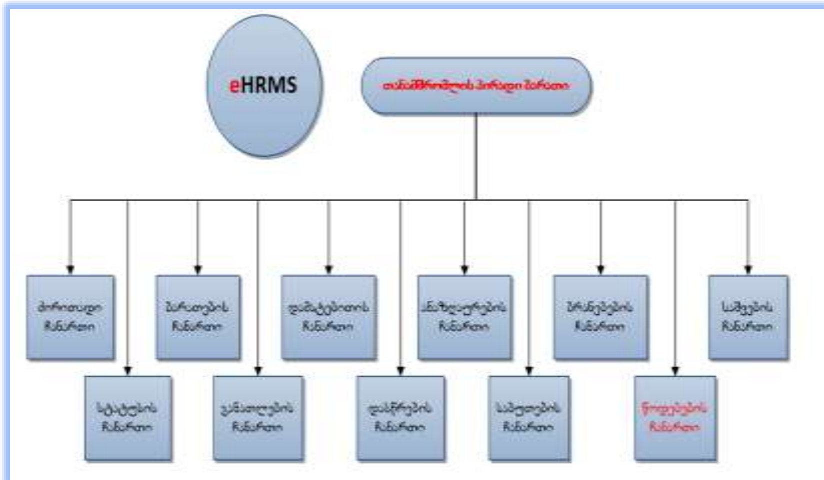
გარდა ძირითადი ინფორმაციის ტაბელისა, თანამშრომლის პირად ბარათზე დატანილია დამატებითი ინფორმაცია, რომელიც ლოგიკურად გადანაწილებულია შესაბამის ტაბებში: „სტატუსი“, „ბარათები“, „განათლება“, „დასწრება“, „ანაზღაურება“ და სხვ. 3.4 ნახაზზე ილუსტრირებულია აპლიკაციის საწყისი გვერდი [56].



ნახ. 3.4. თანამშრომლის პირადი ბარათი, ძირითადი ინფორმაციის ჩანართი

ახალი ფუნქციონალის დამატება საკმაოდ იოლია. მაგალითად, შევჩერდეთ წოდების მოდულზე და გავაანალიზოთ, როგორ არის წოდების ფუნქციონალი ჩაშენებული თანაშრომლის პირად ბარათზე (ნახ.3.5).

ახალი ფუნქციონალის – სპეციალური წოდების დამატების შედეგად, კადრების თანამშრომლებს შესაძლებლობა ექნებათ ინფორმაცია იქონიონ თანამშრომელთა მიერ მიღებული სპეც-წოდებების შესახებ. ასევე, სრულად ასახონ თანამშრომელზე არსებული წოდებების ისტორია, კერძოდ, დაარეგისტრირონ ამა თუ იმ სახელმწიფო ორგანიზაციებში მინიჭიებული წოდებების ნუსხა და აწარმოონ შესაბამისი სტატისტიკა.



ნახ. 3.5. პირადი ბარათის ჩანართები

კადრების თანამშრომელთათვის მნიშვნელოვანი ფუნქ-ცია იქნება წოდების ვადის გასვლის და მომდევნო წოდების მინიჭების შესახებ ინფორმაციის არსებობა (შეტყობინების სახით), წარდგინების დროულად მომზადების მიზნით [56, 57].

რეგისტრირებული წოდების მიხედვით, სისტემაში შესაძლებელი იქნება რეპორტების გენერირება საჭიროები-დან გამომდინარე.

ამ ამოცანის ტექნიკური იმპლემენტაციისთვის საჭირო ძირითადი სამუშაოები ასე გამოიყურება: მონაცემთა სათავსოში ვამატებთ ახალ ცხრილს ([Person].[Rank]); ვწერთ Insert, Update, Delete და Select პროცედურებს. შემდეგ ეტაპზე, ახალი ცხრილი შემოგვაქვს მონაცემთა წვდომის დონეზე და

აღვწერთ შესაბამის ოპერაციებს; ბიზნეს ლოგიკის დონეზე ვამატებთ ახალ კლასს (RankEO.cs), რომელშიც ჩამოთვლილი იქნება ვალიდაციის წესები და უსაფრთხოების ნორმების გათვალისწინებით (უფლებები და როლები) DataContext ობიექტზე გამოვიძახებთ შესაბამის მეთოდებს (Save, Delete, Select). სერვისის კლასში აღვწერთ ახალ ოპერაციებს;

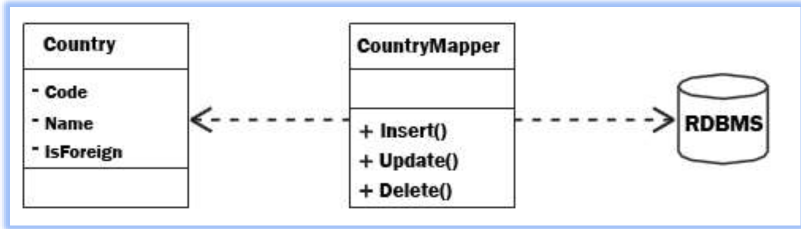
Silverlight კლიენტში განვაახლებთ Service Reference გზავნილს და დავამატებთ ახალ Proxy კლასებს.

ბოლოს, იმპლემენტაციას გავუკეთებთ ViewModel კლასს (RankViewModel.cs) და დავამატებთ წოდებების რეესტრს და მის შესაბამის რედაქტორს (დიალოგულრი ფანჯარა, რომელიც წოდების რედაქტირების ღილაკზე დაჭერის შედეგად გამოდის ეკრანზე).

3.4. მონაცემთა სათავსოსთან წვდომა

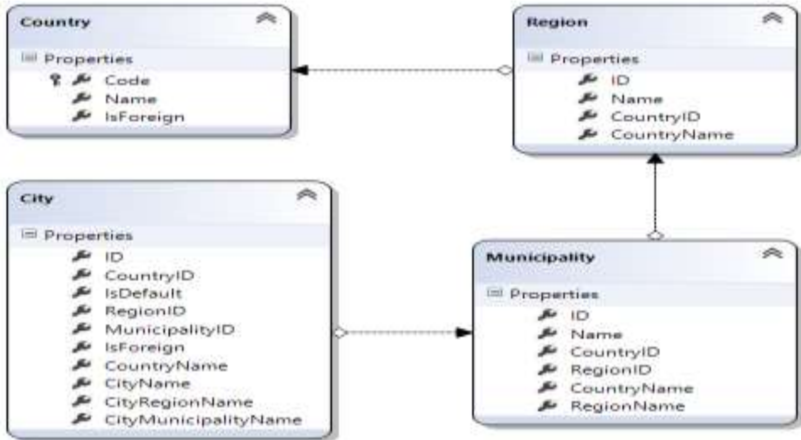
მონაცემთა სათავსოდ გამოყენებული SQL Server რელაციური მონაცემთა ბაზა. პროგრამული უზრუნველყოფა მონაცემთა სათავსოსთან კავშირს ამყარებს მონაცემთა წვდომის მოდულის (DAL) საშუალებით. ამ დონეზე იმპლე-მენტირებულია მონაცემთა სათავსოზე განსაზღვრული ოთხი სტანდარტული ოპერაცია: ახალი ჩანაწერის დამატება, ინფორმაციის ამოკითხვა, ჩანაწერის ცვლილება და წაშლა (CRUD). მონაცემთა წვდომის დონე ასრულებს დამაკავშირე-ბელი რგოლის როლს რელაციურ მონაცემთა სათავსოსა და პროგრამულ მოდულებს შორის. HRMS სისტემაში გამოყენე-ბული DataMapper სტანდარტი უზრუნველყოფს რელაციური

მონაცემთა სათავსოს ცხრილების ასახვას ობიექტზე ორიენტირებულ კლასებსა და სტრუქტურებში (ნახ.3.6).



ნახ. 3.6. DataMapper სტანდარტი: რელაციურ მონაცემთა სათავსოს ჩანაწერების ასახვა Country კლასში

მონაცემთა წვდომის დონეზე გამოყენებულია LINQ to SQL ბიბლიოთეკა, რომელიც ავტომატურად აგენერირებს შუალედურ რგოლს (ნახ.3.7). იგი მონაცემთა ბაზის სტრუქტურის მიხედვით ქმნის დომეინის ობიექტებს.



ნახ. 3.7. LINQ-to-SQL დიზაინერში მონაცემთა ბაზის ცხრილების ასახვა C# ობიექტებში

დამაკავშირებელი ფაილი (.dbml) შეიცავს გრაფიკულ რედაქტორს. ამ გარემოში ხდება DataContext ობიექტის გენერაცია, რომელიც გამოიყენება მონაცემთა ბაზაზე ოპერაციების შესასრულებლად.

გრაფიკულმა რედაქტორმა დააგენერირა Country ცხრილის შესაბამისი C# კლასი, ხოლო DataContext ობიექტის გამოყენებით ვახორციელებთ მოანცემთა ბაზაზე ქმედებებს. DataContext ობიექტი შეიცავს დომეინის თითოეული ტიპის (C# კალასი, Entity) შესახებ ინფორმაციას და იცის თუ რომელი ცხრილი შეესაბამება ამა თუ იმ დომენის ტიპს რელაციურ მონაცემთა სათავსოდან (mapping).

DataContext<Country> წევრი არის System.Data.Linq .Table<User> ტიპის ობიექტი და მისი მოძიება შესაძლებელია როგორც LINQ ენის სინტაქსით, ასევე IEnumerable ინტერ-ფეისზე დაშენებული extension-მეთოდების გამოყენებით.

ქვემოთ კოდის ლისტინგში ნაჩვენებია Country ობიექ-ტის მოძიება. გამოყენებულია ორივე ტიპის სინტაქსი.

```
DataContext dataContext = new DataContext(connection);
// LINQ style ----------
var matchingCountries = from country in dataContext.Countries
where country.Name.Contains("Geo")
select country;

// IEnumerable extensions style ---------
IEnumerable<Country> matchingCountries =
  dataContext.Countries.Where(c => c.Name.Contains("Geo"));
```

LINQ to SQL კომპონენტი მონაცემთა ბაზაში არსებული ცხრილების Relationship კავშირების გაანალიზების შედეგად დომეინ მოდელის C# კლასებზე კავშირებს ავტომატურად დააგენერირებს.

კლასები აღწერილია partial რეზერვირებული სიტყვით, ამიტომ მეთოდების დამატება მარტივად შეგვიძლია, მოდელის ტიპზე რაიმე ქვე-კლასის აღწერის ან ობიექტში გახვევის გარეშე.

გარდა ამისა, DataContext ობიექტს ვიყენებთ მონაცემთა ბაზაში აღწერილი Stored Procedures პროგრამული კოდის გამოსაძახებლად, და ეს პროცესი არაფრით განსხვავდება სტანდარტული მეთოდების გამოძახებისგან.

DataContext ობიექტის ჩატვირთვა და შენარჩუნება მეხსიერებიდან საკმაოდ დიდ რესურსს მოითხოვს, ამიტომ იგი ყოველი მეთოდის დასაწყისში იქმნება და შესასრულე-ბელი სამუშაოების დასრულების შემდეგ ნადგურდება.

LINQ to SQL კომპონენტის გამოყენებით იმპლემენ-ტირებულია Unit of Work სტანდარტი, რაც გულისხმობს მოდელზე ჩატარებული ოპერაციების ერთ, ატომარულ ტრანზაქციებად დაყოფას.

მონაცემთა წვდომის დონეზე თითოეული დომეინის ტიპისთვის იქმნება C# კლასები, რომლებშიც აღწერილია დომენის ტიპზე შესაძლო ოპერაციები: ჩანაწერების მოძიება და წამოღება, წაშლა და სხვ. ქვემოთ მოყვანილ ლისტინგში ნაჩვენებია CityData.cs კლასის კოდის ფრაგმენტი, რომელშიც

ქალაქების მოძიება განხორციელებულია ქვეყნისა და რეგიონის მიხედვით:

```
public List<City> SelectListByCountryRegion(string _CountryID, Guid?
_RegionalID)
{
   DataContext db = null;
   try
   {
      db = CreateContext();
      HRMngSystemContext.ObjectTrackingEnabled = false;
      return db.GetTable<City>().
         Where(W => ((string.IsNullOrEmpty(_CountryID)) ? true
               : W.City_CountryID == _CountryID)
            && ((_RegionalID == null) ? true
               : W.City_RegionID == _RegionalID))
         .OrderByDescending(N => N.City_IsDefault)
         .ThenBy(A => A.CityName).ToList();
   }
   catch { return null; }
   finally { CloseContext(db); }
}
```

3.5. ბიზნეს-ლოგიკის დონე

პროგრამული უზრუნველყოფის ლოგიკა დანაწევრებულია ცალკეულ შრეებში. მონაცემთა წვდომის დონის (DAL) შემდეგ, ზედა საფეხურზე იმყოფება ბიზნეს-ლოგიკის დონე (BLL). იგი აერთიანებს ბიზნეს-წესებს, ვალიდაციებს და უსაფრთხოებას (უფლებების ნაკრები, რომელთა საშუალე-

ბით განისაზღვრება ამა თუ იმ მომხმარებლისთვის სისტემაში დაშვებული ოპერაციები). თითოეული დომეინის ტიპისთვის არსებობს შესაბამისი კლასი ბიზნეს ლოგიკის დონეზე.

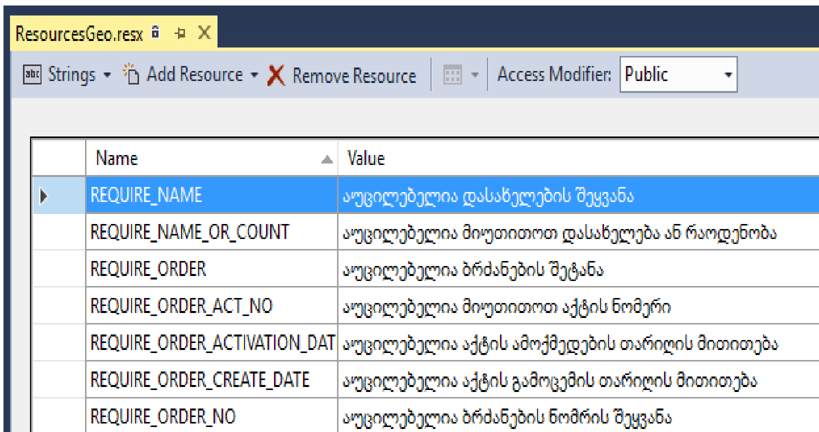
განვიხილოთ CityBO.cs კლასი, რომელიც შეიცავს ქალაქის ობიექტებთან მუშაობის ლოგიკას. იგი შეიცავს City ობიექტების წამოღების, ჩაწერის და წაშლის მეთოდებს.

ეს მეთოდები თავის მხრივ, იძახებს მონაცემთა წვდომის დონეზე განსაზღვრულ CityData.cs კლასის ფუნქციონალს. ობიექტის შენახვის და წაშლის წინ ხდება ბიზნეს წესების და ვალიდაციების შემოწმება. მაგალითად, განვიხილოთ ქალაქის შენახვის ლოგიკა. შენახვის მეთოდის (Save) გამოძახების წინ ხდება ვალიდაციების მეთოდის გამოძახება (ValidateSave) (ნახ. 3.8):

```
protected override bool ValidateSave(DataContext _db, City cityBO)
{
   if (string.IsNullOrEmpty(cityBO.CityName))
ErrorMessage.Add(ResourceLibrary.ResourcesGeo.REQUIRE_NAME);
   if (string.IsNullOrEmpty(cityBO.CountryID))
ErrorMessage.Add(ResourceLibrary.ResourcesGeo.REQUERE_COUN
TRY_NAME);
   if (!ErrorMessage.IsError)
     {  return true;   }
   else
      return false;
}
```

ნახ. 3.8. სავალდებულო ველების ვალიდაცია City ობიექტის შენახვის წინ

იმ შემთხვევაში, თუ რომელიმე სავალდებულო ველი არ არის შევსებული, შენახვის მეთოდის გამოძახება არ მოხდება და მომხმარებლის გარემოში გამოვა ErrorMessage ობიექტში დამატებული შესაბამისი ინფორმაციული შეტყობინება. ეს ტექსტები გატანილია ცალკეულ რესურს-ფაილებში. მოყვანილ მაგალითში გამოყენებულია ქართულ-ენოვანი რესურსის ფაილი (ResourcesGeo.resx) (ნახ.3.9).



ნახ. 3.9. ქართულ ენოვანი რესურსის ფაილი

ვალიდაციის წარმატებით გავლის შემდეგ მოხდება Save მეთოდის გამოძახება. შენახვის ლოგიკას წინ უძღვის ლოგირება (ნახ.3.10). მონაცემთა ბაზაში ინახება ინფორმაცია, თუ რომელმა მომხმარებელმა შეასრულა ოპერაცია და რა მონაცემები შეცვალა. City ობიექტის შემთხვევაში ლოგირე-ბისას ინახება ქალაქის შესახებ ინფორმაცია (რეგიონი, მუნიციპალიტეტი, დასახელება და სხვ.).

```
[Serializable]
9 references
public class CityEO : BaseWithSecurityEO<City>
{

    0 references
    protected override bool Save(City cityBO, DataContext _db,
                     Guid _ActiveUserID, SessionTokenData Session)
    {
        #region Log Data

        ePassportHelper helper = new ePassportHelper();
        string methodName = helper.GetMethodName(MethodBase.GetCurrentMethod());

        helper.dataSource.Add("ID", ((Guid?)cityBO.ID).ToString());
        helper.dataSource.Add("RegionID", cityBO.RegionID.ToString());
        helper.dataSource.Add("MunicipalityID", cityBO.MunicipalityID.ToString());
        helper.dataSource.Add("CountryID", cityBO.CountryID.ToString());
        helper.dataSource.Add("Name", cityBO.CityName);
        helper.dataSource.Add("IsDefault", (cityBO.IsDefault == true) ? "True" : "False");
        helper.dataSource.Add("IsForeign", (cityBO.IsForeign == true) ? "True" : "False");

        #endregion

        if (IsNewRecord())
        {
            Insert
        }
        else
        {
            Update
        }
        return true;
    }
```

ნახ. 3.10. ცვლილებების ლოგირება შენახვის წინ

ლოგირების ინფორმაციის მომზადების შემდეგ პროგ-რამული ლოგიკა გადაინაცვლებს შენახვის ოპერაციაზე. IsNewRecord() მეთოდი აბრუნებს True ან False მნიშვნელობას, იმის და მიხედვით, ახალი ობიექტის დამატების ოპერაცია უნდა შესრულდეს თუ არსებულის რედაქტირება.

განვიხილოთ Insert ლოგიკა. თუ IsNewRecord() მეთოდი დააბრუნებს ჭეშმარიტ მნიშვნელობას, პროგრამის მსვლელობა გადაინაცვლებს #Insert რეგიონში. მოხდება სესიის და მომხმარებლის უფლებების შემოწმება – თუ სესიის ვადა ამოიწურა ან მომხმარებელს არ აქვს მითითებული უფლება (ჩვენს მაგალითში Admin_Tab_City_Add_Button), მაშინ CheckPermissionAndSession დამხმარე მეთოდი დააბრუნებს შესაბამის შეტყობინებას და შენახვის ოპერაცია არ განხორციელდება (ნახ.3.11).

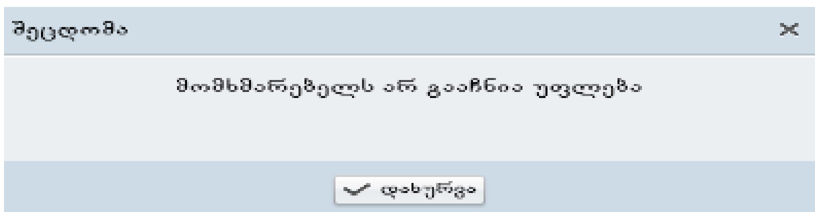
თუ მომხმარებელს აქვს სათანადო უფლება და სესიაც აქტიურია, DataContext ობიექტის საშუალებით ვიძახებთ InsertCity() მეთოდს, რომელიც მონაცემთა ბაზაში აღწერილი პროცედურაა (Stored Procedure).

```
if (IsNewRecord())
{
    #region Insert

    methodName += "(insert)";

    if (helper.CheckPermissionAndSession(
                    Session, methodName,
"Admin_Tab_City_Add_Button"))
    {
        Guid? iD = Guid.Empty;

        _db.InsertCity(ref iD,
            cityBO.CountryID,
            cityBO.RegionID,
```

```
            cityBO.MunicipalityID,
            cityBO.CityName,
            cityBO.IsDefault,
            cityBO.IsForeign,
          _ActiveUserID);

      cityBO.ID = iD;
    }
    else
    {
      ErrorMessage.Add(helper.ErrorText);
      return false;
    }
    #endregion
```

პროცესის წარმატებით დასრულების შედეგად მონაცემთა სათავსოში მოხვდება ახალი ჩანაწერი, ასევე ლოგირების ცხრილში ჩავარდება ინფორმაცია, თუ რომელმა მომხმარებელმა დაამატა ეს ჩანაწერი და რა სახის ინფორმაცია შეიტანა.

იმ შემთხვევაში, თუ მომხმარებელს არ აქვს უფლება ან სესია არ არის აქტიური, გამოდის შეტყობინება (ნახ.3.11).



ნახ. 3.11. შეტყობინება მომხმარებლის უფლების უქონლობის შესახებ

3.6. სერვისის დონე

პროგრამული აპლიკაციების ასაგებ Silverlight ფრეიმ-ვორკს არ აქვს ADO.NET-ის სპეციფიკური კლასები, როგორიცაა DataReader, DataSet და სხვ., რომლებიც საჭიროა მონაცემთა ბაზებთან სამუშაოდ. ამიტომაც მის პროექტებს მონაცემთა ბაზასთან პირდაპირი კავშირის განხორციელება არ შეუძლია.

ასეთი პრობლემის გადაჭრის მიზნით, ბიზნესლოგიკის თავზე დაშენებულ იქნა კიდევ ერთი დონე – *სერვისის დონე*. სწორედ ეს დონე ახორციელებს მონაცემთა ბაზასთან კომუნიკაციას (ანუ კომუნიკაცია წარმოებს *ბიზნესლოგიკისა* და *მონაცემთა წვდომის* დონეების გავლით) და Silverlight აპლიკაციას საჭირო მონაცემებით უზრუნველყოფს.

არ არის სავალდებულო, რომ სერვისიდან მოწოდებუ-ლი მონაცემები უშუალოდ მონაცემთა ბაზიდან მოდიოდეს. შესაძლებელია, მათი რომელიმე სხვა სერვისიდან მიღება. მაგალითად, *ადამიანური რესურსების მართვის სისტემა* იყენებს საჯარო რეესტრის სერვისს, რომელიც პირადი ნომრის მიხედვით აბრუნებს პიროვნების პირად ინფორმა-ციას (***სახელი, გვარი, იურიდიული მისამართი, მოქალაქეობა*** და სხვ.).

პროგრამული უზრუნველყოფის სერვისის დონის იმპლემენტირებისას გამოვიყენეთ WCF სერვისი (Windows Communication Foundation) [58-60]. იგი აიოლებს სერვის-ორიენტირებული პროგრამული აპლიკაციის შემუშავებას და პროგრამისტ-დეველოპერებში პოპულარობით სარგებლობს.

სერვისზე ორიენტირებულ Silverlight პროექტზე მუშაობისას WCF კარგი არჩევანია, რადგან მაღალი ხარისხის კონტროლის საშუალებას იძლევა. შესაძლოა განისაზღვროს:

- რა ტიპის ობიექტებს (კლასებს) ვაწვდით Silverlight კლიენტს;
- კლასის თითოეული წევრისთვის (Field) განვსაზღვროთ მოხვდება თუ არა იგი კლიენტის მხარეს გადაგზავნილ ასლში.

სერვისთან სამუშაოდ პროგრამულ უზრუნველყოფაში შემოგვაქვს სპეციალური კლასები, რომლებიც მონიშნულია [DataContract] ატრიბუტით, ხოლო კლასის წევრები მონიშნულია [DataMember] ატრიბუტით. ამ ატრიბუტების შედეგად მოცემული კლასი და მისი წევრები მოხვდება კლიენტის მხარეს დაგენერირებულ ვერსიაში. ეს ატრიბუტები მუშაობს System.Runtime.Serialization ბიბლიოთეკით.

მაგალითისთვის განვიხილოთ City ობიექტის სერვისის მხარეს გამოსატანად ჩატარებული სამუშაოები. სერვისის კლასებს ვამატებთ სპეციალურ პროექტში, დასახელებით ServiceDataLibrary. ეს პროექტი დომეინ მოდელის ტიპებს აღწერს სერვისთან სამუშაო ფორმატში [DataContract] და [DataMember] ატრიბუტების მითთებით. კლასის ის ცვლადები, რომლებზეც [DataMember] არ არის მითითებული, Silverlight კლიენტში არ გამოჩნდება. ასევე, ის კლასები, რომლებზეც არ არის მითითებული [DataContract] ატრიბუტი, Silverlight კლიენტში არ გამოჩნდება. City

დომენის ტიპისთვის ServiceDataLibrary პროექტში დავამატეთ CityBO_SLData.cs კლასი (იხ. ლისტინგი).

```
namespace ServiceDataLibrary
{
   [DataContract]
   public partial class CityBO_SLData : BaseBO_SLData
   {
      private string _Name;
      [DataMember]
      public string Name
      {
         Get {return _Name; }
         set
         {
            if (_Name != value)
            {
               _Name = value;
               OnPropertyChanged("Name");
            }
         }
      }
               ...
               private Nullable<Guid> _RegionID;
      [DataMember]
      public Nullable<Guid> RegionID ...
               private Nullable<Guid> _ MunicipalityID;
      [DataMember]
      public Nullable<Guid> MunicipalityID...
...
```

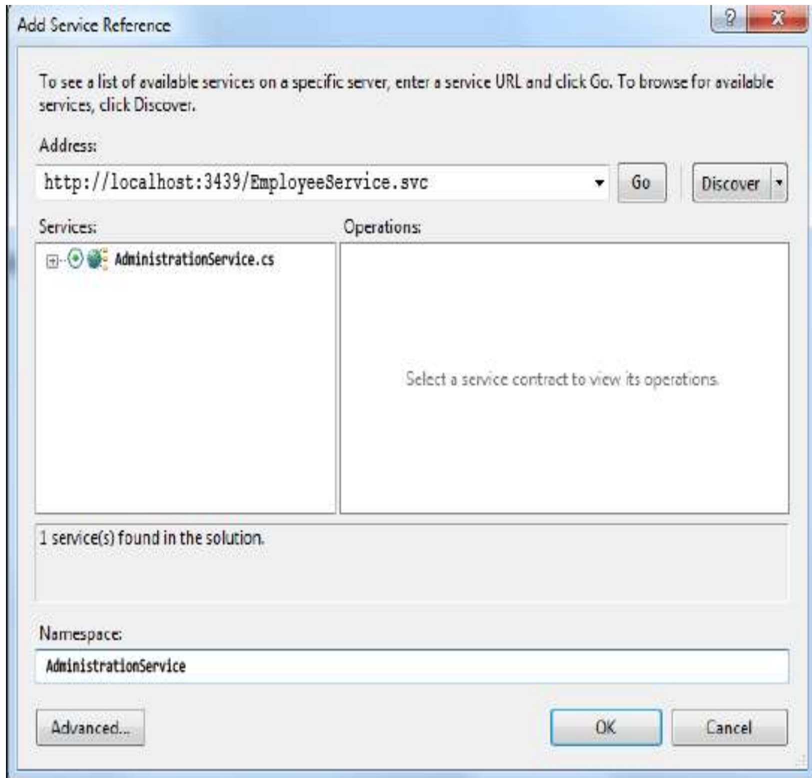
მოდელის და რეპოზიტორიის გამზადების შემდეგ, WCF სერვისში ვამატებთ მეთოდებს, რომელიც Silverlight კლიენტის მხარეს უნდა მივაწოდოთ სერვისიდან.

გავაგრძელოთ *ქალაქების* მაგალითის განხილვა. ამ შემთხვევაში სერვისმა კლიენტის მხარეს უნდა გახსნას ოთხი მეთოდი: *ერთი ქალაქის წამოღება, ქალაქების სიის წამოღება, ქალაქის შენახვა, ქალაქის წაშლა.*

ყოველი მეთოდი უნდა მოინიშნოს [OperationContract] ატრიბუტით. WCF სცენარში, თავად სერვისიც უნდა მოვნიშნოთ [ServiceContract] ატრიბუტით. სერვისის დამატებისას VisualStudio ხელსაწყოში გენერირდება ორი ფაილი: *.svc და *.svc.cs გაფართოებით. ატრიბუტების მითითება სერვისის კონტრაქტების ფაილში ხდება (*.svc.cs). ქალაქების შემთხვევაში სერვისის ფაილის კოდის ფრაგმენტი მოყვანილია ქვემოთ:

```
[ServiceContract]
public interface IAdministrationService
{
...
[OperationContract]
CityBO_SLData GetCityEO(Guid _KeyValue,
  Guid _ActiveUserID, SessionTokenData Session);
[OperationContract]
BaseBOList_SLData<CityBO_SLData> GetCityEOList(
  string _CountryID, Guid? _RegionalID,
  Guid _ActiveUserID, SessionTokenData Session);
[OperationContract]
bool SaveUpdateCityEO(ServiceDataLibrary.CityBO_SLData
_CityBO_SLData,
  Guid _ActiveUserID, SessionTokenData Session);
[OperationContract]
bool DeleteCityEO(Guid _KeyValue,
  Guid _ActiveUserID, SessionTokenData Session);
```

სერვისის პროექტის გზავნილი Silverlight პროექტში დამატებულია პროქტზე Add Service Reference მითითებით და AdminitrationService მონიშვნით (ნახ.3.12).



ნახ. 3.12. Administration.svc სერვისზე გზავნილის დამატება Silverlight კლიენტის პროექტში

ამ ეტაპზე სერვისის მხარე გამართულია და გადავინაცვლოთ ViewModel კლასების შემცველ დონეზე. იგი არქიტექტურაში წარმოდგენილია სერვისის ზედა რგოლზე.

ViewModel დონე ინახავს პროგრამული დანართის მდგომარეობას და განსაზღვრავს პრეზენტაციის დონეზე ჩასატარებელ ოპერაციებს და ცვლილებებს.

ViewModel კლასში სერვისის მეთოდის გამოსაძახებლად უნდა შევქმნათ Proxy კლასის ეგზემპლიარი. თავად Proxy კლასებს Visual Studio ხელსაწყო აგენერირებს სერვისზე გზავნილის დამატების/განახლების პროცესში.

შემდეგ ეტაპზე მივუთითებთ ე.წ. callback მეთოდს, რომელშიც ველოდებით სერვისის მიერ დასაბრუნებელ შედეგს.

WCF სერვისთან კომუნიკაცია ასინქრონულად მიმდინარეობს. ბოლოს, ვიძახებთ სერვისის მეთოდს, როგორც ქვემოთ მოყვანილ პროგრამულ კოდში არის ნაჩვენები:

```
private void LoadCityList()
{
   AdministrationService.AdministrationServiceClient proxy =
          New SilverlightAdministrationBrowser
              .AdministrationService.AdministrationServiceClient();
   proxy.GetCityEOListCompleted += proxy_GetCityEOListCompleted;
   proxy.GetCityEOListAsync();
}
```

თუ callback მეთოდში შეცდომა არ დაფიქსირდა, სერვისის მიერ დაბრუნებულ შედეგს:

AdministrationService.GetCityEOListCompletedEventArgs

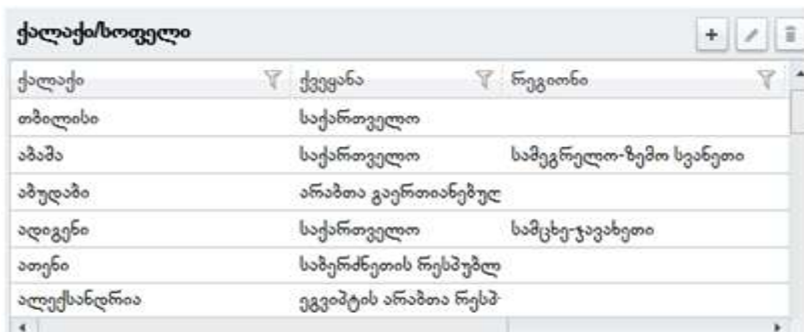
პარამეტრის Result წევრიდან მივწვდებით. ეს property-წევრი იმავე ტიპისაა, რაც სერვისის მეთოდის მიერ დაბრუნებული ობიექტი, ამ შემთხვევაში:

List<SilverlightAdministrationBrowser.AdministrationService .CityBO_SLData>).

სამომხმარებლო ინტერფეისზე DataGrid კონტროლში გამოსატანად გრიდის ItemSource property-წევრს ვანიჭებთ სერვისის მეთოდის დაბრუნებულ შედეგს:

```
void LoadCitiesCompleted(object sender,
   SilverlightAdministrationBrowser.AdministrationService
      .GetCityEOListCompletedEventArgs e)
{
  if (e.Error == null)
      CityDataGrid.ItemsSource = e.Result;
   else
    ErrorMessagesBOList.Add(
         "სერვისთან კომუნიკაციის დროს დაფიქსირდა შეცდომა");
}
```

შედეგად DataGrid კონტროლში ჩაიტვირთება ქალაქების სია (ნახ.3.13).



ნახ. 3.13. სამომხმარებლო ინტერფეისზე ქალაქების სიის გამოტანა

3.7. Silverlight-ით მდიდარი და ინტერაქტიული სამომხმარებლო ინტერფეისის იმპლემენტაცია

Silverlight ტექნოლოგია სხვა ვებ-ტექნოლოგიებთან შედარებით უფრო აიოლებს მდიდარი და ინტერაქტიული სამომხმარებლო ინტერფეისების შექმნას.

მდიდარი სამომხმარებლო ინტერფეისის კარგი მაგალითია *„თანამშრომელთა დასწრების ტაბელი*" ადამიანური რესურსების მართვის ელექტრონულ სისტემაში (eHRMS). დასწრების ჩანართზე შესაძლებელია სხვადასხვა სახის დასწრების და გაცდენის აღრიცხვა.

დასწრებების აღრიცხვა შესაძლებელია ორი გზით:

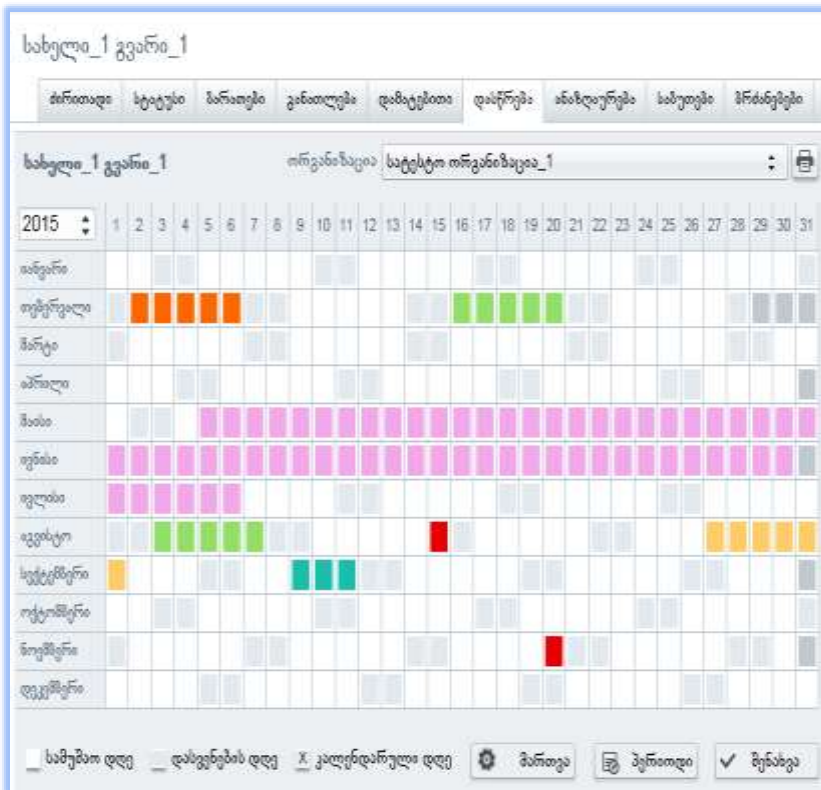
- თითოეულ უჯრაზე დადგომით და სასურველი ფერის არჩევით შეივსოს მონაცემები და შემდეგ მოხდეს შენახვა;
- *ახალ პერიოდზე* დაჭერის შედეგად მიეთითოს სასურველი პერიოდი და დასწრების ტიპი. შეცდომით შევსებული დასწრების შემთხვევაში, შესაძლებელია „კალენდარული დღის" ამორჩევა დასწრების ტიპებიდან, რაც გამოიწვევს სტანდარტული ტიპის ველების დაბრუნებას.

იმ შემთხვევაში, თუ საჭიროა შევსებული ფერის წაშლა, შესაბამის უჯრაში უნდა მიეთითოს „კალენდარული დღე".

დასწრების ტაბელი (ნახ. 3.14) წარმოადგენს კონტროლს (User Control), რომელიც შემდეგი ოპერაციების შესრულების საშუალებას იძლევა (ცხრ.3.2).

დასწრების ტაბელზე ოპერაციების ჩამონათვალი ცხრ.2.2

ოპერაციის დასახელება	ღილაკი
დასწრების/არყოფნის დღის მონიშვნა	კალენდარზე თარიღის აღმნიშვნელი უჯრაზე მაუსის კურსორის დაწკაპება
სამუშაო დღის მონიშვნა	კალენდრის თარიღის უჯრის მონიშვნა და სამუშაო დღე ღილაკზე დაკლიკვა
დასვენების დღის მონიშვნა	კალენდრის თარიღის უჯრის მონიშვნა და დასვენების დღე ღილაკზე დაკლიკვა
კალენდარული დღის მონიშვნა	კალენდრის თარიღის უჯრის მონიშვნა და X კალენდარული დღე ღილაკზე დაკლიკვა
დასწრების ტიპის მითითება	კალენდრის თარიღის უჯრის მონიშვნა და დასწრების სტატუსების სიიდან შესაბამისი ფერის ამსახველ ღილაკზე დაკლიკვა
დასწრების პერიოდის შენახვა	დასწრების პერიოდის ღილაკი

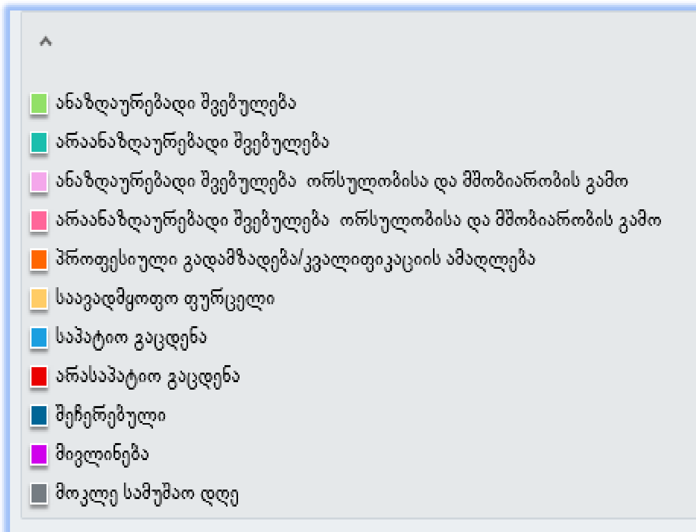


ნახ. 3.14. თანამშრომლის დასწრების ამსახველი კალენდარი

ა) დასწრების დღეების მონიშვნა – სისტემაში დარეგის-ტრირებული პირის არყოფნის შემთხვევაში შესაბამისი დასწრების ტიპის ამორჩევა და კალენდარზე თარიღის აღმნიშვნელი უჯრის მონიშვნა (თვისა და რიცხვის თანაკვეთაზე). შედეგად უჯრა მიიღებს არყოფნის ტიპის შესაბამის ფერს და „შენახვა“ ღილაკზე დაჭერის შემდეგ თანამშრომლის დასწრება შეინახება მონაცემთა ბაზაში.

ბ) დასწრების პერიოდის შენახვა – პერიოდის ღილაკზე დაჭერის შედეგად ეკრანზე გამოდის ფორმა, რომელზეც მომხმარებელი უთითებს პერიოდის საწყის და საბოლოო თარიღებს და დასწრების ტიპს. შენახვის ღილაკზე დაჭერის შემდეგ ხდება პერიოდის შესაბამისი თარიღების შენახვა მონაცემთა ბაზაში. მაგალითად, თუ მონიშნულია პერიოდი 1 მარტიდან 7 მარტის ჩათვლით, ბაზაში ჩავარდება 7 ცალი თარიღის შესაბამისი სტრიქონი.

არყოფნის ტიპის მონიშვნა ხდება სპეციალური ჩამოსაშლელი სიის მეშვეობით, რომელიც განთავსებულია კალენდრის კონტროლის ქვემოთ (ნახ.3.15):



ნახ. 3.15. დასწრების სტატუსების ჩამოსაშლელი სია

კალენდრის კონტროლი არის ე.წ. User Controls-ი და რეალიზებულია დამოუკიდებელ .xaml-ფაილში (Calendar-Control.xaml). კალენდრის კონტროლი თანამშრომლის ძირითად ბარათზე ჩასმულია შემდეგი .xaml დირექტივით (ნახ.3.16):

```
<my2:CalendarControl
    x:Name="myCalendar"
    DataContext="{Binding CalendarViewModel}"
    Visibility="{Binding UserRoleList.UserRole,
        ConverterParameter=EmplCalendarInfoTab,
        Converter={StaticResource
HasUserRoleVisibility}, Mode=OneWay}"
    Width="Auto"
    LabelHeader="{Binding PersonFullName,
Mode=TwoWay}"/>
```

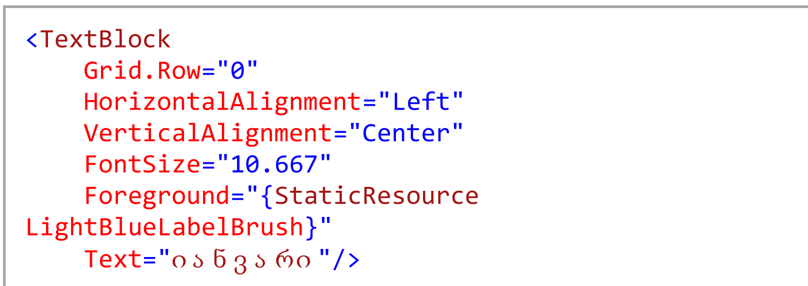
ნახ. 3.16. დასწრების კალენდარი თანამშრომლის პირად ბარათზე

კალენდრის კონტროლი <Grid> კონტროლში გაერთი-ანებული კონტროლების ერთობლიობას, რომელიც შედგება სტრიქონებისა (<Grid.RowDefinitions>) და სვეტებისაგან (<Grid.ColumnDefinitions>).

სტრიქონები შეესაბამება თვეებს, ხოლო სვეტები – თვის რიცხვებს (1 -დან 31-ის ჩათვლით). თვეების დასახელებები გამოტანილია <Grid> კონტროლს პირველ სვეტში ჩადგმულ <StackPanel> კონტეინერში ვერტიკალური თანმიმდევრობით დალაგებული <Label> კონტროლების საშუალებით (ნახ.3.17).

```
<StackPanel x:Name="st_MonthList"
    Margin="5,0,7,0"
    Orientation="Vertical"
    Grid.Row="1"
    Grid.RowSpan="12"
    d:LayoutOverrides="Height">
</StackPanel>
```


ნახ. 3.17. StackPanel კონტეინერი შედგება თვის დასახელებების <label> კონტროლებისგან



```
<TextBlock
    Grid.Row="0"
    HorizontalAlignment="Left"
    VerticalAlignment="Center"
    FontSize="10.667"
    Foreground="{StaticResource
LightBlueLabelBrush}"
    Text="ი ა ნ ვ ა რი "/>
```

ნახ. 3.18. იანვრის თვის ამსახველი <Label> კონტროლი

მაგალითად, ერთ-ერთი <Label> კონტროლი, რომელიც ჩადებულია <StackPanel> კონტეინერში და შეესაბამება ***იანვრის*** თვეს.

თითოეული თვის გასწვრივ განლაგებული 31 უჯრა მოთავსებულია ჰორიზონტალური თანმიმდევრობით <Grid> კონტროლის სვეტებში და შეესაბამება თვის დღეებს.

მაგალითად, ***31*** რიცხვის ამსახველი უჯრა xaml ენაზე ასე გამოიყურება (ნახ.3.19).

```
<my:CalendarItemControl
    DataContext="{Binding}"
    Grid.Column="31"
    Grid.Row="1"
    HorizontalAlignment="Stretch"
    Margin="0.5"
    x:Name="January31"
    VerticalAlignment="Stretch"
    Width="Auto"/>
```

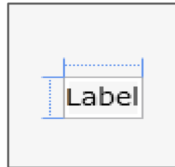
ნახ. 3.19. თვის 31-ე დღის ამსახველი უჯრა

3.20 ნახაზზე ნაჩვენებია მთელი წლის თვეების და რიცხვების ცხრილი – კალენდრის კონტროლისთვის.

ნახ. 3.20. კალენდრის კონტროლი (CalendarControl.xaml)

CalendarItem_Ctrl არის დამოუკიდებელი კონტროლი (User Control), რომელიც რეალიზებულია ცალკე ფაილში და შემდეგ მისი გამოყენება შესაძლებელია ნებისმიერ სხვა კონტროლში შესაბამისი დირექტივის ჩართვის შემდეგ (ნახ.3.21):

xmlns:my="clr-namespace: ContentLibrarySL.Controls"



ნახ. 3.21. CalendarItemControl.xaml

იგი შედგება <Grid> კონტროლში განთავსებული <Rectangle> კონტროლისაგან, რომელიც შეფერილია „არ-ყოფნის“ ტიპის შესაბამისი ფერით და <sdk:Label> კონტრო-ლისაგან.

Interaction Trigger მექანიზმით ხდება CalendarItem-Control კონტროლზე მაუსის მარცხენა ღილაკის დაკლიკვის (EventName="MouseLeftButtonDown") მომენტალური დაჭერა, ხოლო ChangePropertyAction ტრიგერის მეშვეობით ხდება გააქტიურებული უჯრისთვის ფერის მინიჭება.

(CalendarBO_SLData ობიექტის ShiftsDay_BackColor ელემენტი, რომელშიც ფერის მნიშვნელობა აღწერილია როგორც int ტიპის მთელი მნიშვნელობის რიცხვი).

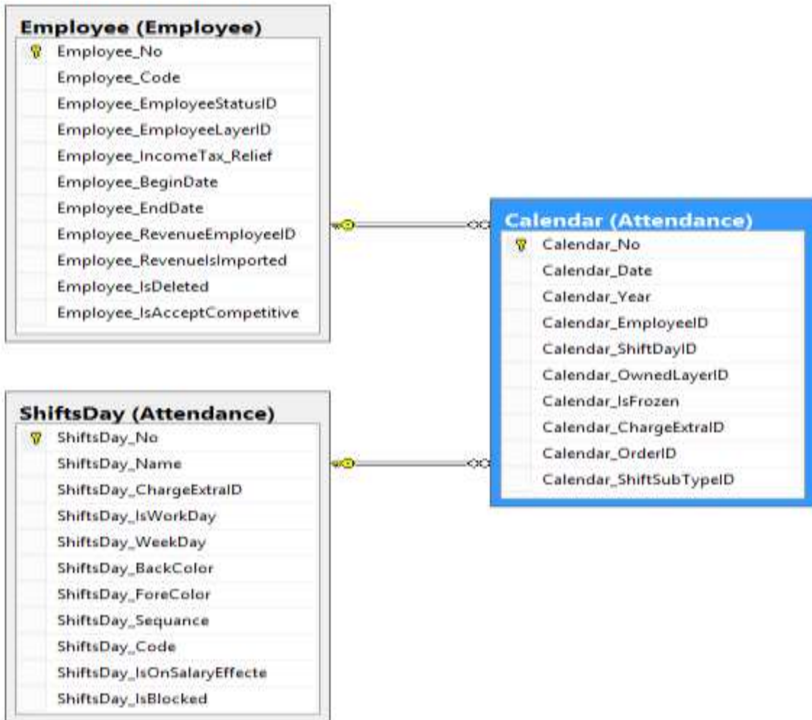
ქვემოთ მოყვანილია CalendarItem_Ctrl კონტროლის .xaml მარკირების კოდის ლისტინგი:

```
<UserControl x:Class="ContentLibrarySL.Controls.CalendarItemControl"
    xmlns:MySlData="clr-
namespace:ServiceLibrary;assembly=ServiceLibrarySL"
    xmlns:MyConvertor="clr-namespace: ContentLibrarySL.Convertors"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
  xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
  xmlns:ei="http://schemas.microsoft.com/expression/2010/interactions"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentati
on/sdk"
    mc:Ignorable="d"
    Height="23"
    Name="CalendarItemControl">
<UserControl.Resources>
   <MySlData:CalendarBO_SLData x:Key="CalendarBO_SLData"
d:IsDataSource="True" />
   <MyConvertor:ConvertorINTToRGB x:Key="ConvertorINTToRGB" />
   <MyConvertor:BoolToColor x:Key="BoolToColor" />
</UserControl.Resources>

<Grid x:Name="LayoutRoot" >
<Rectangle x:Name="CalendarItem_Rect"
   Margin="0"
   StrokeThickness="2"
   DataContext="{Binding Source={StaticResource CalendarBO_SLData}}"
   Stroke="{Binding Path=Calendar_IsSelected,
```

```
   Converter={StaticResource BoolToColor}}"
   Fill="{Binding Path=ShiftsDayBO.ShiftsDay_BackColor,
Mode=OneWay,
   Converter={StaticResource ConvertorINTToRGB}}" />

<sdk:Label Name="CalendarItem_TextBlock"
   FontSize="14"
   HorizontalAlignment="Stretch"
   VerticalAlignment="Stretch"
   VerticalContentAlignment="Center"
   HorizontalContentAlignment="Center"/>

<i:Interaction.Triggers>
   <i:EventTrigger EventName="MouseLeftButtonDown">
      <ei:ChangePropertyAction
         PropertyName="ClickItem"
         Value="{Binding Source={StaticResource CalendarBO_SLData}}"
         TargetObject="{Binding ElementName=CalendarItem_Control,
Path=DataContext}" />
   </i:EventTrigger>

   <i:EventTrigger EventName="Loaded" >
      <ei:ChangePropertyAction  PropertyName="RegisterViewModel"
         Value="{Binding Source={StaticResource CalendarBO_SLData}}"
         TargetObject="{Binding ElementName=CalendarItemControl,
Path=DataContext}" />
   </i:EventTrigger>
</i:Interaction.Triggers>
</Grid>
</UserControl>
```

- *მონაცემთა ბაზის სტრუქტურა*

ინფორმაციის სათავსოდ გამოყენებულია Microsoft SQL Server-ის რელაციური მონაცემთა ბაზა. კალენდრის ფუნქციონალში მონაწილეობს შემდეგი 3 ურთიერთდაკავშირებული ცხრილი (ნახ.3.22).



ნახ. 3.22. მონაცემთა ბაზის სტრუქტურა

- *თანამშრომლის ცხრილი* ([Employee].[Employee]) – ძირითადი ცხრილი, სადაც თავს იყრის სისტემაში დარეგისტრირებული ყველა პირის ჩანაწერი;

- *თანამშრომლის დასწრების ცხრილი* ([Attendance].[Calendar]) – სადაც კალენდრის კონტროლზე „*შენახვა*" ღილაკზე დაჭერის შედეგად წამოსული ინფორმაცია იყრის თავს: თანამშრომლის უნიკალური *ID* (Calendar_EmployeeID), *არყოფნის ტიპი* (Calendar_ShiftsDayID), *თარიღი* (Calendar_Date), *წელიწადი* (Calendar_Year), *ბრძანების დოკუმენტის უნიკალური ID* (Calendar_OrderID) და სხვ.
- *არყოფნის ტიპების ცხრილი* ([Attendance].[ShiftsDay]) – ესაა გაცდენების აღრიცხვის ტიპი, რომელიც ხასიათდება ფერადი ინდიკატორით (ShiftsDay_BackColor), დასახელებით (ShiftsDay_Color) და *კოდით* (ShiftsDay_ChargeExtraID).

ფერის მნიშვნელობები ბაზაში შენახულია int მთელი ტიპის მნიშვნელობათა რიცხვის სახით (ShiftsDay_BackColor სვეტი ცხრილში Attendance.ShiftsDay) (ნახ.3.23):

	ShiftsDay_No	ShiftsDay_Name	ShiftsDay_ChargeExtraID	ShiftsDay_BackColor
▶	bc737a6f-abd1-4f32-9a36-3b...	არაანაზღაურებადი შვებულება	WITHOUTSAL	-1818285
	59e12689-70cd-429f-9f7c-52...	ანაზღაურებადი შვებულება	VACATION	-9625704
	6efc8e43-96d3-440b-96e5-5a...	მივლინება	BUSINESSTRIP	-13631724
	829827d7-178a-49e1-93fb-73...	შეჩერებული	STOPPED	-26006
	3ec95123-410a-4e0a-af50-9e...	ანაზღაურებადი შვებულება ორსულობ...	DECREEWITHSAL	-16033771
	faa3224b-08d2-44a2-b39a-d8...	არასაპატიო გაცდენა	INADEQUATE	-15335424
	6b2da58d-f32e-4cc9-b430-e8...	საპატიო გაცდენა	FORGIVEN	-1810400
	a81ff87d-a3bd-431f-a1c0-eb...	არაანაზღაურებადი შვებულება ორსულ...	DECREEWITHOUTSAL	16737945
	5a3d7d59-0f17-4707-8c85-f3...	მოკლე სამუშაო დღე	SHORTDAY	-7699586
	e418f8ea-e99c-483e-92d0-f8...	დასვენების დღე	WEEKDAY	-14871023
	e418f8ea-e99c-483e-92d0-f8...	პროფესიული გადამზადება/კვალიფიკ...	TRAINING	-16737792
	e428f8ea-e99c-483e-92d0-f8...	საავადმყოფო ფურცელი	BULLETIN	-16764006
*	NULL	NULL	NULL	NULL

1 of 12

ნახ. 3.23. არყოფნის ტიპების ამსახველი ცხრილი მნაცემთა ბაზაში

როგორც ვხედავთ, სამომხმარებლო ინტერფეისზე მთელი რიცხვის ნაცვლად ჩანს ფერადი უჯრა. ეს რეალიზებულია Silverlight ტექნოლოგიის მძლავრი მონაცემთა ბმის (Data Binding) მექანიზმის დახმარებით, კერძოდ კი IConverter ინტერფეისის მქონე converter-ობიექტის მეშვეობით.

3.8. Silverlight Value Converter ობიექტი

Silverlight ტექნოლოგიის კიდევ ერთი მნიშვნელოვანი უპირატესობა არის მძლავრი Data Binding მექანიზმი და მონაცემთა კონვერტაცია. Data Binding მექანიზმი საშუალებას იძლევა სისტემაში არსებული ინფორმაცია გამოვიტანოთ სამომხმარებლო ინტერფეისზე XAML დირექტივებით ან პროგრამული კოდით.

როდესაც მონაცემების გამოტანა ხდება, შესაძლოა საჭირო გახდეს ინფორმაციის დაფორმატება (მაგალითად, თარიღის შემთხვევაში საათების და წუთების ჩამოშორება და ინტერფეისზე გამოტანა ფორმატით dd/MM/yyyy, ან ფერის შემთხვევაში მთელი ტიპის რიცხვის ნაცვლად ფერის ჩვენება). *ამისათვის Silverlight აპლიკაციაში ხდება value converter - ობიექტის შექმნა*, რომელიც შემდეგ აპლიკაციის სხვადასხვა ადგილას შეგვიძლია გამოვიყენოთ მრავალჯერადად და გარდავქმნათ მონაცემთა ბაზიდან წამოსული მონაცემები ჩვენთვის *სასურველ ფორმატსა თუ ფერში* ან თუნდაც ობიექტში [45].

განვიხილოთ კალენდრის შემთხვევაში ფერის აღმნიშვნელი რიცხვის (მაგალითად, „არასაპატიო გაცდენა“ ტიპის შესაბამისი წითელი ფერის ექვივალენტი მთელი

რიცხვი -15335425) კონვერტაცია შესატყვის RGB მნიშვნელობის მქონე SolidColorBrush ობიექტში.

Value Converter ობიექტის შექმნის მიზნით პროექტში უნდა ჩავამატოთ Silverlight-ის კლასი და იმპლემენტაცია გავუკეთოთ ინტერფეისს IConverter (განთავსებულია System.Windows.Data namespace ბიბლიოთეკაში).

ეს ინტერფეისი აღწერს ორ წევრს: Convert და ConvertBack:

- Convert – გამოიყენება წყარო-ობიექტიდან სამომხმარებლო ინტერფეისზე არსებული კონტროლისთვის Data Binding მექანიზმით ცვლადის მნიშვნელობის ასახვისას. Convert-ი ცვლის მონაცემის მნიშვნელობას (ჩვენს შემთხვევაში რიცხვი -15335425 გადაყავს წითელ ფერში);
- ConvertBack – პირიქით, გადაყავს მონაცემი პირვანდელ სახეზე (მაგალითად, მომხმარებელმა შეცვალა არყოფნის ტიპი და სხვა ფერი მონიშნა, ConvertBack-ი ახალი ფერის მნიშვნელობას გადაიყვანს ექვივალენტ int მთელ რიცხვად და ისე გადააწოდებს პროგრამული კოდის უკანა მხარეს).

ორივე წევრის: Convert და ConvertBack შესაბამის მეთოდებს გადაეცემათ ერთი და იგივე ტიპის პარამეტრები, ანუ მეთოდის სიგნატურა (method signature) არის ერთი და იგივე, როგორიც მოყვანილია ქვემოთ:

```
object Convert(object value, Type targetType, object parameter,
            CultureInfo culture);
object ConvertBack(object value, Type targetType, object parameter,
            CultureInfo culture);
```

IValueConverter ინტერფეისის იმპლემენტაციის შედეგად კონვერტერის აღწერა დასრულებულია და იგი მზადაა სამომხმარებლო ინტერფეისზე გამოსაყენებლად.

კონვერტორის გამოსაყენებლად საჭიროა კლასის namespace-ის დეკლარაცია იმ XAML ფაილში, სადაც კონვერტერის გამოყენება გვინდა. ჩვენს შემთხვევაში უნდა მოხდეს CalendarItemControl.xaml ფაილის დასაწყისში შემდეგი კოდის დამატება:

```
xmlns:MyConvertor="clr-
namespace:HRMngSystemV3_ClientLibrarySL.Convertors"
```

და აგრეთვე სამომხმარებლო კონტროლის რესურსების აღწერის სექციაში შემდეგი კოდის ჩამატება:

```
<UserControl.Resources>
   <MyConvertor:ConvertorINTToRGB x:Key="ConvertorINTToRGB" />
</UserControl.Resources>
```

ამის შემდეგ კონვერტერის დადება ხდება უშუალოდ იმ კონტროლზე, რომელიც Data Binding მექანიზმით დაკავშირებულია დასწრების ტიპის ფერის ამსახველ ცვლადთან:

```
<Rectangle x:Name="CalendarItemRectangle"
...
Fill="{Binding Path=ShiftsDayBO.ShiftsDay_BackColor, Mode=OneWay,
Converter={StaticResource ConvertorINTToRGB}}" />
```

INTToRGB კონვერტერის სრული პროგრამული კოდი მოყვანილია ქვემოთ ლისტინგში:

```
// -- IntToRGB კონვერტორის კოდი C# ენაზე----
using System;
using System.Windows.Data;
using System.Windows.Input;
using System.Windows.Media;

namespace ContentLibrarySL.Convertors
{
  public class ConvertorINTToRGB : IValueConverter
  {
    #region IValueConverter Members

    public object Convert(object value, Type targetType,
      object parameter, System.Globalization.CultureInfo culture)
    {
     Nullable<int> TmpIntValue = value as Nullable<int>;
     Brush brush = null;

     if (TmpIntValue.HasValue)
     {
      byte r;
      byte g;
      byte b;
      int B = ( int ) TmpIntValue;

      r = System.Convert.ToByte(Math.Abs(((B / 256) / 256) % 256));
      g = System.Convert.ToByte(Math.Abs((B / 256) % 256));
      b = System.Convert.ToByte(Math.Abs(B % 256));
      brush = new SolidColorBrush(Color.FromArgb(255, r, g, b));
```

```
      return brush;
      }
      else
      {
      return brush;
    }
  }

public object ConvertBack(object value, Type targetType,
       object parameter, System.Globalization.CultureInfo culture)
  {
   SolidColorBrush brush = value as SolidColorBrush;

   if (brush != null)
    {
      byte r = brush.Color.R;
      byte g = brush.Color.G;
      byte b = brush.Color.B;
      return r * 256 * 256 + g * 256 + b;
    }
   else return null;
  }

    #endregion
   }
}
```

3.9. ICommand ინტერფეისის რეალიზაცია კალენდრის მაგალითზე

მომხმარებელი თუ აჭერს „*შენახვა*“ ღილაკს, მაშინ მონაცემთა ბმის (Data Binding) მექანიზმის საშუალებით ხდება კალენდარზე მონიშნული ინფორმაციის გადაწოდება პროგრამული ლოგიკის მხარისთვის. კერძოდ კი Calendar-Control კონტროლის ViewModel კლასში (CalendarListBO-ViewModel) არსებული Command ობიექტისათვის (SaveData-Command). ეს კავშირი განხორციელებულია xaml კოდის შემდეგი ლისტინგით:

```
<!-- Command-ობიექტის ბმა სამომხმარებლო ინტერფეისზე -->
<myControls:CustomButton
   x:Name="cbSaveButton"
   ToolTipService.ToolTip="შენახვა"
   Content="შენახვა}"
   Icon="/ContentLibrarySL;component/Images/SaveBtn.png"

MouseOverIcon="/ContentLibrarySL;component/Images/SaveBtnRe
d.png"
   MainShadowEffect="{StaticResource ButtonDropShadowEffect}"
   Command="{Binding DataContext.SaveDataCommand,
         ElementName=CalendarControl}"
           IsEnabled="{Binding UserRoleList.UserRole,
           ConverterParameter=EmplCalendarInfoTab_Save_Button,
           Converter={StaticResource HasUserRoleEnabledAction},
           Mode=TwoWay}"
   IconHeight="16" IconWidth="16"
   Margin="5,0,0,0" Width="90"/>
```

Model-View-ViewModel (MVVM) არქიტექტურული ნიმუშის ერთ-ერთი მთავარი უპირატესობაა სამომხმარებლო ინტერფეისის ლოგიკისა (Presentation Logic) და ბიზნეს-ლოგიკის (Business Logic) ურთიერთგამიჯვნა.

View და ViewModel დონეებს შორის კავშირი განხორციელებულია მონაცემთა ბმის (Data Binding) მექანიზმის, Command ობიექტების და Notification Event მოვლენების მეშვეობით.

კალენდრის მაგალითზე გამოყენებული SaveDataCommand ობიექტი არის ICommand ინტერფეისის იმპლემენტაცია. როგორც უკვე აღვნიშნეთ, ეს ინტეფეისი გვხვდება MVVM აპლიკაციების უმრავლესობაში და შედგება შემდეგი 3 წევრისგან [11]:

- Execute(object) მეთოდი – გამოიძახება როდესაც Command -ის აქტივაცია ხდება. ამ მეთოდს გადაეცემა ერთი პარამეტრი, რომელიც გამოიყენება გამომძახებელი ობიექტის მიერ Command-ობიექტისთვის დამატებითი ინფორმაციის გადასაწოდებლად;
- CanExecute(object) მეთოდი – აბრუნებს Boolean ტიპის მნიშვნელობას: true მნიშვნელობა ნიშნავს, რომ Command ობიექტის შესრულება დაშვებულია, ხოლო false მნიშვნელობის შემთხვევაში XAML კონტროლი (რომელზეც დამაგრებულია Command ობიექტი) ავტომატურად მიიღებს გაუქმებულ (disabled) მდგომარეობას. პარამეტრად გადაცემული object ტიპის ცვლადი ამჯერადაც იგივეა, რაც Execute() მეთოდის შემთხვევაში;

- CanExecuteChanged – არის Event Handler ობიექტი. იგი აღიძვრება Command ობიექტის იმპლემენტაციის დროს, როდესაც საჭიროა ხელახლა გამოითვალოს CanExecute მეთოდი. XAML კოდში კონტროლის Command ატრიბუტზე მონაცემთა ბმის მექანიზმით ხდება ICommand ობიექტის დაკავშირება, ხოლო CanExecuteChanged მოვლენის აღძვრა გულისხმობს CanExecute() მეთოდის ავტომატურ გამოძახებას და კონტროლის მდგომარეობის ცვლილებას: ჭეშმარიტი მნიშვნელობის შემთხვევაში გახდება აქტიური (Enabled), მცდარი მნიშვნელობის შემთხვევაში კი – Disabled.

„*შენახვა*"-ღილაკზე დაჭერის შედეგად ViewModel-ის შესაბამის Command ობიექტს გადაეცემა კონტროლი და ხდე-ბა ვებ-სერვისის მეთოდის გამოძახება. შენახვის ოპერაციას გარკვეული დრო ჭირდება, რომლის განმავლობაშიც სამომხმარებლო ინტერფეისი მიუწვდომელია.

ვებ-სერვისთან კომუნიკაციის მანძილზე ეკრანზე ტრიალებს Silverlight Toolkit-ის სპეციალური კონტროლი Busy Indicator, რომელიც მომხმარებელს ამცნობს პროცესის მსვლელობის შესახებ და ქრება პროცესის დასრულების-თანავე.

3.10. მესამე თავის დასკვნა

სერვის-ორიენტირებული არქიტექტურა (SOA) არა მხოლოდ Web-სერვისებს მოიცავს, არამედ წარმოადგენს არქიტექტურულ კონცეფციას, რომელიც IT სისტემების ინტეგრაციის საშუალებას იძლევა თავისუფლად დაწყვილე-ბული, საზიარო სერივსების გამოყენებით. ხშირად, Web-

სერვისი არის სერვის-ორიენტირებული RIA იმპლემენტა-ციების ერთ-ერთ ყველაზე მნიშვნელოვანი ნაწილი. რადგან Silverlight ტექნოლოგიით აგებული RIA-აპლიკაციების საფუძველი არის .NET პლატფორმა, ამიტომ პროგრამულ დანართში მიიღწევა პრეზენტაციის დონის აბსტრაქცია და მოდულარული არქიტექტურა.

SOA არქიტექტურის საფუძველზე და RIA პრინციპების გათვალისწინებით შემუშავებულია ადამიანური რესურსების მართვის ელექტრონული სისტემა. Silverlight ტენოლოგიის გამოყენებით სამომხმარებლო ინტერფეისი გამდიდრებულია ეფექტური ანიმაციებით, ტრანსფორმაციებით, კლიენტის მხარის ვალიდაციებით და მონაცემთა ფორმატირების საშუალებებით.

პროგრამის სერვერული ნაწილი აგებულია კომპონენ-ტების ლოგიკურ შრეებზე გადანაწილების არქიტექტურული მიდგომით. ცალკეულ შრეებზე განთავსებული ლოგიკური კომპონენტები ერთმანეთთან კავშირში იმყოფება და ინფორმაციის მიმოცვლა მიმდინარეობს სპეციალურად გამოყოფილი პროგრამული ინტერფეისების საშუალებით.

სერვერულ ნაწილსა და Silverlight-კლიენტს შორის კავშირი მიღწეულია WCF სერვისის და Proxy კლასებით.

მონაცემთა საცავი შემუშავებულია Ms SQL Server რელაციურ მონაცემთა ბაზაზე. ეს ხელსაყრელია ანგარიშ-გებათა გენერაციის და სისტემიდან ინფორმაციის ამოღების თვალსაზრისით. პროგრამა შეიცავს ანგარიშგების მოდულს და მომხმარებელს საშუალება აქვს მოიძიოს მისთვის საინტერესო ინფორმაცია, დააგენერიროს ანგარიშგება ან მოახდინოს ექსპორტირება რომელიმე ფორმატით (PDF, Word, Excel).

თავი 4
მონოლითური და მიკროსერვისული არქიტექტურების მიმოხილვა

4.1. გავრცელებული არქიტექტურული მიდგომები

საინფორმაციო ტექნოლოგიების სწრაფმა განვითარებამ ბუნებრივად განაპირობა ინფორმაციული სისტემების სტრუქტურის, ფუნქციონალური და არქიტექტურული მოდელების დინამიკური ცვლილება და შემდგომი ევოლუციური სრულყოფა [1,2,20-22,41]. საინფორმაციო სისტემების მიმართ მოთხოვნების გაზრდასთან ერთად შეიცვალა სისტემის არქიტექტურული მიდგომებიც და მაქსიმალურად გახდა დეცენტრალიზებული, კომპლექსური სისტემების შემთხვევაში.

ასეთი ორგანიზაციული მართვის (მენეჯმენტის) სისტემების შემუშავებისას რეკომენდებული არქიტექტურული მოდელის გადაწყვეტა არაერთ საკითხზეა დამოკიდებული, როგორიცაა სისტემის შინაარსი, ზომა და კომპლექსურობა, მის მიმართ არსებული ბიზნეს- და ტექნოლოგიური მოთხოვნები, მოსალოდნელი განვითარების მასშტაბები და ტენდენციები და ა.შ.

განვიხილოთ მოკლედ პრაქტიკაში გავრცელებული მონოლითური და განაწილებული სისტემების არქიტექტურული მიდგომები, და მათი უპირატესობის და ნაკლოვანების ასპექტები.

4.1.1. მონოლითური არქიტექტურა

მონოლითური არქიტექტურა არის სისტემის ისეთი დიზაინი, რომელიც ცენტრალიზებულად იმართება როგორც ერთი დიდი მთლიანობა. სისტემების კლასიფიკაციის ერთ-ერთი ზოგადი თეორიით ისინი იყოფა დიდ და მცირე, რთულ და მარტივ სისტემებად [61-63].

არსებობს სისტემები, რომელთა არქიტექტურაც ცალსახად მოითხოვს ცენტრალიზებულ გადაწყვეტას. მცირე და მარტივ სისტემებში, რომლებიც არ განიცდის ხშირ ფუნქციონალურ და დატვირთვის ცვალებადობას, მონოლითური არქიტექტურის უპირატესობა აქ კარგ კომფორტს იძლევა.

მონოლითური არქიტექტურის შემთხვევაში, სისტემები ხასიათდება რიგი თავისებურებებით [22]. მაგალითად, მცირე ზომის პროგრამული აპლიკაციის დამუშავებისას, მონოლითური არქიტექტურა გარკვეულ უპირატესობებს იძლევა:

- აპლიკაციის დანერგვა მარტივი პროცესია, მისი განთავსება და გაშვება ერთი დასანერგი პაკეტის სერვერზე გადატანით შეგვიძლია წარმოვიდგინოთ;
- საწყის ეტაპზე, დეველოპმენტის პროცესი მარტივია, რადგან არ გვხვდება ისეთი ტექნიკური გამოწვევები, როგორიცაა: დეცენტრალიზებულ კომპონენტებს შორის კომუნიკაციის სტანდარტიზება, ოპერაციის ტრანზაქციულობის მართვა, მონაცემთა მდგრადობის შენარჩუნება და ა.შ.

როდესაც, პროდუქტი იზრდება ზომაში და ასევე იმატებს მასზე მომუშავე ადამიანების რიცხვი, ამ დროს

მონოლითური არქიტექტურის შემდეგი ნაკლოვანებები იჩენს თავს:

- დიდი, მონოლითური აპლიკაციის კოდი სისტემის განვითარებასთან ერთად ხდება კომპლექსური და რთულად გასაგები, რაც გამოწვეულია დროთა განმავლობაში მკაცრად იზოლირებული კომპონენტების გართულებით. ირღვევა კოდის მოდულარობა და ფუნქციონალის ლოგიკური ერთეულები (მოდულები) ხდება ერთმანეთზე დამოკიდებული. ეს უარყოფითად აისახება კოდის ხარისხზე;
- რთულია სისტემის კომპონენტების ჩანაცვლება ახალი იმპლემენტაციით;
- ხშირად შეუძლებელია კონკრეტულ ამოცანაზე მორგებული, ოპტიმალური ტექნოლოგიის გამოყენება, რადგან სისტემა როგორც ერთი მთლიანობა, უკვე მორგებულია კონკრეტულ ტექნოლოგიას;
- უწყვეტი მიწოდება (CD - Continuous Deployment) არის პრობლემური: სისტემის ხშირი განახლებადობის პირობებში. პატარა ცვლილების დროსაც, აუცილებელია მთლიანი სისტემის დანერგვა, რაც ხელმისაწვდომობის და ხშირი განახლებადობის კუთხით პრობლემებს წარმოშობს;
- სისტემის ერთ კონკრეტულ ნაწილში არსებული ხარვეზი, ხშირად მთლიანი სისტემის მწყობრიდან გამოსვლას იწვევს;
- გუნდური მუშაობა ხდება პრობლემური: როდესაც აპლიკაცია იზრდება ზომაში და მისი კომპლექსურობაც იმატებს, როგორც წესი, გვიწევს მასზე მომუშავე ადამიანური

რესურსის დაყოფა შესაბამის ლოგიკურ ჯგუფებად. სხვადასხვა ჯგუფების დამოუკიდებლად მუშაობისას მონოლითური არქიტექტურა მოსახერხებელი *არ არის*. ცვლილებების გაკეთება სხვადასხვა ჯგუფის მიერ მოითხოვს დამატებით კოორდინაციას, რომ ერთმა ცვლილებამ სხვა ცვლილება ან არსებული ფუნქციონალი არ დააზიანოს, რაც თავისთავად ნიშნავს იმას, რომ უფრო მეტ დროს და რესურსს ვხარჯავთ;

- მოუქნელი მასშტაბირება (არაოპტიმალური): მონოლითური არქიტექტურის თავისებურებიდან გამომდინარე მასშტაბირება შეიძლება მხოლოდ ერთი მიმართულებით – *ვერტიკალურად*. აპლიკაციის სხვადასხვა კომპონენტებს აქვს სხვადასხვა მოთხოვნები, ზოგ კომპონენტს აქვს დიდი დატვირთვა გამოთვლით რესურსზე (CPU), ზოგს დიდი მოთხოვნა მეხსიერებაზე (Memory) და ამ დროს გამოთვლით რესურსებს მინიმალურად იყენებს. მონოლითური არქიტექტურიდან გამომდინარე, ჩვენ არ შეგვიძლია სხვადასხვა კომპონენტის მასშტაბირება დამოუკიდებლად, ამიტომ გვიწევს ძირითად ინსტანსზე ყველა საჭირო რესურსის გამოყოფა და თუ დგება ჰორიზონტალური მასშტაბირების მოთხოვნა დატვირთვიდან გამომდინარე, ამ დროს არაოპტიმალურად ვიყენებთ ჩვენს ხელთ არსებულ რესურსებს;
- მონოლითური სისტემის ტესტირება გაცილებით რთულია, ვიდრე დისტრიბუციული სისტემის კონკრეტული, მცირე ზომის მქონე, დამოუკიდებელი მპოდულის [20-22].

4.1 ნახაზზე ნაჩვენებია მონოლითური არქიტექტურის მქონე სისტემის ნიმუში.

ნახ. 4.1. მონოლითური არქიტექტურის მქონე სისტემის მაგალითი

როგორც აღვნიშნეთ, არსებობს სისტემები, რომელთა არქიტექტურაც ბუნებრივად მოითხოვს ცენტრალიზებულ გადაწყვეტას თავისი ბიზნეს მოთხოვნებიდან გამომდინარე, თუმცა დიდ კორპორატიულ პროექტებში ხშირად საქმე ასე მარტივად არ არის. ყოველდღიურად ვითარდება ბიზნესის მოთხოვნა, ყოველდღიურად იცვლება და ემატება სისტემას გარკვეული ფუნქციონალი, რომლის ცენტრალიზებული აღქმა ადამიანის გონებისთვისაც წარმოუდგენელი ხდება, აქედან გამომდინარე სისტემის ანალიტიკოსების და არქიტექტორების როლიც სწორედ ისაა, რომ სწორად გათვალონ სისტემის მასშტაბები, მოსალოდნელი განვითა-რების არეალი და დაგეგმვის საწყის ეტაპზე შეიქმნან წარმოდგენა სისტემის ფუნქციონალური კომპონენტების (ერთეულების) შესახებ. დროთა განმავლობაში, შესაძლოა

სისტემის კომპონენტები შეიცვალოს, დაემატოს, გამოაკ-ლდეს, გარდაიქმნას და ა.შ. თუმცა, ძირითადი მიდგომა: განისაზღვროს თუ არა სისტემის დისტრიბუციულობის აუცილებლობა, დომეინიდან გამომდინარე, ანალიზის საწყის ეტაპზევე უნდა გამოიკვეთოს.

4.1.2. მიკროსერვისული არქიტექტურა

საერთაშორისო პრაქტიკული გამოცდილებიდან გამო-მდინარე, მასშტაბური ბიზნეს დომეინის მქონე ამოცანები, სასურველია დანაწევრდეს შედარებით მცირე ზომის ამოცანებად, რომლებიც რეალიზებული იქნება შესაბამის პროგრამული უზრუნველყოფის კომპონენტების სახით. აღნიშნული მიდგომა გვაძლევს საშუალებას, რომ სისტემის კონკრეტული ფუნქციონალური ნაწილი წარმოდგენილი იქნეს როგორც დამოუკიდებელი ერთეული და იყოს მარტივად ინტეგრირებადი სხვადასხვა სისტემაში.

ასეთი მიდგომით რეალიზებული პროგრამული აპლიკაცია *დეცენტრალიზებული ანუ დისტრიბუციული სისტემაა* [64].

მასშტაბური სისტემის დისტრიბუციულ კომპონენტე-ბად დაშლა, დაგეგმვა და განვითარება რთული და შრომატევადი პროცესია. ანალიზის გარდა, ტექნიკურ ასპექტში მრავალი ისეთი დეტალი იჩენს თავს, რომელიც ცენტრალიზებულ მიდგომებში (მონოლითურ არქიტექტუ-რაში) არ გვხვდება. მაგალითად, სისტემის კომპონენტებს შორის კომუნიკაციის მეთოდების (სინქრონული, ასინ-

ქრონული) ანალიზი, სტანდარტიზება და მართვა, პროცესე-ბის ტრანაქციულობის მართვა და მონაცემთა მთლიანობის (Data Consistency) დაცვა, ხელმისაწვდომობას და მონაცემთა მდგრადობას შორის პრიორიტეტების განსაზღვრა და მრავალი სხვ.

მიკროსერვისული არქიტექტურა: დისტრიბუციული სისტემები განვიხილოთ „მიკროსერვისული" არქიტექტურის მაგალითზე.

მიკროსერვისული არქიტექტურა ძირითადად გამოიყე-ნება მემკვიდრეობით და კომპლექსურ სისტემებში. *მიკრო-სერვისების ძირითადი იდეაა სისტემის ლოგიკურ დამოუკიდებელ ერთეულებად (სერვისებად) დაყოფა და დამოუკიდებლად განვითარება.*

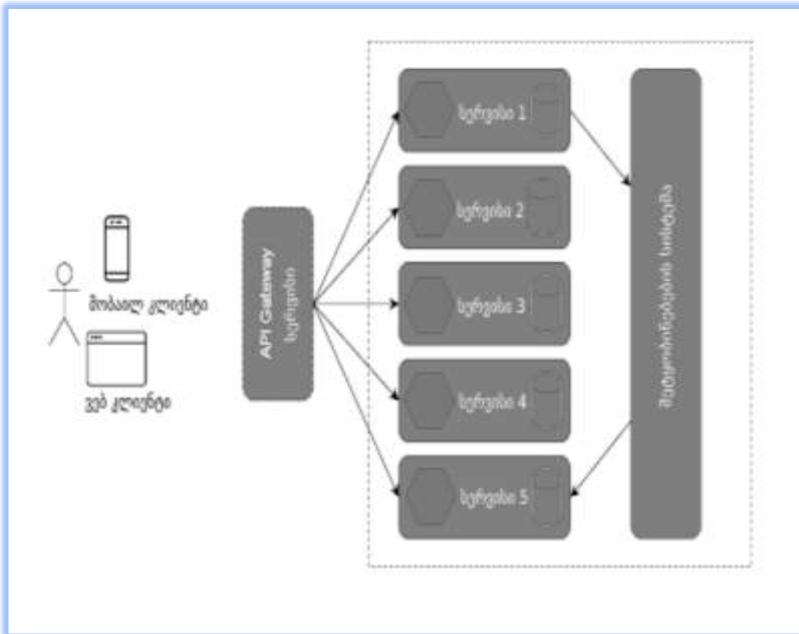
სისტემის დაპროექტების ფაზაში ხდება ბიზნეს მოთხოვნის ღრმა ანალიზი და სისტემის მოდულების წარმოდგენა. მათ შორის საზღვრების ჩამოყალიბება კი ერთ-ერთი მნიშვნელოვანი პროცესია მიკროსერვისული არქი-ტექტურაში. სისტემის სამომავლო განვითარება, მთლიანად ამ საზღვრებსა და მათ ლოგიკურ დეკომპოზიციაზეა დაფუძნებული.

ამ პროცესებისთვის მოქნილი მიდგომაა *დომეინზე ორიენტირებული დიზაინი* (Domain Driven Design – DDD), რომელიც თავისი ძირითადი პრინციპებით, ორიენტირე-ბულია კვლევის ობიექტის ბიზნესმოთხოვნებზე [66]

4.2 ნახაზზე მოცემულია მიკროსერვისული არქი-ტექტურის სისტემის მაგალითი, სადაც ნაჩვენებია

მობაილ/ვებ კლიენტი აპლიკაციიდან დისტრიბუციული სისტემის გამოყენების მაგალითი. სერვისები განთავსებულია დამოუკიდებლად და კლიენტი აპლიკაციები მიკრისერვისებთან კომუნიკაციისთვის იყენებს Api Gateway ვებ სერვისს.

Api Gateway სერვისის ძირითადი დანიშნულებაა კლიენტის მოთხოვნის გადამისამართება შესაბამის მიკროსერვისში (Request Routing - Reverse Proxy) და მონაცემების აგრეგაცია მიკროსერვისებს შორის კლიენტის მოთხოვნის საფუძველზე (Data Aggregation).



ნახ. 4.2. მიკროსერვისული არქიტექტურის სისტემის მაგალითი

სირთულეების მიუხედავად, სისტემის დისტრიბუციულობა მრავალ უპირატესობას გვთავაზობს, როგორებიცაა:

- სისტემის კომპონენტების დამოუკიდებლობა (მიკროსერვისების სხვა კომპონენტებისაგან დამოუკიდებლად შექმნა, განვითარება, მიწოდების შესაძლებლობა);
- სისტემის მოქნილობა როგორც მცირე ცვლილებების მიმართ, ასევე სამომავლო ფართო მოთხოვნების გაჩენის მიმართ;
- კომპონენტების მარტივად ინტეგრირებადობა (სხვადასხვა სისტემაში);
- ტექნიკური და ადამიანური რესურსების ოპტიმალური მოხმარება [67,68];
- ტექნოლოგიების მიმართ აგნოსტიკურობა (თითოეული კომპონენტის ჭრილში სხვადასხვა, ოპტიმალური ტექნოლოგიების გამოყენების შესაძლებლობა და მათ შორის უნივერსალური კომუნიკაციის განსაზღვრის შესაძლებლობა);
- კომპონენტების შეცვლის, განვითარების და ახლით ჩანაცვლების შესაძლებლობა;
- სისტემის მაღალი სიცოცხლისუნარიანობა (როდესაც სისტემაში ფიქსირდება ხარვეზი, შეფერხება ხდება მხოლოდ იმ კომპონენტში სადაც პრობლემაა, სისტემის დანარჩენი ნაწილი კი განაგრძობს ფუნქციონირებას შეფერხების გარეშე);
- სისტემის კომპონენტების როგორც ვერტიკალური, ასევე ჰორიზონტალური მასშტაბირების შესაძლებლობა, მაღალი დატვირთულობის პირობებში. ვერტიკალური მასშტაბირება გულისხმობს რესურსების მართვას მოთხოვნის

მიხედვით კონკრეტული მიკროსერვისის დანერგილი ერთეულისათვის (Instance), ხოლო ჰორიზონტალური მასშტაბირება გულისხმობს მაღალი დატვირთვის პერიოდში მიკროსერვისების დანერგილი ერთეულების მასშტაბირებას (გამრავლებას) მოთხოვნის მიხედვით;

- ჰორიზონტალური მასშტაბირების პროცესში ხარჯების ოპტიმიზაცი;
- სისტემის მარტივი გარჩევადობა (რაც უფრო მცირე, მოდულარული და ლოგიკურად დალაგებულია პროგრამული კოდი, მით უფრო მარტივია მისი აღქმა და გარჩევა, ეს კი ავტომატურად ნიშნავს იმას, რომ ვზოგავთ დროს/რესურსს და ჩვენი სისტემა მარტივად გასაგებია ახალი დეველოპერებისთვის);
- სისტემის შექმნის პროცესის მაქსიმალური დეცენტრალიზება გუნდებს შორის (სისტემის აგება შესაძლებელია განხორციელდეს პარალელურად, სხვადასხვა გუნდების მიერ, რომლებიც დამოუკიდებლად ახდენენ კონკრეტული ბიზნესამოცანის იმპლემენტაციას და გააჩნიათ შესაბამისი კომპეტენცია, ამ ბიზნეს მოთხოვნის ირგვლივ);
- პროცესების მართვის Agile მეთოდოლოგიებზე მარტივად მორგებადობა;
- ბიზნესისა და ტექნიკური გუნდის გამარტივებული კომუნიკაცია, კონკრეტული კომპონენტის შექმნის პროცესში (Domain Driven Design - Ubiquitous language) [66];
- სისტემის ასინქრონულად მუშაობის მოქნილი შესაძლებლობები;

- სისტემის გამარტივებული ტესტირებადობა და სხვ. [20,69].

სისტემის დისტრიბუციულობა მრავალ უპირატესობას გვთავაზობს, თუმცა უნდა აღინიშნოს, რომ კომპლექსუ-რობიდან გამომდინარე, მისი მართვა არაერთ სირთულესთან არის დაკავშირებული. მათ შორის მნიშვნელოვანია შემდეგი მახასიათებლები:

- როდესაც იზრდება კომპონენტების (სერვისების) რაოდენობა და თითო სერვისი არის დამოუკიდებელი ერთეული (მოდული), რთულდება მათი მართვა და მონიტორინგი. თითოეული კომპონენტის ვერსიების მენეჯმენტი, Continuous Delivery - უწყვეტი მიწოდება და განახლება, დატვირთვის მიხედვით ავტომატური მასშტაბი-რება - საკმაოდ დიდი გამოწვევაა, რომლის ადამიანური რესურსით მართვა შეუძლებელი ხდება. ამიტომ აუცილებე-ლია კონტეინერიზაციის პრინციპების შემოტანა და კომპო-ნენტების მართვის ავტომატური ორკესტრირება. ამისათვის გამოიყენება ისეთი ტექნოლოგიები, როგორიცაა Docker, Kubernetes, Docker Swarm, Openshift და სხვ.
- კომუნიკაციის კომპლექსურობა – აუცილებელია თავი-დანვე განისაზღვროს კომპონენტებს შორის ძირითადი კომუნიკაციის ტიპები და საშუალებები;
- მონაცემთა მდგრადობა (Data consistency): იქიდან გამომდინარე, რომ Database Per Service მიდგომით, თითო-ეულ სერვისს თავისი დამოუკიდებელი ბაზა აქვს და ერთი ცენტრალიზებული მონაცემთა საცავი არ არსებობს, რთულია

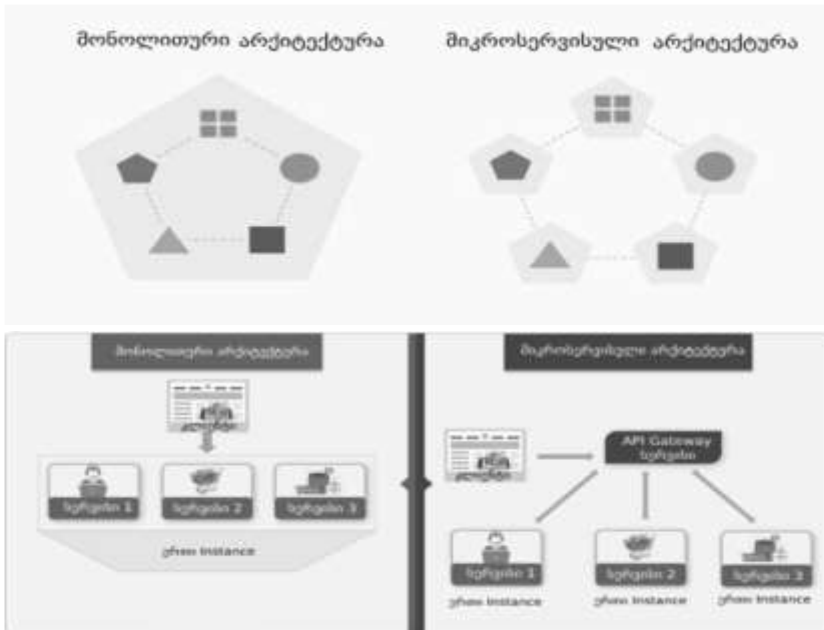
მონაცემთა ერთიანობის და მდგრადობის შენარჩუნება. მონაცემთა მდგრადობის შენარჩუნება მიკროსერვისულ არქიტექტურაში ერთ-ერთი ფუნდამენტური გამოწვევაა და არაერთი მიდგომა გვხვდება, რომელიც ამ პრობლემის მოგვარებაში გვეხმარება.

- დისტრიბუციული ოპერაციის ტრანზაქციულობა: მიკროსერვისულ არქიტექტურაში, ერთი ბიზნესოპერაცია წარმოდგენილია როგორც რამდენიმე სერვისის ჯაჭვური გამოძახება. ამ შემთხვევაში, თუ რომელიმე კონკრეტულ კომპონენტში მოხდება ფუნქციონალური ჩავარდნა, რთულია მონაცემთა საწყის მდგომარეობაში დაბრუნების პროცესის მართვა (Rollback). ამ პრობლემის გადასაჭრელად გვჭირდება *პროცესმენეჯერების* შემოტანა სისტემაში, ან სხვადასხვა კომპონენტის მხრიდან სხვადასხვა მოვლენის თანმიმდევრული დამუშავება (Event Subscription), რომ ოპერაციის მთლიანობის აღქმა არ დავკარგოთ (Saga Pattern) [70].

4.3 ნახაზზე მოცემულია მონოლითური და მიკროსერვისული არქიტექტურის მქონე სისტემების მაგალითები, სადაც კლიენტი აპლიკაციიდან ხდება განსხვავებული არქიტექტურით რეალიზებულ Back End სერვისებზე წვდომა.

მონოლითის შემთხვევაში, რამდენიმე ლოგიკური მოდული დანერგილია როგორც ერთი მთლიანი სერვისი, ხოლო მიკროსერვისების შემთხვევაში, ლოგიკური მოდულები, მცირე ზომის სერვისების სახით გვაქვს წარმოდგენილი, რომელთა შორის კომუნიკაცია უნივერსალური არხების საშუალებით ხდება.

მიკროსერვისის ჭრილში, ვებ სერვისის (Api) გარდა, შესაძლოა რამდენიმე დამატებითი კომპონენტი იყოს დანერგილი, მაგალითად Message Handler Windows Service ან Job Manager Windows Service, რომლებიც მიკროსერვისის ნაწილს წარმოადგენს და მათი არსებობა და სტრუქტურა მთლიანად Solution Architecture გადაწყვეტის ნაწილია.



ნახ. 4.3. მონოლითური და მიკროსერვისული არქიტექტურის მქონე სისტემების მაგალითები

მიკროსერვისული არქიტექტურა, დროთა განმავლობაში, გამოცდილების ზრდასთან ერთად, უფროდაუფრო პოპულარული ხდება და საკმაოდ დიდი გამოწვევა გახდა

ისეთი კორპორაციებისთვის, რომლებმაც ტექნიკური განვითარების დიდი გზა გაიარა და მოთხოვნებიდან გამომდინარე, ბუნებრივად მივიდა სისტემების დისტრიბუციულად განვითარების აუცილებლობამდე. ისეთი კომპანიები, როგორიცაა Netflix, Amazon, Google და ა.შ., ჯერ კიდევ ათეული წლის წინ დადგნენ მონოლითური სისტემების დეკომპოზიციის გამოწვევის წინაშე და დღეს უკვე თამამად გვიზიარებენ გამოცდილებას ამ პროცესის შესახებ.

მსოფლიოში არაერთი მეცნიერი მუშაობს დისტრიბუციული არქიტექტურის დამუშავებაზე და პრაქტიკული გამოცდილებებიდან გამომდინარე, თანამედროვე ტექნიკურ სივრცეებში ყოველდღიურად ვხვდებით სხვადასხვა გადაწყვეტებს თუ რეკომენდაციებს. ჩვენი ნაშრომის ამ ნაწილის ერთ-ერთი მიზანია ჩავატაროთ დისტრიბუციული სისტემების დამუშავების ღრმა ანალიზი და პრაქტიკული მაგალითების საფუძველზე წარმოვაჩინოთ დისტრიბუციული სისტემების აგების სირთულეებისა და უპირატესობების ასპექტები, რომლებიც ამ სფეროში გვხვდება.

თავი 5
მცირე და საშუალო ბიზნესის მართვის ციფრული პლატფორმის არქიტექტურული მოდელი

დღესდღეობით ელექტრონული მართვის სისტემების გარეშე ბიზნესის არსებობა წარმოუდგენელია. კვლევის პროცესში გავაანალიზეთ არსებული გამოწვევები და საჭიროებები მცირე და საშუალო ზომის ბიზნესის განვითარების კუთხით, რის შედეგადაც გამოიკვეთა, რომ მცირე და საშუალო ბიზნესის წარმომადგენლობისათვის განსაკუთრებულად დიდ გამოწვევას წარმოადგენს ბიზნესის ციფრული ტრანსფორმაცია [24,71,72].

ისეთი კომპანიებისათვის, რომელთა ბიზნესიც განვითარების საწყის ეტაპზეა და წარუმატებლობის რისკები გაცილებით მაღალია, სისტემის შემუშავება, დაგეგმვა და ზუსტი მოთხოვნების ჩამოყალიბება რთულ პროცესს წარმოადგენს. ასევე, უნდა აღინიშნოს, რომ საკუთარ მოთხოვნებზე მორგებული სისტემის შექმნისათვის აუცილე-ბელი რესურსები და ტექნოლოგიური კომპეტენცია, მცირე ზომის კომპანიებისათვის ხშირ შემთხვევაში ყველაზე დიდი პრობლემაა.

მცირე და საშუალო ბიზნესის წარმომადგენლობას იშვიათად აქვს რესურსი საკუთარი პროგრამული უზრუნ-ველყოფის დეტალური ბიზნესპროცესების ტექნიკური ანალიზისათვის, სისტემის შექმნა-განვითარება-შენარჩუნე-ბისათვის. შესაბამისად, ყოველდღიურად იზრდება მოთხოვ-ნა ისეთ სერვისებზე, რომელთა საშუალებითაც კომპანიებს

ექნება შესაძლებლობა მართონ საკუთარი ბიზნესი და განახორციელონ პროდუქციის რეალიზაცია.

ჩვენ მიერ წარმოდგენილი გარემოებებიდან გამომდინარე, ნაშრომის ფარგლებში შევიმუშავეთ *მცირე და საშუალო ბიზნესის მართვის ციფრული პლატფორმის Enterprise არქიტექტურა.*

აქ განხილული ციფრული პლატფორმა დისტრიბუციული სისტემის მაგალითია, რომლის საშუალებითაც, კომპანიებს შეუძლია ტექნიკური გამოცდილების გარეშე, შეიძინოს პროდუქციის მართვის და რეალიზების არსებული გადაწყვეტა და საკუთარი მოთხოვნების საფუძველზე მოახდინოს სისტემის მასშტაბირება.

პლატფორმას აქვს მოდულარული სტრუქტურა, მორგებულია სხვადასხვა ზომის/ტიპის კომპანიაზე და მათ შესაძლო მოთხოვნებზე.

5.1. სისტემის სერვისის სახით მიწოდების უპირატესობები

სისტემის მიწოდებას სერვისის სახით აქვს არაერთი უპირატესობა, როგორებიცაა:

- კომპანიას არ უწევს პროგრამული უზრუნველყოფის ტექნიკური ანალიზი და მისი დეველოპმენტი;
- სისტემა მთლიანად განთავსებულია პროვაიდერის (მომწოდებლის) მხარეს და კომპანიას არ ჭირდება არანაირი ინფრასტრუქტურის შექმნა სისტემის დასანერგად/შესანარჩუნებლად;

- პლატფორმის ხელმისაწვდომობა, მონაცემთა დაცულობა და მოთხოვნების მიხედვით სისტემის მასშტაბირება სერვისის მომწოდებლის პასუხისმგებლობაა;
- მცირე ბიზნესს ეძლევა საშუალება, მინიმალური დანახარჯების სანაცვლოდ, საპილოტედ შეიძინოს სისტემის საბაზისო პაკეტი და შედეგებიდან გამომდინარე განსაზღვროს სერვისის გაფართოება;
- მოთხოვნების საფუძველზე დაგროვილი ცოდნის და რეალიზებული ფუნქციონალის საშუალებით, მცირე ბიზნესისათვის მარტივდება სამომავლო განვითარების სტრატეგიის შემუშავება.

პროდუქტის სამომხმარებლო ღირებულება დგინდება პაკეტების მიხედვით. პაკეტი განისაზღვრება კომპანიის ზომის და განხორციელებული ტრანზაქციების საშუალო რაოდენობით, მონაცემთა ბაზის ზომით და ინფრასტრუქტურული (სერვერული) დატვირთვის მიხედვით.

5.2. სისტემის საბაზისო პაკეტის ფუნქციონალის აღწერა

- პროდუქციის მარაგების აღწერა და მართვა;
- პროდუქციის ელექტრონული რეალიზაცია (კომპანიის ბრენდის სპეციფიკაციებზე მორგების მოქნილი შესაძლებლობებით);
- ადგილზე მიტანის სერვისი;
- შეთავაზებების და აქციების მართვის მოქნილი ფუნქციონალი;

- ადმინისტრირების მოდული კომპანიის თანამშრომლე-ბისათვის (პროდუქციის – მარაგების და შეთავაზებების მარ-თვა, შეკვეთების კონტროლი, გადახდების მართვა და ა.შ.);
- რეპორტინგის ფუნქციონალი.

სისტემის არქიტექტურა, ბიზნეს დომეინის კომპლექ-სურობიდან და სწრაფად მზარდი მოთხოვნებიდან გამომდინარე, სასურველია, რომ არ იყოს ცენტრალიზებული.

დისტრიბუციული გადაწყვეტა გვაძლევს საშუალებას, რომ სისტემა განვავითაროთ როგორც დამოუკიდებელი, მარტივად ინტეგრირებადი კომპონენტების ერთობლიობა და მივაღწიოთ მაღალ ხელმისაწვდომობას მომხმარებლისათვის კრიტიკული ფუნქციონალის ფარგლებში [73-76].

ასევე, სისტემის მასშტაბირებადობა დამოკიდებულია მომხმარებელი კომპანიის ზომასა და მოთხოვნებზე. აქედან გამომდინარე, სისტემაში საჭიროა კონკრეტული ფუნქციო-ნალის მასშტაბირების მოქნილი მექანიზმის არსებობა.

პროგრამული უზრუნველყოფა უნდა გამოირჩეოდეს ტესტირების მაღალი ხარისხით [5], გაზრდილი ბიზნეს მოთხოვნების პირობებში, ხშირი დანერგვების დროს, უნდა შეგვეძლოს კონკრეტული მცირე ფუნქციონალის დამოუკი-დებელი განახლება და ეს არ უნდა აყენებდეს ეჭქვეშ სისტემის დანარჩენი კომპონენტების ვერსიის საიმედოობას [5,75, 77-80].

5.3. პოდუქციის ელექტრონული შესყიდვის ბიზნეს პროცესი

განვიხილოთ სისტემის მომხმარებლის (User) მხარეს არსებული ფუნქციონალი (მომხმარებლის პორტალი).

მომხმარებელი შედის კონკრეტული ბრენდის ონლაინ გაყიდვების პორტალზე, სადაც უჩანს აქტიური შეთავაზებების, აქციების და სარეალიზაციო პროდუქციის სია.

პროდუქტის ნახვა მომხმარებელს შეუძლია ავტორიზაციის გარეშე (პორტალზე ხელმისაწვდომია როგორც პროდუქტის დეტალური აღწერა და ფასი, ასევე შესაბამისი მარაგების შესახებ ინფორმაცია).

პროდუქციის შეძენის შემთხვევაში, მომხმარებელი გადის აუთენტიფიკაცია/ავტორიზაციის პროცესს და ამატებს კალათში სასურველ პროდუქტ(ებ)ს, რომლის მარაგებიც იმ ეტაპზე აქტიურია. მას შემდეგ, რაც მომხმარებელი გააკეთებს საკუთარ არჩევანს, აჭერს ღილაკს ,,***შეძენა***“ და გადადის შემდეგ ეტაპზე.

მომხმარებელი ირჩევს სასურველ მისამართს მიტანის სერვისისთვის და ახორციელებს ბარათით გადახდას. გადახდის წარმატებით დასრულების შემთხვევაში სისტემაში ხდება მიტანის სერვისის ჯავშნის გაკეთება და მომხმარებელს ეგზავნება შეტყობინება (Email/Sms Notification) შეკვეთის სტატუსის და პროდუქტის დეტალების შესახებ.

მომხმარებელს შესყიდულ პროდუქციას მითითებულ მისამართზე კომპანია თავად მიაწვდის.

მას შემდეგ, რაც მოხდება პროდუქტის მომხმარე-ბელთან მიტანა, ხდება შეკვეთის სტატუსის ცვლილება როგორც ,,***დასრულებული***".

5.3.1. ონლაინ გაყიდვების ბიზნესპროცესში მონაწილე მიკროსერვისები

ონლაინ გაყიდვების ბიზნეს პროცესში მონაწილე მიკროსერვისები წარმოდგენილია შემდეგნაირად:

1) *მომხმარებლების მართვის მიკროსერვისი* (Identity Service) – მოიცავს მომხმარებლების და მათი უფლებების ინფორმაციას. სერვისში გაერთიანებულია მომხმარებლების რეგისტრაციის, აუთენტიფიკაცია/ავტორიზაციის, დაბლოკვა /განბლოკვის და მომხმარებელთა უფლებების მართვის ფუნქციონალი;

2) *შეკვეთების მიკროსერვისი* (Order service) – პროდუქ-ციის შეკვეთის განთავსების/გაუქმების ფუნქციონალი. შეკვეთის შესახებ ინფორმაციის დეტალების ძირითადი სანახი სივრცე (Master);

3) *გადახდების მიკროსერვისი* (Payment Service) – გადახ-დის დამუშავების და გადახდების მენეჯმენტის ძირითადი ფუნქციონალი. გადახდების ინფორმაციის ძირითადი სანახი სივრცე (Master);

4) *მარაგების მართვის მიკროსერვისი* (Inventory Service) – მარაგების აღრიცხვის, განახლების ფუნქციონალი. პროდუქციის მარაგის შესახებ ინფორმაციის ძირითადი სანახი სივრცე (Master);

5) *ადგილზე მიტანის მომსახურების მიკროსერვისი* (Delivery Service) – მოიცავს პროდუქციის ადგილზე მიტანის სერვისის მართვის ფუნქციონალს;

6) *შეტყობინებების მიკროსერვისი* (Notification Service) – მომხმარებლებთან, თანამშრომლებთან, კომპანიის ბიზნეს წარმომადგენლობასთან შეტყობინებების (eMail/SMS) გაგზავნის და მართვის ძირითადი ფუნქციონალი;

7) *პროდუქტების მიკროსერვისი* (Products Service) – პროდუქციის მართვის ფუნქციონალი;

8) შეთავაზებების მიკროსერვისი (Promotion Service) – მომხმარებლის შეთავაზებების / აქციების მართვის ფუნქციონალი.

5.3.2. პროდუქციის ელექტრონულად შესყიდვის ბიზნესპროცესის ბიჯები

პროდუქციის ელექტრონულად შესყიდვის ბიზნეს პროცესის ბიჯები გამოიყურება შემდეგნაირად:

1) შეკვეთის განთავსება დასამუშავებლად (შეკვეთების მიკროსერვისი – Order service);

2) გადახდის ინფორმაციის დამუშავება (გადახდების მიკროსერვისი – Payment Service);

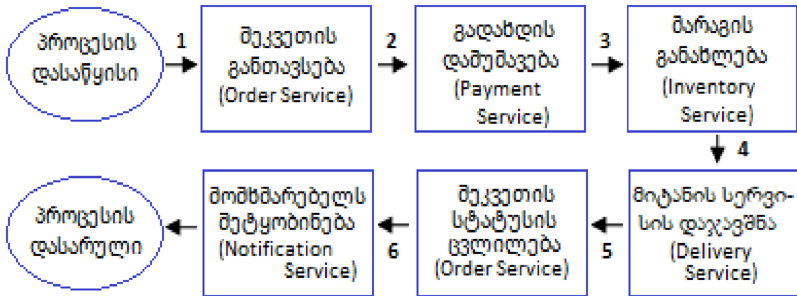
3) მარაგის განახლება (მარაგების მიკროსერვისი – Inventory Service);

4) მიტანის სერვისის დაჯავშნა (Delivery Service);

5) შეკვეთის სტატუსის ცვლილება მნიშვნელობით ,,მზად არის მისაწოდებლად“;

6) მომხმარებლისთვის შეკვეთის დეტალების შეტყობი-ნების გაგზავნა (Notification Service).

5.1 ნახაზზე წარმოდგენილია პროდუქციის შესყიდვის ბიზნეს პროცესის სქემა.



ნახ. 5.1. პროდუქციის ელექტრონულად შესყიდვის ბიზნეს ოპერაცია მიკროსერვისული არქიტექტურის ბაზაზე

5.4. სისტემის ზოგადი ტექნიკური აღწერა

თითოეული კლიენტი კომპანიისათვის ხდება შესაბა-მისი ვებ-აპლიკაციის განთავსება ინტერნეტ სივრცეში, რომელიც არის ძირითადი სისტემის ქვედომენი (Subdomain) და რომლის მისამართსაც იყენებს ბიზნეს წარმომადგენლობა საკუთარი გაყიდვების რეკლამირებისათვის.

თითოეული ბრენდის პორტალზე გადამისამართება ასევე შესაძლებელია სისტემის ძირითადი მისამართიდან. სურვილისამებრ, შესაძლებელია კომპანიის უკვე ხელთ არსებული მისამართის გამოყენებაც.

სერვისის საბაზისო პაკეტით სარგებლობისას, კომპანიას შეუძლია სტანდარტული შესყიდვების პორტალისათვის საკუთარი ბრენდინგის შესაბამისი დიზაინის შექმნა და პორტალის ვიზუალური სახეცვლილება (კონკრეტული ჩარჩოების ფარგლებში).

ვებ აპლიკაცია შექმნილია ისეთი პრინციპებით, რომ კომპანიის პორტალის ძირითადი ელემენტების მოდიფიცირება, ლოგოს სტილების და ვიზუალური დეტალების კონფიგურირება მარტივად სამართავი იყოს ახალი კომპანიის რეგისტრაციისას და არ დაჭირდეს დამატებითი, კრიტიკული ზომის დეველოპმენტი.

რაც შეეხება შეთავაზებების და აქციების ვიზუალს, მათი კონტროლი ხდება დინამიკურად, ადმინისტრირების მოდულიდან თითოეული კომპანიის ჭრილში, კომპანიის თანამშრომლის მიერ. თანამშრომელს ადმინისტრირების მოდულში უჩანს html დინამიკური ტექსტ-ედიტორი, რომელიც საშუალებას აძლევს არატექნიკური კომპეტენციის მქონე პირს, „drug and drop“ (მაუსით გადატანა) პრინციპით ააწყოს, სასურველი ვიზუალის მქონე შეთავაზება და შეინახოს სისტემაში. არსებული შეთავაზება, პორტალზე გამზადებული სახით გამოუჩნდება შესაბამის მომხმარებელს.

იმ შემთხვევაში, თუ ბიზნესის წარმომადგენელი კომპანიის სასურველი პორტალის ვიზუალური ნაწილი რადიკალურად განსხვავდება არსებული შეთავაზებული ვიზუალური შაბლონისაგან და სერვისის პროვაიდერის მხრიდან უზრუნველყოფილი მოდიფიკაციის საშუალებები

არ აკმაყოფილებს მომხმარებელი კომპანიის სასურველ ვიზუალურ მხარეს, ბიზნესის წარმომადგენელს აქვს საშუალება გამოიყენოს არსებული Back End გადაწყვეტა და მხოლოდ პორტალის ნაწილი მოაწყოს საკუთარი სურვილისამებრ (დამოუკიდებლად), სისტემის ფუნქციონალური ნაწილი კი გამოიყენოს უცვლელად.

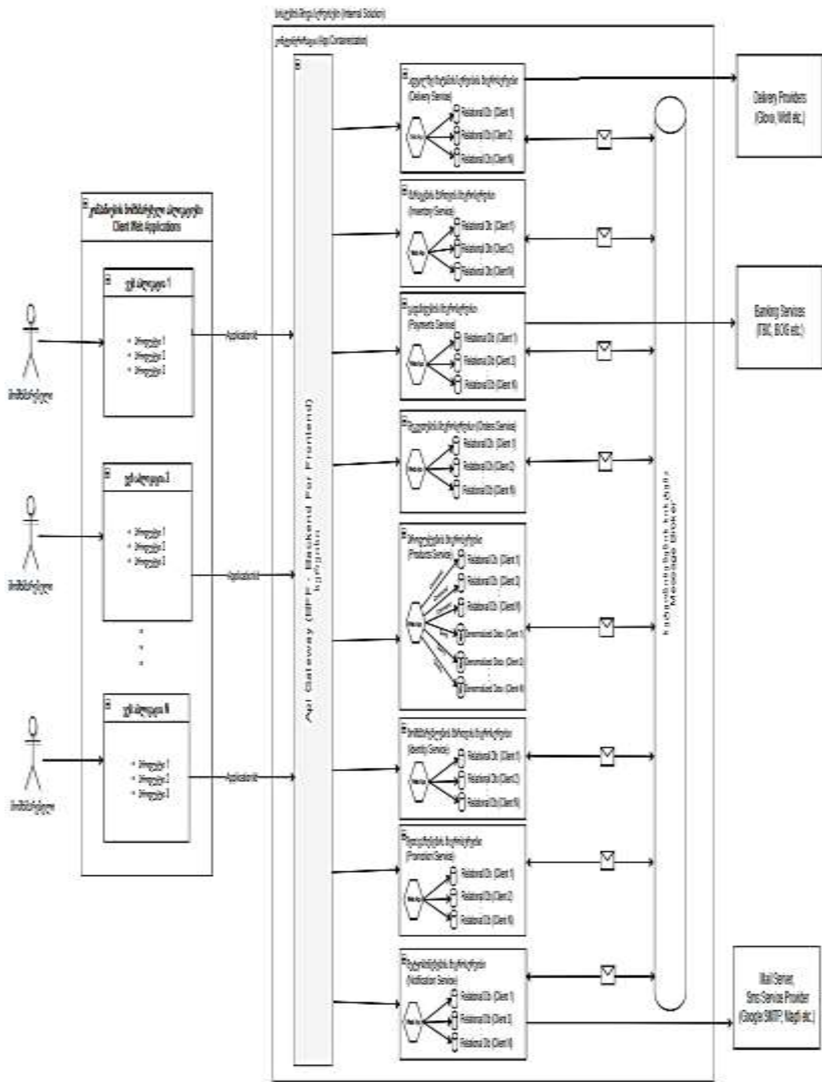
კლიენტი ვებ-აპლიკაცია არის Single Page Application (SPA) და Back End სერვისებთან კომუნიკაციას ახორციელებს Api Gateway (Backend For Frontend – BFF) სერვისული შრის გავლით.

დისტრიბუციულ სისტემებში, კლასიკური სინქრონული კომუნიკაციის ნაცვლად, *ასინქრონული კომუნიკაცია* ხშირად დიდ უპირატესობას იძლევა (Message/Event Driven Architecture – შეტყობინებაზე/მოვლენაზე ორიენტირებული არქიტექტურა).

მიკროსერვისები თავის მხრივ ახდენს ასინქრონულ კომუნიკაციას ერთმანეთთან შეტყობინებების სისტემის (Message Broker) მეშვეობით (შესაძლო ტექნოლოგიებია – RabbitMq, Kafka და სხვ.) [81,82].

5.5. სისტემის მომხმარებლის პორტალის არქიტექტურული მოდელი

ვებ-პორტალის არქიტექტურის დიაგრამა გამოიყენება მომხმარებლის ვებ-პორტალის საერთო სტრუქტურის აღსაწერად. 5.2 ნახაზზე ნაჩვენებია ერთი ნიმუშის ფრაგმენტი მიკროსერვისების მაგალითით.



ნახ. 5.2. მცირე და საშუალო ბიზნესის მართვის სისტემის მომხმარებლის პორტალის არქიტექტურული მოდელი

5.6. დისტრიბუციული ტრანზაქციის მართვა

დისტრიბუციულ სისტემაში კონკრეტული ბიზნეს ოპერაცია წარმოდგენილია როგორც რამდენიმე სერვისის ფუნქციონალის თანმიმდევრული (იშვიათ შემთხვევებში პარალელური) შესრულება. ოპერაცია წარმატებულია, თუ ყველა გამოძახება წარმატებულია *(დისტრიბუციული ტრანზაქცია).*

პროდუქციის შესყიდვის ოპერაცია არის დისტრიბუციული პროცესი. თუ დადგება სისტემაში მდგომარეობა, როდესაც შეკვეთის განთავსება წარმატებით დასრულდა, ხოლო გადახდის პროცესი ვერ შესრულდა ამათუიმ მიზეზით, მთლიანად ოპერაცია უნდა ჩაითვალოს წარუმატებლად და მონაცემთა მდგომარეობა უნდა დაბრუნდეს საწყის პოზიციას.

დისტრიბუციულ სისტემებში ასინქრონული კომუნიკაცია მრავალ უპირატესობას იძლევა, სისტემის კომპონენტების დამოუკიდებლობის და სისტემის ხელმისაწვდომობის თვალსაზრისით.

მაგალითად, განვიხილოთ ჩვენს მიერ დაპროექტებულ სისტემაში, *შეკვეთის განთავსების* ბიზნესპროცესი. შეკვეთების ფუნქციონალი უნდა გამოირჩეოდეს მაღალი ხელმისაწვდომობით, შესაბამისად სისტემის კომპონენტები არ უნდა იყოს ერთმანეთთან მჭიდროდ დაკავშირებული (loosely coupling). მაგალითად, *მარაგების მართვის* სერვისის დროებითი შეფერხების შემთხვევაში (აპლიკაციის ხარვეზის,

ქსელის გაუმართაობის, თუ ახალი ვერსიის დანერგვის შემთხვევაში) არ უნდა მოხდეს შეკვეთების ფუნქციონალის შეფერხება. ასევე, UI/UX კუთხით შეკვეთის განთავსება უნდა გამოირჩეოდეს ოპტიმალური დამუშავების დროით.

ზემოთ ჩამოთვლილი მიზეზების გამო, სერვისებს შორის კომუნიკაცია ნაცვლად სინქრონული გადაწყვეტისა, უმჯობესია განხორციელდეს *ასინქრონულად.*

ასინქრონული კომუნიკაციის მაგალითში, მარაგების მართვის სერვისის დროებითი შეფერხების შემთხვევაში, სისტემა მაინც იღებს შეკვეთას დასამუშავებლად და პროცესი გაგრძელდება მარაგების სერვისის ამუშავებისთანავე.

ტექნიკურად პროცესი გამოიყურება შემდეგნაირად: შეკვეთის განთავსების დროს, მარაგების სერვისის (Api) სინქრონულად გამოძახების ნაცვლად, შეკვეთის განხორციელების შესახებ შეტყობინება თავსდება შეტყობინებათა (Message Broker) სისტემის რიგში (Queue) და ამ მესიჯს ასინქრონულად ამუშავებს მარაგების მიკროსერვისი (Inventory Service).

შედეგად, მომხმარებელი არეგისტრირებს შეკვეთას, პორტალი აწვდის ინფორმაციას, რომ შეკვეთა მიღებულია და მუშავდება, მას შემდეგ რაც დასრულდება backend პროცესები და შეკვეთა გაივლის ყველა აუცილებელ ფუნქციონალურ ეტაპს შესაბამის სერვისებში, მომხმარებელს UI-ზე Live რეჟიმში გამოუჩნდება შეკვეთის სტატუსი და მიუვა (eMail/Sms) შეტყობინება (Live რეჟიმში განახლების ნაცვლად, შესაძლებელია Refresh ფუნქციის გამოყენებაც).

სისტემის დისტრიბუციულობიდან გამომდინარე, ტრანზაქციულობის მართვა ვერ ხერხდება მონაცემთა სანახის (საცავის, ბაზის) ან კონკრეტული სერვისის ბიზნეს ლოგიკის დონეზე. შესაბამისად, საჭირო ხდება *სერვისებს შორის დისტრიბუციული პროცესების მენეჯმენტის და ტრანზაქციულობის მართვის სტრატეგიის* შემოღება.

იმ შემთხვევაში თუ დისტრიბუციული ტრანზაქციის შესრულების პროცესში რომელიმე კონკრეტულ კომპონენ-ტში მოხდება ფუნქციონალური ან ინფრასტრუქტული შეფერხება, ან თუნდაც არავალიდური ბიზნესპროცესის გამო მოხდება პროცესის შუაში გაწყვეტა, აუცილებელია წინა ლოკალური ოპერაციებით შეცვლილი მონაცემების საწყის მდგომარეობაში დაბრუნება (კომპენსაცია – Distributed Trans-action Compensation) მთლიანი სისტემის დონეზე, რათა შევინარჩუნოთ მონაცემთა მთლიანობა.

დისტრიბუციული ტრანზაქციის მართვაში გვეხმარება დისტრიბუციული სისტემების ერთ-ერთი არქიტექტურული მიდგომა - ,,Saga Pattern“. *Saga* არის *ლოკალურ ტრანზაქციათა (ბიზნესოპერაციათა) მიმდევრობა და მათი მართვის მექა-ნიზმი* [65, 83]

პრაქტიკაში გვხვდება 2 ტიპის Saga: ქორეოგრაფიაზე დაფუძნებული საგა (Choreography-based saga) და ორკესტრირებაზე დაფუძნებული საგა (Orchestration-based saga) [65].

განვიხილოთ ეს საკითხი უფრო დეტალურად.

5.6.1. ქორეორგაფიაზე დაფუძნებული საგა

ქორეოგრაფიაზე დაფუძნებული საგა (Choreography-based saga) არის ასინქრონული კომუნიკაციის მაგალითი (publish/subscribe პრინციპი). თითოეული სერვისი გამოძახებისას ასრულებს კონკრეტულ, ბიზნესისთვის სპეციფიურ ლოკალურ ოპერაციას და აქვეყნებს ინფორმაციას მოვლენის მოხდენის შესახებ (Publish Event) message broker სისტემაში, ხოლო სხვა მიკროსერვისები, თავის მხრივ, ამუშავებს ამ მოვლენებს (Subscribe Event) და განაგრძობს საკუთარი ბიზნეს პროცესების შესრულებას.

ქორეოგრაფიულ საგაში, მიდგომის მიხედვით, არ არსებობს ცენტრალიზებული კოორდინაციის ფუნქციონალი, სერვისებს არ აქვს ინფორმაცია, თუ რა ქმედება უნდა მოყვეს მათ მიერ გამოქვეყნებულ მოვლენას (Event).

ქორეოგრაფიაზე დაფუძნებული საგა სიმარტივის უპირატესობას იძლევა არა მასშტაბური სისტემებისთვის, სადაც კომპონენტების და ლოკალური ტრანზაქციების რაოდენობა მცირეა.

სისტემის ზრდასთან ერთად, მიკროსერვიების კომუნიკაციის პროცესი იღებს ქაოტურ სახეს, რთულდება როგორც დეველოპმენტი და შეცდომის არეალის ლოკალიზება, ასევე ურთიერთდამოკიდებული პროცესების ანალიზი, შედეგად სისტემა ხდება რთულად სამართავი [84, 85].

5.6.2. ორკესტრირებაზე დაფუძნებული საგა

ორკესტრირებაზე დაფუძნებული საგა (Orchestration-based saga) არის ბიზნესპროცესის ორკესტრირების ცენტრალიზებული იმპლემენტაცია. პროცესის კოორდინაცია ხდება ორკესტრატორების (Process Manager, Saga Execution Coordinator - SEC) მიერ, კონკრეტულ მიკროსერვისებში ბრძანებების (Commands) მიმდევრობით შესრულების საფუძველზე [72,76].

ქორეოგრაფიაზე დაფუძნებული საგასგან განსხვავებით, ორკესტრირებაზე დაფუძნებულ საგაში უშუალოდ მიკროსერვისების ბიზნესური ქცევა არ არის დამოკიდებული სხვა სერვისების მიერ გამოქვეყნებულ მოვლენებზე.

მიკროსერვისები აქვეყნებს ინფორმაციას საკუთარი ბიზნესოპერაციის მოვლენის მოხდენის შესახებ (State Mutation Event – ინფორმაცია მონაცემის მდგომარეობის ცვლილების შესახებ), პროცესის მენეჯერი ახდენს ამ მოვლენების დამუშავებას და აგრძელებს პროცესს, სხვა სერვისების გამოძახების ხარჯზე.

პროცესის მენეჯერებს აქვთ საკუთარი მონაცემთა საცავი, სადაც ყველა დისტრიბუციული ტრანზაქციისათვის ინახავენ ინფორმაციას, თუ რომელი ლოკალური ოპერაცია დასრულდა წარმატებით, ან რა ეტაპზე იმყოფება მიმდინარე დისტრიბუციული ტრანზაქცია.

იმ შემთხვევაში, თუ მოხდება ინფრასტრუქტურული ან ლოგიკური შეფერხება, პროცესის მენეჯერები იწყებენ ტრანზაქციის კომპენსირების პროცესს, რაც გულისხმობს, კონკრეტულ სერვისებში კომპენსაციის პროცესების შესაბამისი

მიმდევრობით ინიცირებას, რათა სისტემის დონეზე აღდგეს მონაცემთა საწყისი მდგომარეობა.

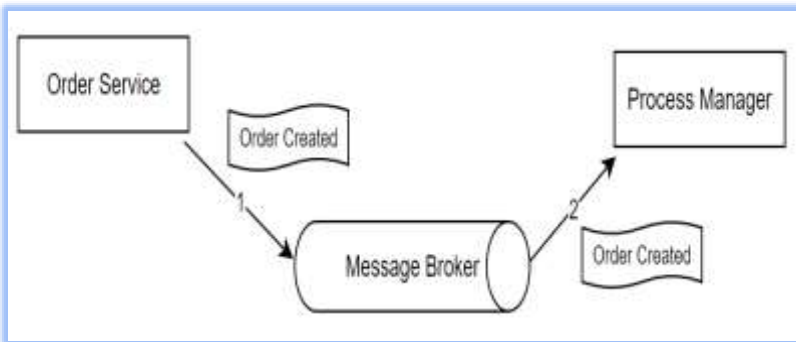
ორკესტრირებაზე დაფუძნებული საგა კომპლექსურ სისტემებში იძლევა ბიზნეს ლოგიკის მართვის იზოლირების საშუალებას, რაც თავის მხრივ აგვარებს სერვისებს შორის ურთიერთდამოკიდებულების პრობლემას. მიკროსერვისებს არ აქვს ინფორმაცია ერთმანეთის ქცევის შესახებ.

პროცესის მენეჯერების საშუალებით მარტივია ბიზნეს პროცესის ანალიზი და ცვლილება, რადგან სრული კონტროლი გაგვაჩნია კოდის ერთ კონკრეტულ წერტილში. მეორეს მხრივ, კოორდინატორების შემოტანა სისტემაში დეველოპმენტის პროცესს ხდის გაცილებით კომპლექსურს და სისტემაში შემოაქვს ხელმისაწვდომობის კუთხით კრიტიკული კომპონენტი, რომლის შეფერხებაც, გამოიწვევს მთლიანი სისტემის შეფერხებას (Single Point Of Failure - SPOF) [84, 85].

კომპლექსურობის გათვალისწინებით, იქიდან გამომდინარე, რომ ჩვენ სისტემაში წარმოდგენილი სერვისების და ლოკალური ტრანზაქციების რაოდენობა არ არის მცირე, ასევე მოსალოდნელია სისტემის ზრდა და ბიზნესფუნქციონალის ხშირი ცვალებადობა, დისტრიბუციული ტრანზაქციის მართვის მეთოდად ავირჩიეთ ორკესტრირებაზე დაფუძნებული საგა (Orchestration-Based Saga).

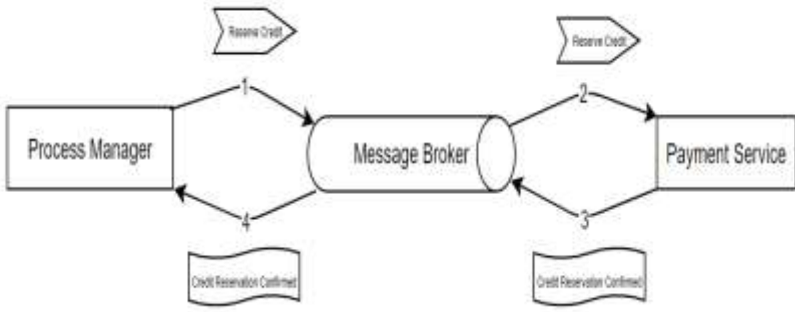
ორკესტრირებაზე დაფუძნებული საგას მაგალითზე, 5.3-5.8 ნახაზებზე წარმოდგენილია პროდუქციის ელექტრონული შესყიდვის დისტრიბუციული ტრანზაქციის მართვის მექანიზმი:

ბიჯი 1: Order Service ახდენს მომხმარებლის შეკვეთის მიღებას, ქმნის შეკვეთას დასამუშავებელი სტატუსით და აქვეყნებს Order Created მოვლენას (event) message broker საკომუნიკაციო არხში. პროცესის მენეჯერი (Orchestrator) - ახდენს Order Created მოვლენის (event) დამუშავებას და აღძრავს დისტრიბუციული ტრანზაქციის მართვის პროცესს, ბრძანებების (command) თანმიმდევრულად შესრულების ხარჯზე.



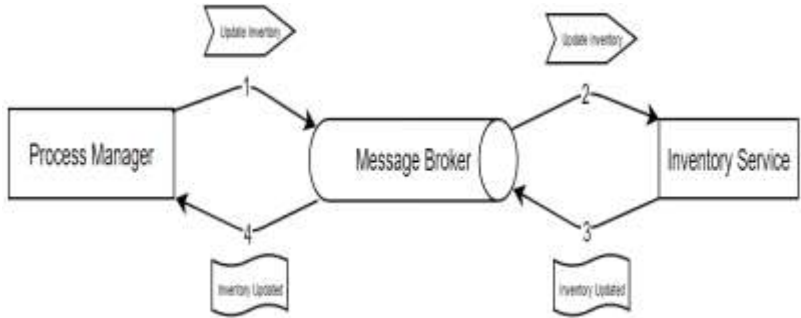
ნახ. 5.3

ბიჯი 2: პროცესის მენეჯერი (Orchestrator) წარმოშობს Reserve Credit ბრძანებას (command) Payment სერვისში, Payment Service ბრძანების შესრულების შემდეგ აქვეყნებს Credit Reservation Confirmed მოვლენის წარმატებით/წარუმატებლად დასრულების ინფორმაციას (event) message broker საკომუნიკაცო არხში. პროცესის მენეჯერი ამუშავებს სერვისის გამოქვეყნებულ მოვლენებს და ბრძანებების შესრულების სტატუსის მიხედვით ახდენს ბიზნეს პროცესის გაგრძელებას ან კომპენსაციის ლოგიკის ინიცირებას.



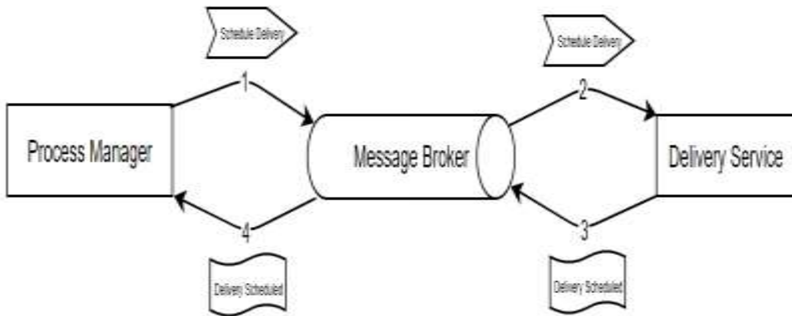
ნახ. 5.4

ბიჯი 3: პროცესის მენეჯერი (Orchestrator) წარმოშობს Update Inventory ბრძანებას (command) Inventory სერვისში, Inventory Service ბრძანების შესრულების შემდეგ აქვეყნებს Inventory Updated მოვლენის წარმატებით/წარუმატებლად დასრულების ინფორმაციას (event) message broker საკო-მუნიკაცო არხში, პროცესის მენეჯერი ამუშავებს სერვისის გამოქვეყნებულ მოვლენებს და ბრძანებების შესრულების სტატუსის მიხედვით ახდენს ბიზნესპროცესის გაგრძელებას ან კომპენსაციის ლოგიკის ინიცირებას.



ნახ. 5.5

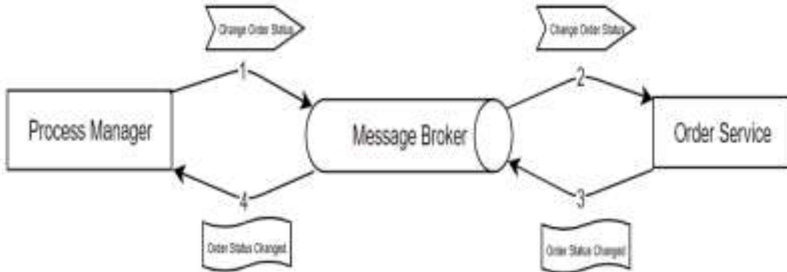
ბიჯი 4: პროცესის მენეჯერი (Orchestrator) წარმოშობს Schedule Delivery ბრძანებას (command) Delivery სერვისში, Delivery Service ბრძანების შესრულების შემდეგ აქვეყნებს Delivery Scheduled მოვლენის წარმატებით/წარუმატებლად დასრულების ინფორმაციას (event) message broker საკომუნიკაცო არხში, პროცესის მენეჯერი ამუშავებს სერვისის გამოქვეყნებულ მოვლენებს და ბრძანებების შესრულების სტატუსის მიხედვით ახდენს ბიზნეს პროცესის გაგრძელებას ან კომპენსაციის ლოგიკის ინიცირებას.



ნახ. 5.6

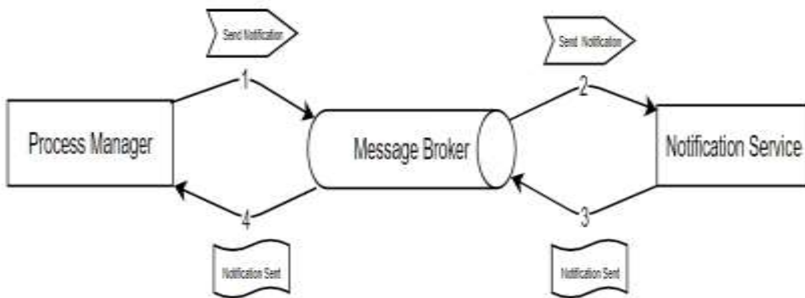
ბიჯი 5: პროცესის მენეჯერი (Orchestrator) წარმოშობს Change Order Status ბრძანებას (command) Order სერვისში, Order Service ბრძანების შესრულების შემდეგ აქვეყნებს Order Status Changed მოვლენის წარმატებით/წარუმატებლად დასრულების ინფორმაციას (event) message broker საკომუნიკაცო არხში. პროცესის მენეჯერი ამუშავებს სერვისის გამოქვეყნებულ მოვლენებს და ბრძანებების შესრულების

სტატუსის მიხედვით ახდენს ბიზნეს პროცესის გაგრძელებას ან კომპენსაციის ლოგიკის ინიცირებას.



ნახ. 5.7

ბიჯი 6: პროცესის მენეჯერი (Orchestrator) წარმოშობს Send Notification ბრძანებას (command) Notification სერვისში, Notification Service ბრძანების შესრულების შემდეგ აქვეყნებს Notification Sent მოვლენის წარმატებით/წარუმატებლად დასრულების ინფორმაციას (event) message broker საკომუ-ნიკაცო არხში, პროცესის მენეჯერი ამუშავებს სერვისის გამოქვეყნებულ მოვლენებს და ახდენს დისტრიბუციული ტრანზაქციის დასრულებას.



ნახ. 5.8

5.6.3. დისტრიბუციული ტრანზაქციის კომპენსაცია (Eventual Consistency)

თუ დისტრიბუციული ტრანზაქციის შესრულების რომელიმე ეტაპზე მოხდება პროცესის შეწყვეტა, უნდა მოხდეს *კომპენსაციის პროცესის* (Distributed Transaction Compensation, Backward Recovery) შესრულება, რაც გულისხმობს ტრანზაქციის შესაბამისი მაკომპენსირებელი ფუნქციონალის (ბიჯების) აღძვრას მიკროსერვისებში.

ისევე, როგორც დისტრიბუციული ტრანზაქცია, კომპენსაციის ოპერაციაც დისტრიბუციულია და გულისხმობს, სხვადასხვა მიკროსერვისში კონკრეტული მონაცემის ვალიდურ მდგომარეობაში დაბრუნებას (Backward Recovery, Rollback). Backward Recovery-ის ყველა ლოკალური ტრანზაქციის წარმატებით დასრულების შემდგომ, შეგვიძლია ჩავთვალოთ, რომ სისტემის დონეზე ოპერაციის დისტრიბუციული ტრანზაქციულობა შენარჩუნებულია.

იმისათვის, რომ შეგვეძლოს რომელიმე მონაცემის ძველი მდგომარეობის აღდგენა, საჭიროა სერვისის ისეთი მეთოდებისთვის, რომლებიც იწვევს მონაცემთა მდგომარეობის ცვლილებას (State Mutation) (T1, T2 … Tn), გაგვაჩნდეს კომპენსაციის შესაბამისი იდემპოტენტური ფუნქციონალი (Retryable Compensation Steps) (C1, C2 … Cn) [86, 87].

ისეთი ლოკალური ტრანზაქციებისთვის, რომელთა კომპენსაცია ფუნქციონალიდან გამომდინარე შეუძლებელია მხოლოდ ბაზის დონეზე (მაგალითად, მომხმარებელთან მეილის გაგზავნის ფუნქციონალი), უნდა არსებობდეს

მაკომპენსირებელი სპეციფიკური ფუნქციონალი (ახალი მეილის გაგზავნა მომხმარებლისათვის, სადაც წინა ნოტიფიკაციის პრობლემურობა იქნება აღწერილი).

თუ კონკრეტულ მომენტში ვერ ხერხდება კომპენსაციის ბიჯის წარმატებით დასრულება, მისი გამოძახება ხდება გარკვეული პერიოდის შემდეგ *retry* პრინციპით იქამდე, სანამ არ მივიღებთ წარმატებულ შედეგს.

მონაცემთა ავტონომიურობიდან გამომდინარე, კომპენსაციის პროცესში, ხშირად შესაძლებელია მაკომპენსირებელი ბიჯების *პარალელურად აღძვრა*, რაც საშუალებას გვაძლევს გაცილებით სწრაფად მოხდეს მონაცემთა მთლიანობის აღდგენა სისტემაში, ვიდრე ბიზნეს ოპერაციის შებრუნებული თანმიმდევრობით შესრულებისას მიიღწეოდა.

ბიზნესოპერაციის და კომპენსაციის დისტრიბუციული პროცესის მართვა კომპლექსური ამოცანაა დეველოპერებისათვის. პროცესმენეჯერებში რეკომენდებულია გამოყენებულ იქნას ისეთი ბიბლიოთეკები, რომლებიც გვთავაზობს პროცესის workflow-ების ოპტიმალური დიზაინის და დეველოპმენტის საშუალებას. მათ აქვს მოქნილი ინსტრუმენტები ჯაჭვური პროცესების სამართავად (მაგალითად Workflow Core .Net ის შემთხვევაში).

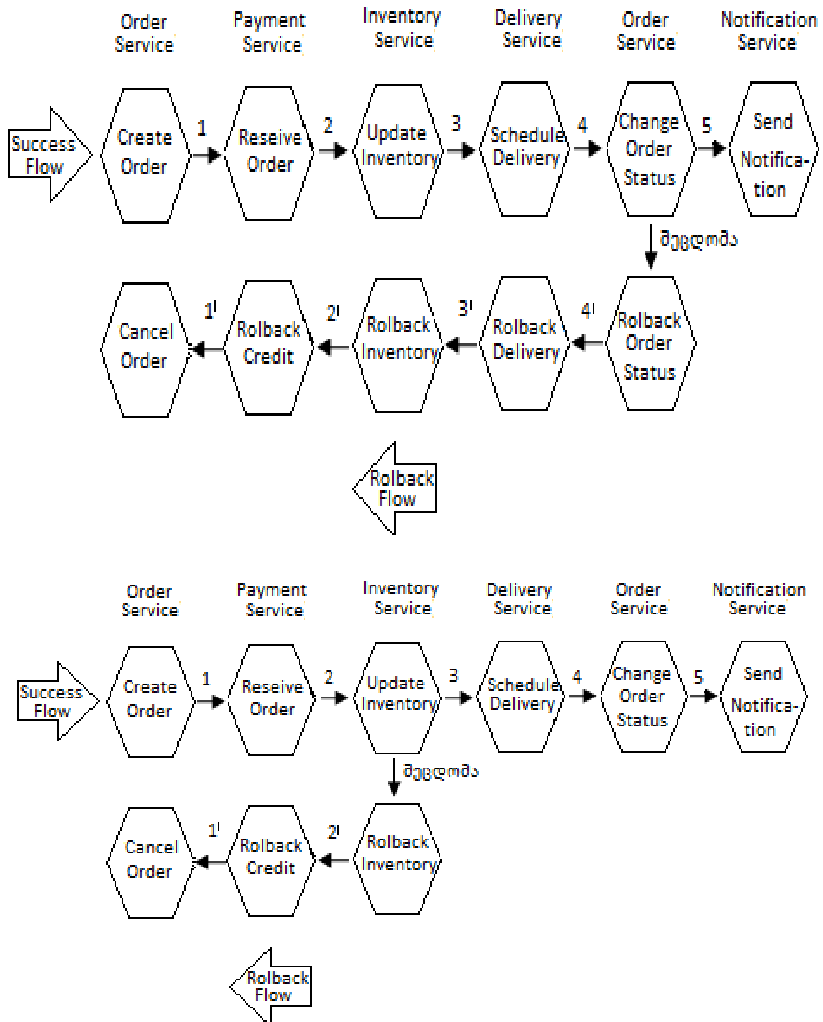
კოორდინატორებს აქვთ საკუთარი მონაცემთა ლოგი, რომელშიც ხდება თითოეული ლოკალური ტრანზაქციის მდგომარეობის ლოგირება (აქედან გამომდინარე, პროცეს მენეჯერის ავარიული გათიშვა ან გადატვირთვა, არ იწვევს დისტრიბუციული პროცესის გაწყვეტას ან ინფორმაციის

დაკარგავს). იმ შემთხვევაში თუ პროცესმენეჯერში მოხდება რაიმე სახის ინფრასტრუქტურული ჩავარდნა, პრობლემის აღმოფხვრის შემდგომ, წინა იტერაციის მონაცემებზე დაყრდნობით, პროცესი გრძელდება იმ წერტილიდან, სადაც დაფიქსირდა წყვეტა. პროდუქციის ელექტრონულად შესყი-დვის დისტრიბუციული ტრანზაქციის კომპენსაციის პროცესი გამოიყურება შემდეგნაირად (ნახ.5.9):

მაგალითად, *შეკვეთის განთავსება, გადახდა, მარაგის რეზერვირება და მიტანის სერვისის დაჯავშნა* წარმატებით დასრულდა ხოლო *შეკვეთის სტატუსის ცვლილება* ვერ ხერხდება რამდენიმე ცდის შედეგად პროგრამული ხარვეზის გამო. ამ შემთხვევაში, იმისათვის, რომ არ გამოვიწვიოთ მომხმარებლის უკმაყოფილება და ასევე შევინარჩუნოთ მონაცემთა მდგრადობა სისტემაში, ხდება ოპერაციის ჭრილ-ში შესრულებული ბიჯების კომპენსაცია.

5.7. მეხუთე თავის დასკვნა

მცირე და საშუალო ბიზნესის მართვის სისტემები-სათვის შემუშავებულია ციფრული პლატფორმის დისტრი-ბუციული არქიტექტურა. იგი დამოუკიდებელი, მარტივად ინტეგრირებადი კომპონენტების ერთობლიობაა და იძლევა მაღალ ხელმისაწვდომობას მომხმარებლებისათვის. კომპა-ნიებს შეუძლია ტექნიკური გამოცდილების გარეშე, შეიძინოს პროდუქციის მართვის და რეალიზების არსებული გადაწყვეტა და საკუთარი მოთხოვნების საფუძველზე მოახდინოს სისტემის მასშტაბირება.



ნახ. 5.9. პროდუქციის ელექტრონულად შესყიდვის ბიზნეს ოპერაცის დისტრიბუციული ტრანზაქციის კომპენსაცია

თავი 6
დისტრიბუციული სისტემის მიკროსერვისები და მათი შემადგენელი კომპონენტები

6.1. მიკროსერვისების ტექნიკური აღწერა

თითოეული მიკროსერვისი, ისევე როგორც Api Gateway სერვისი, წარმოადგენს ვებ-სერვისს (Restful Web Api) და აქვს საკუთარი დოკუმენტაცია OpenApi სპეციფიკაციების გათვალისწინებით, რაც ამარტივებს სისტემის ანალიზს (ადამიანისათვის სისტემის ფუნქციონალი მარტივად გარჩევადია) და ინტეგრაციის პროცესს, ასევე ქმნის შესაძლებლობას მარტივად დავაგენერიროთ http კლიენტი (სხვადასხვა ტექნოლოგიაზე) მომკითხავი სისტემებისთვის.

ვებ-სერვისებს აქვს ვერსიულობის (Api Versioning) მხარდაჭერა, რათა გაზრდილი მოთხოვნების პირობებში, სერვისების ხშირი განახლებისას შევძლოთ უკუთავსებადობის შენარჩუნება (Backward Compatibility).

ინფრასტრუქტურულად, სერვისები დანერგილია კონტეინერების სახით (Docker) და გაზრდილი დატვირთულობის პირობებში, ავტომატურად ხდება სერვისების ჰორიზონტალური მასშტაბირება (სერვისის ინსტანსების რაოდენობის გაზრდა). შესაძლო ტექნოლოგიები: Kubernetes, Openshift etc. [94-96].

სისტემის უსაფრთხოების კუთხით, დეველოპმენტი ხორციელდება ,,OWASP Top Ten" მიდგომების სრული დაცვით და ,,უსაფრთხო კოდის წერის" სხვა გავრცელებული პრაქტიკების გათვალისწინებით [27].

სერვისების მონიტორინგისთვის, გარკვეული პერიოდულობით, მუდმივად ეშვება Health Check ფუნქციონალი, რომელიც უზრუნველყოფს სერვისის სიცოცხლისუნარიანობის შემოწმებას. Health Check პროცესში მოწმდება, რამდენად გამართულად მუშაობს ესათუის სერვისი. Health Check მოიცავს როგორც სერვისის და მისი მონაცემების ან კონკრეტული სპეციფიური ბიზნესლოგიკის მქონე შემოწმების ინიცირებას, ასევე ინფრასტრუქტურული ხელმისაწვდომობის შემოწმებასაც (წვდომა მონაცემთა ბაზაზე, წვდომა Message Broker-ზე, წვდომა ქეშზე და ა.შ.).

ყველა სერვისს აქვს Health Route, რომლის გამოძახებით, მონიტორინგის სისტემებს შეუძლია დაადგინოს სერვისის სიცოცხლისუნარიანობის შესაბამისი სტატუსი.

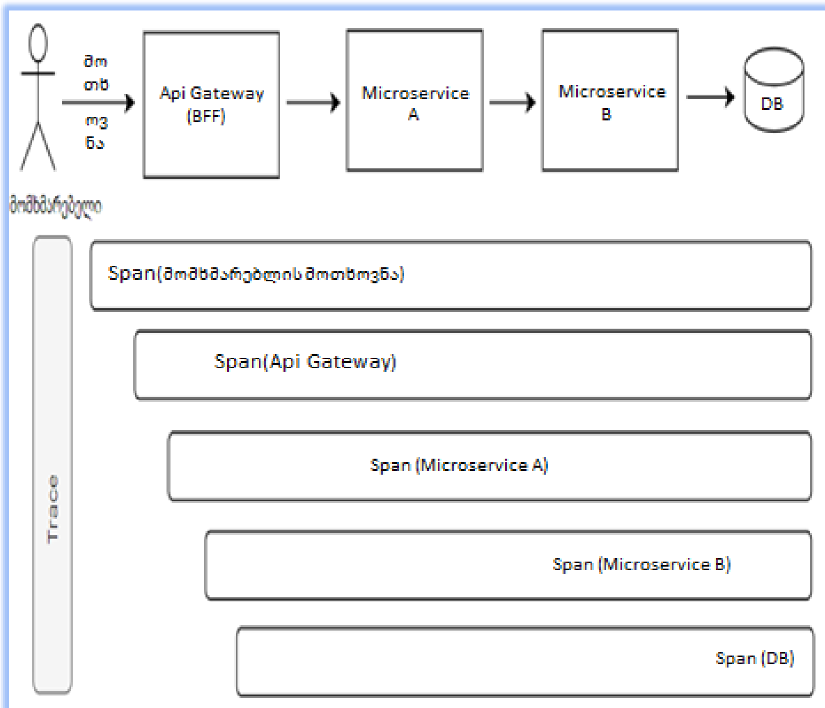
Health Check-ის მიერ დაბრუნებული შედეგის მიხედვით, მონიტორინგის სისტემები აინიცირებს შესაბამის Alert პროცესებს (პრობლემის ამსახველი მეილის ავტომატური დაგზავნა ადმინისტრატორებთან და ა.შ.).

სერვისებს აქვს დეტალური ლოგირება თანამედროვე ლოგირების სისტემებში, რომლებიც გათვლილია მაღალ დატვირთვაზე და მონაცემთა ოპტიმალურ ძებნაზე (მსგავსი ტექნოლოგიები ძებნის ოპტიმალურობას ძირითადად Elastic-Search-ზე დაფუძნებული იმპლემენტაციით აღწევენ. ხშირად გამოყენებადი ტექნოლოგიებია: Graylog, Seq და ა.შ.) [97].

მიკროსერვისებს აქვს ბიზნეს პროცესების ტექნიკური მსვლელობის დეტალური მონიტორინგის და დიაგნოსტირების საშუალება - დისტრიბუციული ტრასირება (Distributed

Tracing). თითოეული სერვისი მიერთებულია Tracing ინფრასტრუქტურასთან და ყოველი ქმედების შემდეგ ახდენს საკუთარი პროცესის მსვლელობის მეტრიკების ლოგირებას.

დისტრიბუციული ოპერაციის შემთხვევაში, ერთი ოპერაციის ქვეშ (Trace) გაერთიანებული ინფრასტრუქტურული გამოძახებები (Span), ერთიანდება კონკრეტული უნიკალური კორელაციის id-ის ქვეშ (ნახ.6.1).



ნახ. 6.1. დისტრიბუციული ტრასირება
(Distributed Tracing)

დისტრიბუციული ტრასირების შედეგად ვიღებთ პროცესის დეტალურ დიარგამას, პროცესის დასაწყისიდან, დასრულების ჩათვლით, სადაც კარგად ჩანს რომელ სერვისში დაიწყო პროცესი, სად გაგრძელდა, სერვისის რომელ შრეზე რა ბიზნეს ლოგიკის შესრულებას რა დრო მოანდომა სისტემამ, რა იყო მოთხოვნების დამუშავების საშუალო დრო, დღის რა მონაკვეთში იზრდება დატვირთვა და სისტემის რომელ წერტილშია დროის მაქსიმალური დანახარჯი.

აღნიშნული მიდგომა ძალზე ეხმარება სისტემის ადმინისტრატორებს და დეველოპერებს პროგრამული ხარვეზის შემთხვევაში, პრობლემური მონაკვეთის დიაგნოს-ტირებაში [88].

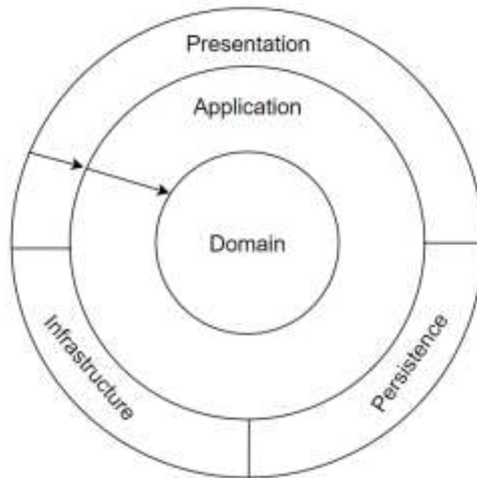
მონიტორინგის სისტემებს აქვს საშუალება, მუდმივად აკონტროლოს სისტემის პრობლემურობის სტატისტიკური მონაცემები. კონკრეტული კრიტიკული ზღვარის გადაც-დენის შემთხვევაში (მოთხოვნის დამუშავების გაზრდილი დრო, Error ლოგების მატება, სისტემის დატვირთულობის ზრდა და სხვ.) ავტომატურად დააგენერიროს (Email/Sms) შეტყობინებები სისტემის ადმინისტრატორებისათვის.

6.1.1. სუფთა არქიტექტურა (CA)

დეველოპმენტის კუთხით, პროგრამულ კოდში აქტიუ-რად გამოიყენება პრაქტიკაში დამკვირდებული თანამედ-როვე პრინციპები და დიზაინის შაბლონები (OOP, SOLID Principles, Software Design Patterns). მიკროსერვისების კოდის არქიტექტურა (Solution Architecture) რეალიზებულია ,,Clean Architecture" (სუფთა არქიტექტურის) მიდგომის მიხედვით,

რათა კონკრეტული სერვისის ჭრილში ბიზნეს მოთხოვნების ზრდამ არ გამოიწვიოს კოდის ხარისხის დეგრადაცია და სერვისში არსებული ფუნქციონალური შრეები იყოს მაქსიმალურად ორგანიზებული/აბსტრაგირებული, ავტომატურად ტესტირებადი, გასაგები და მარტივად რეალიზებადი (ჩანაცვლებადი). მოცემულ არქიტექტურაში ცხადად არის წარმოდგენილი კოდის ფრაგმენტების ორგანიზების წესები. Clean არქიტექტურას საფუძველი ჩაუყარა რობერტ მარტინმა (Robert Cecil Martin - ,,Uncle Bob") [25, 89]

Clean არქიტექტურაში, ფუნქციონალის ბირთვი მოთავსებულია კოდის ცენტრალურ ნაწილში, რომელიც მოიცავს სისტემის ძირითად ობიექტებს და მათ ზოგად ქცევას. არქიტექტურის ამ ნაწილს *Domain* ეწოდება და მას არ გააჩნია დამოკიდებულება კოდის დანარჩენ კომპონენტებზე (ნახ.6.2).



ნახ. 6.2. ,,Clean" არქიტექტურა

Application შრეზე წარმოდგენილია სისტემის ფუნქციონალური ნაწილის (ბიზნესლოგიკის) იმპლემენტაცია, რომელსაც პირდაპირი დამოკიდებულება აქვს მხოლოდ Domain შრეზე. Application შრეზე, გარე სერვისებთან და მონაცემთა საცავებთან ურთიერთობა ხდება კოდის აბსტრაგირებული კომპონენტების გავლით.

Application შრეს არ აქვს ინფორმაცია, მონაცემთა ბაზის პროვაიდერის ან გარე სერვისებთან კომუნიკაციის დეტალების შესახებ. შესაბამისად, Application შრე არის ბიზნეს ლოგიკის მდგრადი კომპონენტი. მისთვის გარე სამყაროსთან კომუნიკაციისათვის საჭირო კომპონენტების გამოცვლა მარტივია – ისე, რომ არ შეიცვალოს ფუნქციონალის ლოგიკური ნაწილი [42]. გარე ინფრასტრუქტურასთან კომუნიკაციის კონკრეტული იმპლემენტაცია წარმოდგენილია შესაბამისად Persistence/ Infrastructure შრეებზე.

Persistence შრე არის მონაცემთა საცავებთან ურთიერთობის აბსტრაქტული კონტრაქტის კონკრეტული იმპლემენტაცია, ხოლო Infrastructure შრე კი – გარე სერვისებთან ურთიერთობის აბსტრაქტული კონტრაქტის კონკრეტული იმპლემენტაცია (ასევე, მიღებულია ამ ორი შრის Infrastructure შრეზე გაერთიანებაც, რაც გულისხმობს ბიზნეს ლოგიკის მონაცემთა წყაროებთან კომუნიკაციის შრის განზოგადებას როგორც მონაცემთა ბაზის, ასევე გარე სერვისებისათვის).

გამომდინარე იქიდან, რომ Entity-ები განთავსებულია Domain შრეზე, არქიტექტურა გვაძლევს საშუალებას, რომ

Infrastructure/Persistence ფუნქციონალური შრე იყოს დამოკიდებული Application შრეზე და არა პირიქით.

Presentation შრე არის კომპონენტის გარე ინტერფეისის იმპლემენტაცია (UI, Controller, etc.). Presentation შრეს დამოკიდებულება აქვს სერვისის ბიზნესლოგიკის შრეზე (Application Layer).

<u>შენიშვნა:</u>

განხილული საკითხის შესაბამისი პროგრამული კოდის რეალიზების პრაქტიკული მაგალითი იხილეთ წიგნის დანართში.

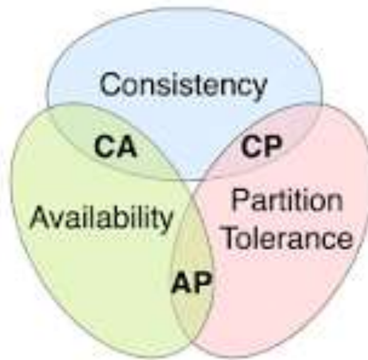
6.1.2. CAP თეორემა

მიკროსერვისულ არქიტექტურაში, სისტემის დისტრიბუციულობიდან გამომდინარე (Partitioning), ხშირად დგება ისეთი მდგომარეობა, როდესაც *წვდომადობასა (Availability)* და *მონაცემთა მთლიანობას* (Data Consistency) შორის გვიწევს არჩევანის გაკეთება (ორივეს ერთდროულად მიღწევა მსგავს შემთხვევაში შეუძლებელია [40,98].

CAP თეორემა (Consistency, Availability, Partition_ tolerance) ან ბრიუერის (Brewer Eric) თეორემა არის ევრისტიკული მტკიცება იმის შესახებ, რომ განაწილებული გამოთვლების ნებისმიერ რეალიზაციაში შესაძლებელია სამი (C, A, P) თვისებიდან მხოლოდ ორის უზრუნველყოფა (ნახ. 6.3) [98,99].

მსგავსი საკითხები წყდება კონკრეტული ამოცანის ანალიზის საფუძველზე, თუ ბიზნეს პროცესში მონაცემთა ბოლო მდგომარეობით დამუშავება პრიორიტეტულია, მაშინ

სინქრონული კომუნიკაცია გარდაუვალია (შემთხვევა, როდესაც სერვისები პირდაპირ იძახებს ერთმანეთს). ამ შემთხვევაში, სისტემა დააბრუნებს პროგრამული ხარვეზის მნიშვნელობას (Error), როდესაც გამოძახებისას რომელიმე კომპონენტში დაფიქსირდება ხარვეზი.



ნახ. 6.3. CAP თეორემის ეილერის მოდელი

ხოლო თუ ვირჩევთ მაღალ ხელმისაწვდომობას კომპონენტების იზოლირების საშუალებით (loosely coupling), მოთხოვნის დამუშავება განხორციელდება წარმატებით, ბოლო ხელთარსებული მონაცემის მდგომარეობით (მაგალითად, Materialized Views), რომელიც შესაძლოა არ იყოს სრულ თანხვედრაში ძირითად მონაცემთან (Master Data). როდესაც ვსაუბრობთ დაყოვნებაზე, ძირითადად საუბარია მილიწამიწამის მასშტაბებზე.

6.1.3. Api Gateway სერვისი

Api Gateway არის ვებ სერვისი და ახდენს შესაბამისი ფუნქციონალის რეალიზაციას:

- კლიენტის მოთხოვნის გადამისამართება შესაბამის მიკროსერვისში (Request Routing - Reverse Proxy);
- მონაცემების აგრეგაცია მიკროსერვისებს შორის (Data Aggregation). საჭიროების შემთხვევაში, კლიენტი აპლიკაციის მოთხოვნის საფუძველზე (Request), Api Gateway ახდენს შესაბამისი მიკროსერვისების გამოძახებას, მონაცემების აგრეგაციას და კლიენტისათვის გაერთიანებული შედეგია დაბრუნებას;
- მომხმარებლის აუთენტიფიკაცია/ავტორიზაცია (Identity Service-ს გამოყენებით, შესაბამის Request-ებზე), რის შემდეგაც ახორციელებს ავტორიზებული მომხმარებლისათვის ბიზნეს პროცესის შესაბამისი მიკროსერვისების გამოძახებას;
- აქვს ქეშირების მექანიზმი, მოთხოვნის ოპტიმალური დამუშავებისათვის;
- ვებ აპლიკაციიდან Api Gateway სერვისის გამოძახებისას, მოთხოვნას (Request) მიყვება კლიენტი კომპანიის ინდეტიფიკატორი (source), რომელიც საბოლოდ გადაეწოდება მიკროსერვისს (გამოიყენება კლიენტი კომპანიის იდენტიფიცირებისათვის) [1, 90].

6.1.4. მიკროსერვისის შემადგენელი კომპონენტები

თითოეულ მიკროსერვისს აქვს საკუთარი Event Handler Windows Service, რომელიც ახდენს შეტყობინებათა სისტემაში განთავსებული მესიჯების (Event Message) დამუშავებას (Subscribe) და კონკრეტული ბიზნესლოგიკის რეალიზებას (მონაცემების შენახვა მონაცემთა ბაზაში, სხვა სერვისის გამოძახება და ა.შ.).

მიკროსერვისებს საჭიროების მიხედვით აქვს ბრძანებების ასინქრონული დამუშავების საშუალება. ამ შემთხვევაში მიკროსერვისის ჭრილში ინერგება Command Handler Windows Service, რომელიც ახდენს შეტყობინებების სისტემიდან (Message Broker) ბრძანებების (Command Message) დამუშავებას (Subscribe) და მათ შესრულებას, შედეგს კი ათავსებს საპასუხო მონაცემთა რიგში (Reply Queue).

მოთხოვნების (Command) ასინქრონულ დამუშავებას აქვს რამდენიმე უპირატესობა. განსაკუთრებით მაღალი დატვირთვის მქონე პროცესები, რომელთა შემთხვევაში ჰორიზონტალური და ვერტიკალური მასშტაბირება შედეგს ვეღარ იძლევა (ასევე მონაცემთა ბაზის დატვირთულობა), ასინქრონული დამუშავება უზრუნველყოფს დატვირთულობის ბალანსს.

გამომძახებელი სისტემა ათავსებს მოთხოვნის შეტყობინებას (Command Message) Message Broker-სისტემაში და არ ელოდება სინქრონული გზით პასუხის დაბრუნებას. სერვისი, როგორც კი შეძლებს თავად დაამუშავებს მონაცემთა რიგში

მოთხოვნას და დაუბრუნებს სისტემას პასუხს ასინქრონულად. მეორე უპირატესობაა წვდომადობა. იმ შემთხვევაში თუ სერვისი ხელმისაწვდომი არ არის, მომკითხავი სისტემა მაინც ახერხებს მოთხოვნის განთავსებას შეტყობინებების სისტემაში (Message Broker), როგორც სერვისი აღადგენს ფუნქციონირებას, იგი თავად დაამუშავებს მონაცემთა რიგიდან მოთხოვნას და სისტემას პასუხს ასინქრონულად დაუბრუნებს.

ერთი სერვისის ფუნქციონალის ფარგლებში, სინქრონულ და ასინქრონულ დამუშავების მექანიზმებს პრაქტიკულად ვიყენებთ სხვადასხვა პროცესებში. მაგალითად, პროდუქციის შესყიდვის დისტრიბუციული ტრანზაქციის მართვისას მარაგის რეზერვირება (შემცირება) ხდება ასინქრონულად, ხოლო პროდუქციის მიღებისას, მარაგების გაზრდა ადმინისტრირების მოდულიდან ხდება სინქრონულად, მარაგების სერვისის (Inventory Service) Web Api-ს გამოძახებით.

მიკროსერვისებს აქვს Event Publisher Windows Service, რომელიც ახორციელებს შეტყობინებების სისტემაში (Message Broker) ნოტიფიკაციების (Message – Event/Command) განთავსებას (Publish). მონაცემთა მდგომარეობის ცვლილება (State Mutation) ქვეყნდება შესაბამისი ბიზნესპროცესის დასახელების მქონე მოვლენის შეტყობინებით (Event Message) message broker საკომუნიკაციო არხში. მაგალითად, როდესაც პროდუქტების მიკროსერვისში (Products Microservice) ხდება ახალი პროდუქტის დამატება, Event Publisher სერვისი

აქვეყნებს ინფორმაციას პროდუქტის დამატების მოვლენის (Event) შესახებ შეტყობინებათა სისტემაში.

Message-ს აქვს შემდეგი სახე: მოვლენის დასახელება (EventName) – Product Created, მოვლენის დეტალები (Message Payload) – {„productName": „testProduct", „color": „blue", „height": "100cm"…}.

თითოეული კლიენტი კომპანიისთვის, შეტყობინებების (Message Broker) სისტემაში იქმნება ინდივიდუალური რიგი (Queue), რათა თავიდან ავირიდოთ სხვადასხვა კომპანიის ინფორმაციის (მესიჯების) ერთ ნაკადში დამუშავება და მივიღოთ მაღალი წარმადობა.

Event-ების გენერირება ხდება მიკროსერვისულ არქი-ტექტურაში გავრცელებული მიდგომის – Event Sourcing მიდგომის პრინციპებზე დაყრდნობით. მიდგომის მიხედვით, სისტემაში ძირითადი ობიექტების მონაცემთა მდგომარეობის ცვლილება წარმოშობს შესაბამის მოვლენას (State Mutation Event) და ხდება თითოეული ამ ცვლილების მოვლენის ინფორმაციის შენახვა მონაცემთა საცავში.

თითოეულ ობიექტზე, შექმნისას და შემდგომ ყოველი ცვლილებისას, ემატება Event-ის ჩანაწერი ბაზაში. აღწერილი პრინციპი, გვაძლევს საშუალებას, აღვადგინოთ მონაცემთა გარკვეული პერიოდის წინ არსებული მდგომარეობა.

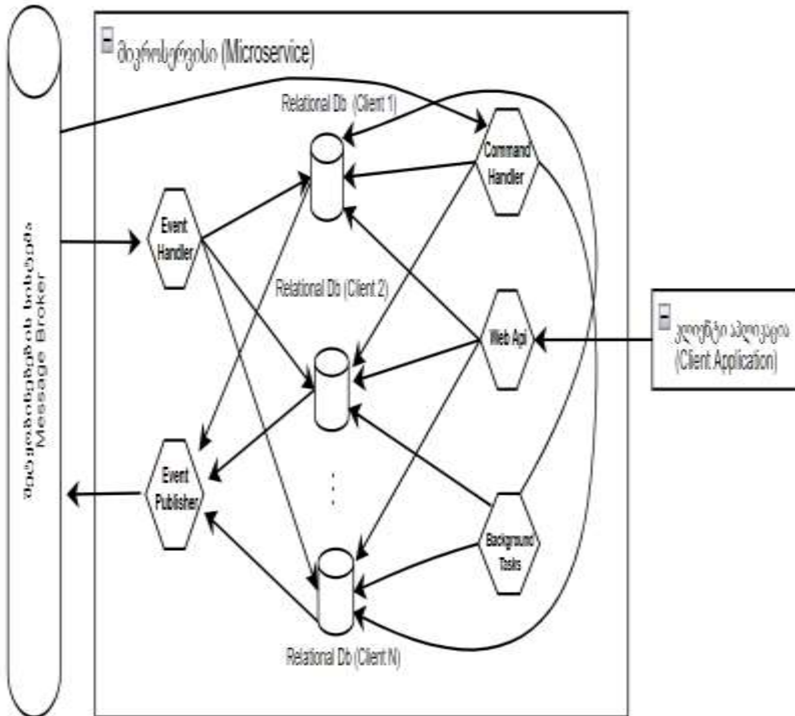
პროცესი გამოიყურება შემდეგნაირად: თუ ავიღებთ რომელიმე ბიზნესობიექტის ამჟამინდელ მდგომარეობას (Current State) და პირობითად 1 კვირის ჭრილში ამავე ობიექ-ტზე მომხდარ მოვლენებს ავსახავთ ჩვენს მიერ შერჩეულ

ობიექტზე უკუსვლით (Play Back), მივიღებთ ობიექტის 1 კვირის წინანდელ მდგომარეობას. Event Sourcing მიდგომის საშუალებით, მარტივია ანალიზი თუ რა გზა და ბიზნეს პროცესი გაიარა კონკრეტულმა ობიექტმა ამათუიმ პერიოდის ან არსებობის სრულ ეტაპზე. ობიექტის ბოლო მდგომარეობის ფორმირება შესაძლებელია ნებისმიერ დროს, Event-ებზე დაყრდნობით.

საჭიროების შემთხვევაში სერვისებს აქვთ დამატებითი კომპონენტი - BackgroundTasks, რომელიც უზრუნველყოფს დაგეგმილი გრაფიკის მიხედვით სხვადასხვა ფუნქციონალის შესრულებას (Background Jobs).

სამუშაოები ეშვება დამოუკიდებლად კონკრეტულ თარიღში და/ან კონკრეტული პერიოდულობით, ადამიანის ჩარევის გარეშე. ფუნქციონალის შესრულების გრაფიკის დასაგეგმად, გვაქვს კონფიგურაციის რამდენიმე საშუალება.

ყველაზე ხშირად გამოყენებადი კონფიგურაციაა კრონ შაბლონის (Cron Pattern) განსაზღვრა, რომლის საშუალებით მარტივად შეგვიძლია დავგეგმოთ ამათუიმ სამუშაოს შესრულების პერიოდულობა. სამუშაოები უმეტესად გამოიყენება მონაცემთა ავტომატური დამუშავების *ბეჩური* პროცესებისთვის (მონაცემთა არქივაცია, დაბლოკილი მომხმარებლების ავტომატური განბლოკვა, დაგეგმილი რეპორტების გენერირება, მომხმარებელი კომპანიების სტატუსის კონტროლი და ა.შ.). მიკროსერვისების შემადგენელი ძირითადი კომპონენტები (აპლიკაციები) გამოიყურება შემდეგნაირად (ნახ.6.4).



ნახ. 6.4. მიკროსერვისების შემადგენელი ძირითადი კომპონენტები

6.2. პროდუქციის შესყიდვის პროცესში მონაწილე მიკროსერვისების აღწერა (მომხმარებლის პორტალი)

6.2.1. პროდუქტების მიკროსერვისი

პროდუქტების მიკროსერვისი არის პროდუქციის მართვის ძირითადი ფუნქციონალის რეალიზაცია. კლიენტი (გამომყენებელი) აპლიკაციებისათვის ბიზნესფუნქციონალი წვდომადია Rest Api-ს საშუალებით.

პროდუქტების მიკროსერვისის გავლით ხდება პრო-დუქციის ინფორმაციის წაკითხვა (პროდუქტების სია, კონ-კრეტული პროდუქტის დეტალური ინფორმაცია) სხვადასხვა პარამეტრებისა და ფილტრების მიხედვით, ასევე პრო-დუქტის და მისი დეტალების დამატება, რედაქტირება, წაშლა, დროებით შეჩერება, სტატუსების მართვა და სხვა პროდუქტთან დაკავშირებული ფუნქციონალის შესრულება, რომელიც გამოიყენება როგორც მომხმარებლის მხარეს არსებულ ბიზნეს პროცესებში, ასევე სისტემის ადმინის-ტრირების მოდულში.

პროდუქტების მიკროსერვისს გარე სერვისებთან ინტე-გრაცია არ აქვს.

გამომდინარე იქიდან, რომ პროდუქციის მრავალ-ფეროვნება და რაოდენობა დამოკიდებულია მომხმარებელი კომპანიის (ბიზნესწარმომადგენლის) ბიზნესსაქმიანობაზე და მის მასშტაბებზე, პროდუქტების სერვისის ფუნქციონალი სხვა სერვისებისაგან გამოირჩევა მაქსიმალურად დინამი-კური სტრუქტურით და მომხმარებელს სთავაზობს მისთვის სპეციფიკური სარეალიზაციო პროდუქტის შენახვისა და

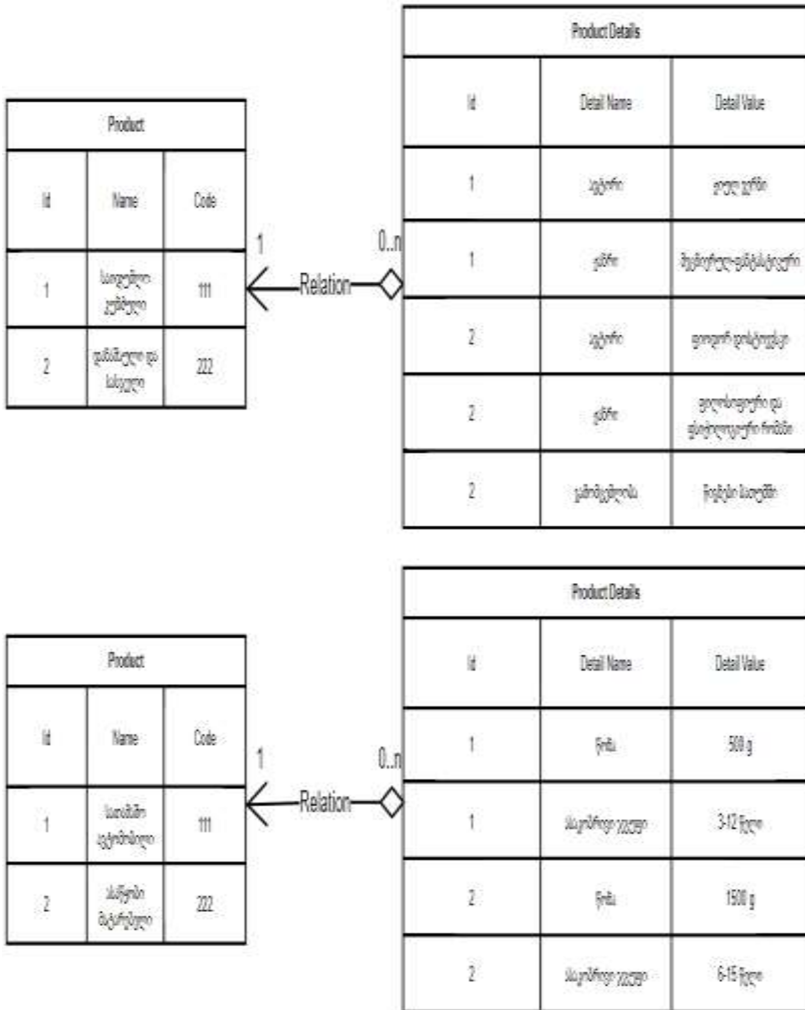
ძებნის განზოგადებულ ფუნქციონალს – კონფიგურირების მოქნილი საშუალებებით.

ყველა სხვადასხვა კომპანიის პროდუქტი წარმოდგენილია განსხვავებული აღწერილობით, მაგალითად, სათამაშოების მაღაზიის პროდუქციის დეტალები გამოიყურება შემდეგნაირად: ***ასაკობრივი ჯგუფი, წონა, მასალა*** და ა.შ. ხოლო წიგნების მაღაზიას პროდუქციის აღსაწერად ჭირდება განსხვავებული პარამეტრები: ***ჟანრი, ასაკობრივი ჯგუფი, ავტორი, გამომცემლობა*** და ა.შ. აქედან გამომდინარე, მონაცემთა ბაზაში პროდუქტი წარმოდგენილია ძირითადი, სხვადასხვა პროდუქციისთვის საერთო პარამეტრების და სპეციფიკური დეტალების სახით (ნახ.6.5).

პროდუქციის ძებნის ფუნქციონალი სხვადასხვა კომპლექსურ მოთხოვნას უნდა აკმაყოფილებდეს, როგორიცაა სისწრაფე, საძიებო პარამეტრების მორგება მომხმარებლის პროდუქციის სპეციფიკაციებზე და სხვ.

პროდუქტის ძებნის ოპტიმალურობა (სისწრაფე) დამოკიდებულია როგორც პროდუქციის რაოდენობასა და მრავალფეროვნებაზე, ასევე ტექნიკურ გადაწყვეტაზე.

პროდუქტის ძებნის მოთხოვნას (Request) შეგვიძლია გადავაწოდოთ დინამიკური ფილტრის მნიშვნელობები (საძიებო ველის და მნიშვნელობების სია - list of key value pair), რაც საშუალებას გვაძლევს პროდუქციის ფილტრაცია მოვახდინოთ არამხოლოდ ძირითადი ველებით, არამედ პროდუქტის სპეციფიკური მნიშვნელობითაც (განსხვავებული ბიზნეს წარმომადგენლის ჭრილში განსხვავებული მნიშვნელობები).



ნახ. 6.5. პროდუქციის დეტალების განთავსება მონაცემთა ბაზაში, წიგნებისა და სათამაშოების მაღაზიის მაგალითზე

პროდუქციის ძებნის ფუნქციონალი რეალიზებულია Pagin-ის საშუალებით და სისტემაში დაწესებულია მოთხოვნილი მონაცემების რაოდენობის მაქსიმალური ლიმიტი (Max Page Size), რათა არასწორი request-ის შემთხვევაში თავიდან ავირიდოთ სერვერის დატვირთულობა.

პროდუქციის სიის ძებნის და ფილტრაციის დროს, დასაბრუნებელ მნიშვნელობაში გვჭირდება პროდუქტის ძირითადი ველები (***დასახელება, კოდი, ფოტო*** და ა.შ.)

პროდუქტის ინფორმაცია ინახება მონაცემთა რელაციურ ბაზაში (ძირითადი საოპერაციო ბაზა) ნორმალიზებულად (ნორმალურ ფორმათა თეორიის საფუძველზე [63, 100]), პროდუქციის ძებნა კი ერთ-ერთი ყველაზე ხშირად შესრულებადი მოთხოვნაა სისტემაში.

აქედან გამომდინარე, ძირითად საოპერაციო ბაზაში (Product Master Database) პროდუქტის ძიება გამოიწვევს ბაზის მაღალ დატვირთულობას და მონაცემთა ნორმალიზების გამო, მოთხოვნის დამუშავების დრო (Query Execution Time) არ იქნება ოპტიმალური.

გამომდინარე იქიდან, რომ სისტემაში პროდუქციის განახლება არ ხდება ძალიან ხშირად (წამში ასობით რაოდენობა ან თუნდაც ყოველწამიერად), მონაცემთა მდგრადობა (მონაცემთა უახლესი მდგომარეობის შენარჩუნება ყოველი მოთხოვნისას), მოთხოვნის ოპტიმალურ დამუშავებასთან შედარებით დაბალ პრიორიტეტულია და მონაცემთა განახლების მცირე დროით დაყოვნება კონკრეტულ ბიზნესპროცესში სავსებით მისაღებია.

აღნიშნულ პრობლემას ოპტიმალურად ერგება მიკროსერვისულ არქიტექტურაში გავრცელებული მიდგომა CQRS (Command-Query Responsibility Segregation). ამ მიდგომის ძირითადი იდეაა მონაცემთა წაკითხვის (Query) და ჩაწერის ოპერაციების (Command) მოდელების იზოლირება.

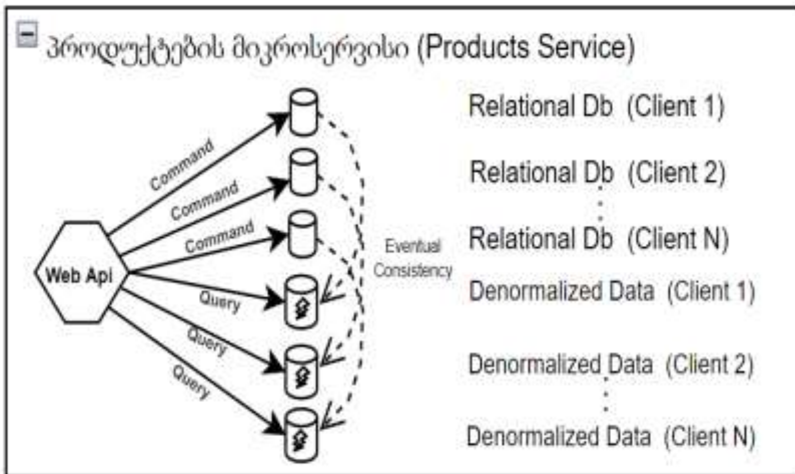
Query ოპერაციები არასოდეს იწვევს მონაცემთა მდგომარეობის ცვლილებას და აქვს სინქრონული ბუნება. Command-ს აქვს ბიზნესოპერაციის შესაბამისი ბრძანების დასახელება (მაგალითად, Create Order Command – შეკვეთის განთავსების შემთხვევაში) და შესაძლოა დამუშავებული იქნას როგორც სინქრონულად, ასევე ასინქრონულად.

მოდელების გამიჯვნასთან ერთად, ოპერაციების გამიჯვნა მონაცემთა საცავების დონეზე საკმაოდ ოპტიმალური და მიღებული პრაქტიკაა.

პროცესი გამოიყურება შემდეგნაირად, ჩაწერის ოპერაცია სრულდება ძირითად საოპერაციო ბაზაში, ხოლო წაკითხვის ოპერაცია სრულდება წაკითხვაზე მორგებულ – *დენორმალიზებულ* მონაცემებში, რომლებიც განცალკევებით, უმეტესად არარელაციურ (NoSQL) მონაცემთა ბაზაში ინახება (შესაძლო ტექნოლოგიები: ElasticSearch, MongoDB და სხვ.) და Event-ების საფუძველზე ახლდება [98,101,102].

ზემოთ აღწერილი პრინციპების მიხედვით, პროდუქტის დამატება, რედაქტირება, წაშლა, დროებით შეჩერება, სტატუსის ცვლილება არის Command ტიპის ოპერაციები, ხოლო პროდუქციის სიის ან პროდუქტის დეტალების მონაცემების გაცემა სერვისიდან არის Query ოპერაცია.

პროდუქტის ობიექტზე ცვლილება ხდება რელაციურ, საოპერაციო ბაზაში, შედეგად აღიძვრება მდგომარეობის ცვლილების Event Message Broker სისტემაში. Event Handler Windows Service ამუშავებს Event-ებს, რის საფუძველზეც ახდენს არარელაციურ მონაცემთა ბაზაში წაკითხვაზე მორგებული სტრუქტურით პროდუქტის ანალოგი მონაცემების ჩაწერას და ლაივ რეჟიმში მისი ცვლილებების ასახვას.



ნახ. 6.5. CQRS მიდგომის რეალიზაცია პროდუქტების მიკროსერვისში

პროდუქციის მართვის მიკროსერვისს გარე სისტემებთან ინტეგრაცია არ აქვს.

6.2.2. შეკვეთების მიკროსერვისი

შეკვეთების მიკროსერვისი (Orders MicroService) არის შეკვეთების მართვის და მასთან დაკავშირებული ბიზნეს ფუნქციონალის რეალიზაცია.

სერვისს აქვს: შეკვეთის განთავსების/გაუქმების, შეკვეთის სტატუსის ცვლილების, შეკვეთის დასრულების, შეკვეთების ისტორიის ნახვის/ექსპორტის – ფუნქციონალები. აგრეთვე *მოთხოვნის დამუშავების სინქრონული და ასინქრონული* საშუალებები (Restful Web Api, Command Handler Windows Service). როგორც უკვე აღვნიშნეთ, შეკვეთის განთავსების ბიზნეს პროცესი წარმოდგენილია სხვადასხვა სერვისებს შორის დისტრიბუციული ტრანზაქციის სახით, შესაბამისად Web Api (Web Service), Command Handler (Windows Service), Event Handler (Windows Service), Event Publisher (Windows Service) კომპონენტებთან ერთად აქვს Order Process Manager Windows Service, როგორც დამატებითი კომპონენტი (Saga Execution Coordinator - SEC).

შეკვეთების მიკროსერვისი თითოეული მომხმარებელი კომპანიის მონაცემთა ბაზაში ახდენს მისთვის საჭირო ინფორმაციის პროექციული ცხრილების (Materialized View) ფორმირებას (***კომპანიის მომხმარებლები, პროდუქტები*** და ა.შ)

პროდუქციის ელექტრონული რეალიზაციის პორტალზე მომხმარებელს უჩანს საკუთარი *შეკვეთების ისტორია, შეკვეთის სტატუსი და შეკვეთის მართვის ფუნქციონალი,* რომელიც ხორციელდება შეკვეთების მიკროსერვისის გავლით. სურვილის შემთხვევაში, მომხმარებელს შეუძლია

კონკრეტული შეკვეთის რეპორტის ამობეჭდვა ან გარკვეული პერიოდის ჭრილში განხორციელებული შეკვეთების ინფორმაციის ექსპორტი.

შეკვეთის განთავსებიდან, რამდენიმე საათის განმავლობაში მომხმარებელს აქვს შესაძლებლობა, სურვილისამებრ გააუქმოს საკუთარი შეკვეთა, თუმცა ამ დროს მოქმედებს საკომისიო სქემა. *შეკვეთის გაუქმება,* ისევე როგორც შეკვეთის განთავსება, დისტრიბუციული პროცესია და საჭიროა ყველა იმ სერვისში, რომლითაც შეკვეთის განთავსების პროცესი განხორციელდა, მოხდეს *შეკვეთის უკუგატარება* (თანხის უკან დაბრუნება, მარაგების აღდგენა, მიტანის სერვისის ჯავშნის გაუქმება, შეკვეთის სტატუსის ცვლილება და მომხმარებელთან შეკვეთის გაუქმების შეტყობინების გაგზავნა).

შეკვეთების მიკროსერვისი გამოიყენება როგორც მომხმარებლის მხარეს, ასევე ადმინისტრირების ნაწილში (შეკვეთის გადამოწმება, შეკვეთის სტატუსის ცვლილება, შეკვეთების რეპორტის გენერირება და სხვ.).

შეკვეთების მართვის მიკროსერვისს გარე სისტემებთან ინტეგრაცია არ გააჩნია.

6.2.3. მომხმარებლების მართვის მიკროსერვისი

მომხმარებლების მართვის მიკროსერვისი (Identity MicroService) არის მომხმარებლების და მათი უფლებების მართვის ძირითადი ფუნქციონალის იმპლემენტაცია. სერვისში გაერთიანებულია მომხმარებლის რეგისტრაციის, აუთენტიფიკაცია/ავტორიზაციის, დაბლოკვა/განბლოკვის,

უფლებების მართვის და სეგმენტის მართვის ფუნქციონალი. მომხმარებელთა სეგმენტის მიხედვით ხდება მათი უფლებათა ჯგუფის გავრცელება, ასევე სეგმენტის მიხედვით შესაძლებელია მომხმარებლებზე კონკრეტული შეთავაზებების გავრცელება.

კლიენტი (გამომყენებელი) აპლიკაციებისათვის ბიზნეს ფუნქციონალი წვდომადია Rest Api-ს საშუალებით.

მიკროსერვისის ჭრილში, სტანდარტული კომპონენტების (სერვისების) გარდა დანერგილია Identity.BackgroudTasks Windows Service, რომელიც უზრუნველყოფს დროებით დაბლოკილი მომხმარებლების ავტომატურ განბლოკვას კონკრეტული პერიოდის შემდეგ (სტანდარტულად 10-15 წუთი, კონფიგურირებადი კომპანიის ჭრილში).

მომხმარებლის დაბლოკვა ხდება აუთენტიფიკაციის მცდელობისას მონაცემების ზედიზედ რამდენიმეჯერ არასწორად შეყვანის შემთხვევაში (სტანდარტულად 3 ცდა, კონფიგურირებადი კომპანიის ჭრილში).

მომხმარებლის აუთენტიფიკაციის პროცესი გამოიყურება შემდეგნაირად:

კლიენტი აპლიკცია (Web Application) Api Gateway Service-ის გავლით იძახებს Identity Service-ს და Request-ს აყოლებს მომხმარებლის აუთენტიფიკაციისათვის აუცილებელ მონაცემებს (***მომხმარებლის სახელი და პაროლი***), Identity Service ამუშავებს მოთხოვნას და შედეგად აბრუნებს Jwt Bearer Access Token-ს და მის შესაბამის Refresh Token-ს, რომლებსაც კლიენტი აპლიკაცია ინახავს ბრაუზერის მხარეს.

ყველა ავტორიზებულ მოთხოვნას (Request) აუცილებელია მიყვებოდეს ვალიდური Access Token. რომელიც ვალიდურია რამდენიმე წუთის განმავლობაში.

არავალიდური ან ვადაგასული Access Token-ის გაგზავნის შემთხვევაში, სერვისი აბრუნებს არაავტორიზებულ შედეგს (Response - 401 Status Code).

Access Token-ის დაძველების შემთხვევაში, მომხმარებლის სესიის გასახანგრძლივებლად (იმისათვის, რომ თავიდან არ გაიაროს აუთენტიფიკაცია/ავტორიზაციის პროცესი), კლიენტ აპლიკაციას აქვს შესაძლებლობა განაახლოს Access token, Refresh Token-ის დახმარებით.

ამისათვის, Api Gateway სერვისის გავლით იძახებს Refresh Token ფუნქციონალს Identity Service-ში და მოთხოვნას აყოლებს თავისთან შენახულ Refresh Token-ს.

როდესაც ზემოაღწერილი ყველა ბიჯი თანმიმდევრულად, ვალიდურად სრულდება, სერვისი კლიენტ აპლიკაციას უბრუნებს განახლებულ Access/Refresh Token-ებს, რის საფუძველზეც, ხდება მომხმარებლის ავტორიზებული სესიის ვადის გახანგრძლივება.

მომხმარებლების მართვის მიკროსერვისი გამოიყენება როგორც მომხმარებლის მხარეს, ასევე ადმინისტრირების ნაწილში (მომხმარებლის ინფორმაციის გადამოწმება, მომხმარებლის დროებით დაბლოკვა/განბლოკვა და ა.შ.).

პროდუქტების მართვის მიკროსერვისს გარე სისტემებთან ინტეგრაცია არ აქვს.

6.2.4. გადახდების მიკროსერვისი

გადახდების მიკროსერვისის (Payment MicroService) ძირითადი ფუნქციონალური ნაწილია პროდუქციის რეალიზაციის გადახდებთან დაკავშირებული ფუნქციონალი და მომხმარებლის ბარათების მართვა. გადახდების მიკროსერვისს აქვს მოთხოვნების როგორც სინქრონული (Restful Web Api), ასევე ასინქრონული (Command Handler Windows Service) დამუშავების საშუალება.

გადახდების მიკროსერვისის ფუნქციონალის გავლით, პროდუქციის ელექტრონულად რეალიზაციის პროცესში, შესაძლებელია გადახდის განხორციელება ასინქრონულად, სხვადასხვა პროვაიდერის საშუალებით.

უსაფრთხოების და პირადი მონაცემების დაცვის საკანონმდებლო მოთხოვნებიდან გამომდინარე, კერძო ბიზნესის წარმომადგენლებს არ აქვთ უფლება საკუთარ მონაცემთა ბაზაში მოახდინონ მომხმარებლის ბარათის ინფორმაციის შენახვა.

მას შემდეგ, რაც მომხმარებელი დარეგისტრირდება სისტემაში, შეკვეთის განხორციელებამდე, ახდენს საბანკო ბარათის(ების) მიბმას, რომლითაც შემდგომში განახორციელებს გადახდას, სხვადასხვა შეკვეთის ფარგლებში.

ბარათის მიბმა ხორციელდება სინქრონულად, გადახდების მიკროსერვისის გავლით, რომელიც თავის მხრივ ახდენს გადახდების პროვაიდერი სისტემის ინტეგრაციას. ბარათის მიბმისას, იტვირთება ბარათის პროვაიდერი კომპანიის ვებგვერდი, რათა მომხმარებლის ბარათის

ინფორმაციის შეყვანა და დამუშავება მოხდეს უსაფრთხოდ. გადახდების (ბარათის) პროვაიდერი სისტემა ახდენს ბარათის იდენტიფიცირებას, და გვიბრუნებს გარკვეული ტიპის იდენტიფიკატორს და ზოგად აღწერას ბარათის შესახებ, რომლის განთავსებაც შეგვიძლია გადახდების მიკროსერვისის მონაცემთა ბაზაში (დაბრუნებული მონაცემი არ შეიცავს ბარათის დეტალურ ინფორმაციას და აკმაყოფილებს უსაფრთხოების ნორმებს. უშუალოდ გადახდის დამუშავებისას, არჩეული ბარათის იდენტიფიცირებას გადახდების მიკროსერვისი ახდენს ამ კონკრეტულ მონაცემზე დაყრდნობით).

მას შემდეგ რაც მომხმარებლის მხრიდან სისტემაში მოხდება ბარათის რეგისტრირება, შესაძლებელია შეკვეთის განხორციელება. შეკვეთის განთავსებისას, მომხმარებელი ირჩევს ბარათს, საიდანაც სურს გადახდის განხორციელება.

მომხმარებელს ნებისმიერ დროს შეუძლია წაშალოს სისტემაში რეგისტრირებული ბარათი და მიაბას ახალი ბარათი (შემთხვევა, როდესაც მოხდა ბარათის დაბლოკვა, ბარათს გაუვიდა ვადა და ა.შ.).

მომხმარებელზე აქტიური ბარათების სია, ბრუნდება გადახდების მიკროსერვისიდან (ფუნქციონალი გამოიყენება როგორც ელექტრონული რეალიზაციის პორტალზე, ასევე ადმინისტრირების მოდულში – კომპანიის თანამშრომლებისათვის).

შეკვეთის შესყიდვის დასრულების შემდეგ მომხმარებელს აქვს შესაძლებლობა ნახოს გადახდის სტატუსი.

როგორც უკვე აღვნიშნეთ, გადახდების მიკროსერვისი გამოიყენება არამხოლოდ მომხმარებლის ფუნქციონალურ ნაწილში, არამედ ადმინისტრირების მოდულშიც.

გადახდების მიკროსერვისიდან კომპანიის თანამშრომელს შეუძლია სხვადასხვა ტიპის ამონაწერის ამოღება, შეკვეთების და მომხმარებლის ჭრილში გადახდების მონიტორინგი. პრობლემური შეკვეთების შემთხვევაში, ადმინისტრირების მოდულიდან შესაძლებელია შეკვეთის ჭრილში თანხის სრული ან ნაწილობრივი დაბრუნება მომხმარებლისათვის (Reversal/Refund).

6.2.5. ადგილზე მიტანის მომსახურების მიკროსერვისი

პროდუქციის შესყიდვის პროცესში მომხმარებელი ირჩევს სასურველ მისამართს, სადაც უნდა მოხდეს *პროდუქციის მიწოდება* (Delivery MicroService). აღნიშნული ფუნქციონალი დისტრიბუციული ტრანზაქციის მართვის პროცესში ხორციელდება ასინქრონულად, შესაბამისად, Delivery Service-ს სტანდარტული სინქრონული კომუნიკაციის ფუნქციონალის გარდა (Restful Web Api), გააჩნია ასინქრონული კომუნიკაციის საშუალება (Command Handler Windows Service).

შეკვეთის განთავსებიდან გარკვეული პერიოდის განმავლობაში (სტანდარტულად 2 საათი, კონფიგურირებადი კომპანიის ჭრილში) მომხმარებელს შეუძლია მისამართის შეცვლა (ხარვეზის შემთხვევაში). 2 საათის შემდეგ, გაყიდვების პორტალის ინტერფეისიდან (Web

Application UI) შეუძლებელი ხდება მისამართის ცვლილება. არსებული შეკვეთის მისამართის შესწორება ხდება სინქრონულად ვებ აპლიკაციიდან Api Gateway სერვისის გავლით, რომელიც თავის მხრივ იძახებს Delivery Service - ის მისამართის ცვლილების ფუნქციონალს.

გამომდინარე იქიდან, რომ პლატფორმა გათვლილია სხვადასხვა კომპანიაზე და თითოეული კომპანიის გაყიდვების მასშტაბი განისაზღვრება უშუალოდ ბიზნესის ზომით, პროვაიდერს (სისტემის მიმწოდებელს) არ აქვს რესურსი, რომ უზრუნველყოს ყველა კომპანიის ჯამური გაყიდვების შესაბამისი მიტანის სერვისი.

შესაბამისად, პლატფორმის კონტრაქტორად განიხილე-ბა ბაზარზე არსებული მიტანის სერვისის წარმომადგენლობა (Glovo,Wolt და სხვ.). მიტანის სერვისის საფასურს მოიცავს სისტემის სერვისის სახით მიწოდების საფასური და დამოკი-დებულია გაყიდვების რაოდენობაზე (პაკეტის ზომაზე).

ტექნიკურად, Delivery Service ახდენს გარე ვებ სერვისების ინტეგრაციას (Glovo, Wolt), სადაც უშუალოდ „გლოვოს" და/ან „ვოლტის" საშუალებით ხორციელდება მომხმარებლის სასურველ დროს მიტანის სერვისის დაჯავშნა. ჯავშნის შესახებ ინფორმაცია ინახება Delivery Service-ის მონაცემთა ბაზაში და მიტანის სერვისის ჯავშნის განთავსებიდან, მიტანის დასრულებამდე, როგორც მომხმა-რებელს, ასევე კომპანიის თანაშრომელს ადმინისტრირების მოდულიდან ნებისმიერ დროს შეუძლია დაგეგმილი მიტანის სერვისის პროგრესის ნახვა, სტატუსის გადამოწმება.

მას შემდეგ, რაც მოხდება პროდუქციის ადგილზე მიტანა, მიტანის სერვისის პროვაიდერი სისტემიდან ჩვენი სერვისების გამოძახების საფუძველზე, სისტემაში ავტომატურად ხდება შეკვეთის დასრულება.

როგორც ფუნქციონალიდან გამომდინარე აღვნიშნეთ, ადგილზე მიტანის სერვისის მიკროსერვისი (Delivery Service) გამოიყენება, როგორც მომხმარებლის, ასევე ადმინისტრირების მოდულის ბიზნეს პროცესებში.

სერვისიდან შესაძლებელია მიტანის ისტორიის ნახვა სხვადასხვა პარამეტრების საშუალებით (კონკრეტული შეკვეთის ჭრილში, პერიოდის შუალედში ფილტრით, მომხმარებლის ფილტრით), რომელიც აქტიურად გამოიყენება კომპანიის თანამშრომლების მიერ ადმინისტრირების მოდულიდან.

შესაბამისად ადგილზე მიტანის მიკროსერვისს მონაცემთა ბაზის დონეზე აქვს შეკვეთის და მომხმარებლის ინფორმაციის პროექციული ცხრილები (Materialized Views), რომელთა განახლებაც Event-ების საფუძველზე ხორციელდება.

6.2.6. შეტყობინებათა მიკროსერვისი

შეტყობინებათა მიკროსერვისი (Notification MicroService) უზრუნველყოფს მომხმარებლებთან და კომპანიის თანამშრომლებთან eMmail/Sms შეტყობინებების გაგზავნის ფუნქციონალს. შეტყობინებების სერვისს აქვს ბრძანებების როგორც სინქრონული (Restful Web Api), ასევე ასინქრონული (Command Handler Windows Service) დამუშავების საშუალება

(შეტყობინების გაგზავნა მომხმარებელთან შეკვეთის დასრულების შედეგ, ხდება ასინქრონულად).

შეტყობინებების მიკროსერვისს აქვს გარე ინტეგრაცია უშუალოდ შეტყობინების დამგზავნ პროვაიდერ სისტემებთან, როგორებიცაა Google Smtp Mail Servers – eMail შეტყობინებების შემთხვევაში და Magticom SMS შეტყობინებების შემთხვევაში.

ერთი მოთხოვნის ჭრილში, შეტყობინებათა მიკროსერვისიდან შესაძლებელია მხოლოდ Sms შეტყობინების, მხოლოდ eMail შეტყობინების ან რამდენიმე სხვადასხვა ტიპის და რაოდენობის შეტყობინების ბეჩურად გაგზავნა.

6.2.7. მარაგების მართვის მიკროსერვისი

მარაგების მართვის მიკროსერვისი (Inventory Microservice) არის მარაგების მართვის და აღწერის ძირითადი ფუნქციონალი. მიკროსერვისს აქვს მოთხოვნის როგორც სინქრონული (Restful Web Api), ასევე ასინქრონული დამუშავების საშუალება (Command Handler Windows Service).

მარაგების მიკროსერვისის მონაცემთა ბაზაში ინახება ინფორმაცია პროდუქციის რაოდენობის, მიღების და რეალიზაციის შესახებ. საქონლის ბრუნვა დეტალურად არის ასახული მოცემულ მიკროსერვისში.

მარაგების მიკროსერვისი მონაწილეობს როგორც მომხმარებლის ნაწილში არსებულ ფუნქციონალში, ასევე ადმინისტრირების მოდულში.

მარაგების მიკროსერვისი ასინქრონულად ამუშავებს მომხმარებლის მხრიდან შეკვეთის ჭრილში ინიცირებულ

მარაგის დაჯავშნის მოთხოვნას, ხოლო სინქრონულად ხდება ადმინისტრირების მოდულიდან მარაგის მიღების მოთხოვნის დამუშავება.

მარაგების მიკროსერვისის საშუალებით კომპანიის თანამშრომლებს შეუძლიათ მარაგების მიღების/შევსების შესახებ ინფორმაციის აღრიცხვა, პროდუქციის მიღება/გაცემის რეპორტების გენერირება/ექსპორტი სხვადასხვა ფილტრების გამოყენებით (***პროდუქტი, თარიღი, პროდუქტის რაოდენობა*** და სხვ.).

6.2.8. შეთავაზებების მიკროსერვისი

შეთავაზებების მიკროსერვისი (Promotion MicroService) არის აქციების და შეთავაზებების მართვის ძირითადი ფუნქციონალი. მიკროსერვისი გამოიყენება როგორც მომხმარებლისთვის განკუთვნილ ინტერფეისში, ასევე ადმინისტრირების მოდულის ფუნქციონალში. შეთავაზებების მიკროსერვისს აქვს მოთხოვნების სინქრონულად დამუშავების საშუალება [91-93].

სააქციო შეთავაზებების მართვა ხორციელდება დინამიუკრად Promotion მიკროსერვისიდან. შეთავაზებების დამატება ხდება კომპანიის თანამშრომლის მიერ ადმინისტრირების მოდულიდან. შეთავაზების გააქტიურებამდე, შესაძლებელია Preview რეჟიმში შეთავაზების ნახვა.

ადმინისტრირების მოდულში შეთავაზების დამატების პროცესს აქვს HTML ედიტორი, რომ კონკრეტული კომპანიის წარმომადგენელს, რომელსაც არ გააჩნია ტექნიკური კომპეტენცია, drug and drop პრინციპით შეეძლოს საკუთარი

კომპანიის შეთავაზების ვიზუალური ნაწილის სურვილისამებრ ცვლილება, ბრენდინგზე მორგება და ა.შ.

შეთავაზება არსებობს სხვადასხვა ტიპის და კატეგორიის. უმეტესად შეთავაზება აქტიურია კონკრეტული თარიღის შუალედში, აქედან გამომდინარე, შეთავაზებების აქტიურობის შეწყვეტა ხდება დამატებითი კომპონენტის (Promotions.BackgroundTasks Windows Service-ის) საშუალებით. ყოველ ღამით გაიშვება დავალება, რომელიც ამოწმებს შეთავაზებას აქვს თუ არა აქტივობის ვადა გასული, და შესაბამისად უცვლის სტატუსს ისეთ შეთავაზებებს, რომელთა აქტივობის ვადა ამოიწურა.

როგორც უკვე აღვნიშნეთ მომხმარებლების მართვის მიკროსერვისის აღწერისას, მომხმარებელზე შესაძლებელია ამა თუ იმ სეგმენტის მინიჭება, რომლის მიხედვითაც ხდება მომხმარებლებზე კონკრეტული შეთავაზებების გავრცელება. სისტემის მომხმარებელს უჩანს აქტიური, მის სეგმენტზე არსებული ან ზოგადი – სეგმენტგარეშე შეთავაზებები.

6.3. მეექვსე თავის დასკვნა

შემუშავებული და წარმოდგენილია მცირე და საშუალო ბიზნესის მართვის ციფრული პლატფორმის არქიტექტურისთვის საჭირო მიკროსერვისების შედგენილობა და სტრუქტურა, ურთიერთკავშირი. გაანალიზებულია მათი ფუნქციონალური ნაწილი და განსაზღვრულია შესრულების ალგორითმები, რომლებიც საბოლოოდ რეალიზებულია პროგრამულად.

თავი 7

მიკროსერვისული არქიტექტურის მონაცემთა საცავის ჰორიზონტალური მასშტაბირება და სისტემის ადმინისტრირება

7.1. დეცენტრალიზებული მონაცემთა მართვა

მონაცემთა ბაზა არის მონაცემთა ორგანიზებული კოლექცია, რომელიც ჩვეულებრივ ინახება და ხელმისაწვდომია ელექტრონულად კომპიუტერული სისტემიდან. ის მხარს უჭერს მონაცემთა ელექტრონულ შენახვასა და მანიპულირებას [103].

სისტემის დეკომპოზიციისას აუცილებელი ხდება მონაცემთა სანახების (საცავების) დონეზე კომპონენტების იზოლირება (Data Hiding). მიკროსერვისულ არქიტექტურაში არსებობს მონაცემთა მართვის სხვადასხვა სტრატეგიები (შაბლონები) და სერვისებს დასაშვებია ჰქონდეს განსხვა-ვებული მონაცემთა მართვის მოდელი. აქედან გამომდინარე, მიკროსერვისების მართვაში გავრცელებულია თითოეული სერვისისთვის დამოუკიდებელი მონაცემთა ბაზის უზრუნ-ველყოფის მიდგომა (Database Per Service). ამიტომაც არაა ერთი ცენტრალიზებული საცავი. შესაბამისად, *რთულია მონაცემთა მთლიანობის შენარჩუნება* სისტემის დონეზე (Data consistency). *ეს პრობლემა* მიკროსერვისულ არქიტექ-ტურაში ერთ-ერთი ფუნდამენტური გამოწვევაა, რომლის გადაწყვეტის რამდენიმე (პრაქტიკაში გავრცელებული) მიდგომა არსებობს [104, 105, 107].

ჩვენ მიერ განხილულ სისტემაში თითოეული კომპანიისათვის მიკროსერვისებს აქვს დამოუკიდებელი მონაცემთა საცავიი (ან ბაზა), რათა მოხდეს მონაცემთა სრული იზოლირება, შევინარჩუნოთ მონაცემთა ბაზის ოპტიმალური ზომა და ტექნიკური პროცესები გამოირჩეოდეს მაღალი წარმადობით. მიკროსერვისში შემოსულ მოთხოვნას (Request) Api Gateway სერვისი გადმოსცემს source მნიშვნელობას, რომელიც განსაზღვრავს კლიენტი კომპანიის ინდეტიფიკატორს. სერვისები შესაბამისი იდენტიფიკატორის მიხედვით განსაზღვრავს თუ რომელი მონაცემთა ბაზა უნდა იქნას გამოყენებული მოთხოვნის დასამუშავებლად.

კომპანიათა ინფორმაცია მიკროსერვისებისთვის წარმოდგენილია კეშის სახით და გამოირჩევა მონაცემთა სწრაფი დამუშავების უპირატესობით. ყველა მოთხოვნის დამუშავება მიკროსერვისებში ხდება კეშში, კომპანიის იდენტიფიცირების შემდეგ. ეს ადგენს თუ რომელ ბაზას დავუკავშირდეთ (DB Connection) მოთხოვნის დამუშავებისას.

როგორც აღვნიშნეთ, იმისათვის რომ მიღწეული იყოს მაღალი ხელმისაწვდომობა და წარმადობა, მიკროსერვისები უნდა იყოს დამოუკიდებელი და არ უნდა ჰქონდეს ერთმანეთთან პირდაპირი (სინქრონული) დამოკიდებულება (იშვიათი გამონაკლისების გარდა, როდესაც მონაცემის სწორი მდგომარეობით დამუშავება უპირატესია, ვიდრე წვდომადობა და დამუშავების სისწრაფე).

მიკროსერვისი ახდენს საკუთარი ბიზნეს დომეინის მოთხოვნებთან შესაბამისი მონაცემების იზოლირებას და

წარმოადგენს ამ მონაცემების ძირითად სამართავ (Master) სისტემას. ხშირად დგება მომენტი, როდესაც კონკრეტულ მიკროსერვისში არსებული ძირითადი მონაცემი დაკავშირებულია სხვა მიკროსერვისში არსებულ მონაცმეთან და ამავდროულად, მათ შორის არის ცხადი ლოგიკური საზღვრები. ეს კი განსაზღვრავს მონაცემების კონკრეტულ მიკროსერვისთან მიკუთვნების პრინციპს (DDD) [66].

ინფორმაციული სისტემა წარმოდგენილია როგორც სერვისების ერთობლიობა და წყვეტს კონკრეტული ბიზნესისათვის სპეციფიკურ გლობალურ პრობლემას. ხშირად მოთხოვნის დასამუშავებლად საჭიროა კონკრეტული მონაცემის მნიშვნელობა, ოღონდაც სხვა მიკროსერვისიდან.

მაგალითად, *პროდუქციის შესყიდვისას* განხორციელებული *შეკვეთა,* რეგისტრირდება *Order მიკროსერვისის* მონაცემთა რელაციურ ბაზაში, ხოლო *პროდუქტის* შესაბამისი დეტალები შენახულია *Product სერვისის* მონაცემთა რელაციურ ბაზაში. ტექნიკურად, 1 კონკრეტული შეკვეთა დაკავშირებულია პროდუქტთან პროდუქტის იდენტიფიკატორ(ებ)ის საშუალებით 1:N კავშირით (One To Many Relation).

როდესაც გვჭირდება შეკვეთის დეტალების ნახვა, პროდუქტის იდენტიფიკატორი ინფორმაციულად ბიზნესისთვის გაუგებარ მნიშვნელობას იძლევა. მოთხოვნა ჩნდება შეკვეთასთან ერთად, რეალიზებული პროდუქციის ინფორმაციის ნახვის შესახებ.

გამომდინარე იქიდან, რომ *შეკვეთის* და *პროდუქტის* შესახებ ინფორმაცია ფიზიკურად სხვადასხვა მონაცემთა

ბაზაში ინახება, მონაცემთა ლოგიკურად გადაბმა ბაზის დონეზე და ერთიანად ამოკითხვა (მაგალითად join-ით) შეუძლებელია. ხოლო *შეკვეთების მიკროსერვისიდან* სინქ-რონული მიმართვა (Api Call) *პროდუქტების მიკროსერვისზე* არღვევს მიკროსერვისების ავტონომიურობას (დამოუკიდებ-ლობას, Loosely Coupling), რაც იმას ნიშნავს, რომ თუ პროდუქტების სერვისს ექნება დროებითი ტექნიკური შეფერხება, შეკვეთების სერვისიდან შეკვეთის ინფორმაციის ნახვა არ იქნება შესაძლებელი.

მეორე პრობლემა, რაც სინქრონული კომუნიკაციის გზით მონაცემთა გაერთიანებას აქვს, არის *მოთხოვნის დამუშავების დრო.* შეკვეთის ინფორმაციის გაცემისას დამატებითი გამოძახება, დროის არაოპტიმალურ დანახარ-ჯებთან არის დაკავშირებული. იგივე პრობლემა დგება სხვადასხვა ბიზნეს პროცესში (მაგალითად, მიტანის სერვისის ჯავშნის ნახვისას, საჭიროა შეკვეთის დეტალების დამატება, რომ მარტივად მოხდეს შეკვეთის იდენტიფიცი-რება. შეთავაზებების სერვისიდან მომხმარებლის მიერ გამოყენებული შეთავაზებების/აქციების რეპორტის ამოღები-სას მომხმარებლის ინფორმაციის დამატება იდენტიფიცირე-ბისათვის და მრავალი სხვ.).

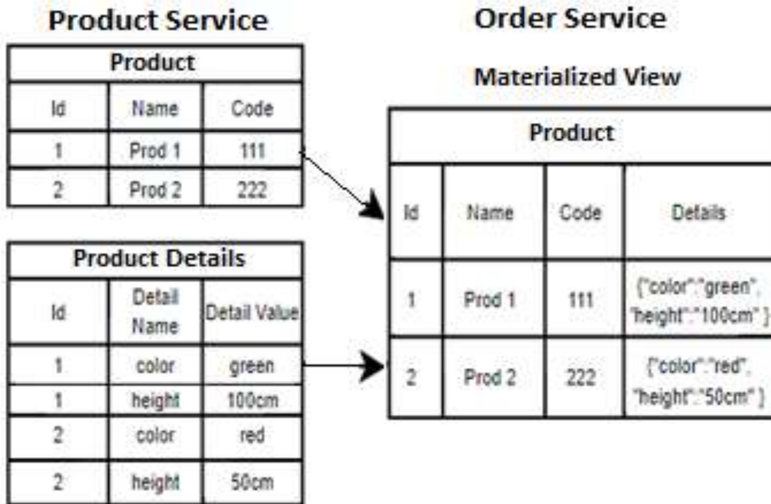
ამგვარი პრობლემის მოსაგვარებლად, ჩვენ მიერ აღწე-რილ სერვისებში ვიყენებთ მიკროსერვისულ არქიტექტურაში გავრცელებულ – „Materialized View" მიდგომას.

ამ მიდგომის მიხედვით, სერვისები, რომლებსაც ესაჭი-როება სხვა მიკროსერვისის მართვის ქვეშ მყოფი მონაცემები,

საკუთარ მონაცემთა ბაზაში აჩენს მათ მოთხოვნებზე მორგებულ მონაცემთა ოპტიმალურ, წაკითხვაზე ორიენტირებულ, *დენორმალიზებულ პროექციულ ცხრილებს* (Read-Only Views), რომელთა განახლებაც სხვა სერვისების გამოქვეყნებული მოვლენების (State Mutation Event) საფუძველზე ხდება. [108, 109]

უნდა აღინიშნოს, რომ პროექციული ცხრილების მოვლენებზე დაფუძნებულ განახლებას (Event Based Update) ყოველთვის აქვს რაღაც გარკვეული დროითი დაყოვნება (უმეტეს შემთხვევაში საუბარია მილიწამის ან წამის მასშტაბებზე). ტექნიკურად, პროცესი გამოიყურება შემდეგნაირად: მიკროსერვისმა, რომელიც ახორციელებს ძირითადი მონაცემის მართვას (მაგალითად, პროდუქტების Product-მიკროსერვისი ზემხსენებული მაგალითიდან), ახდენს მონაცემთა მდგომარეობის ცვლილებას (დამატება, რედაქტირება, წაშლა - State Mutation) და ამ მოვლენის (Event) შესახებ აქვეყნებს ინფორმაციას (Message) შეტყობინებების სისტემაში (Message Broker), რომელსაც აყოლებს მონაცემთა მდგომარეობის ცვლილების დეტალებს (Event Details).

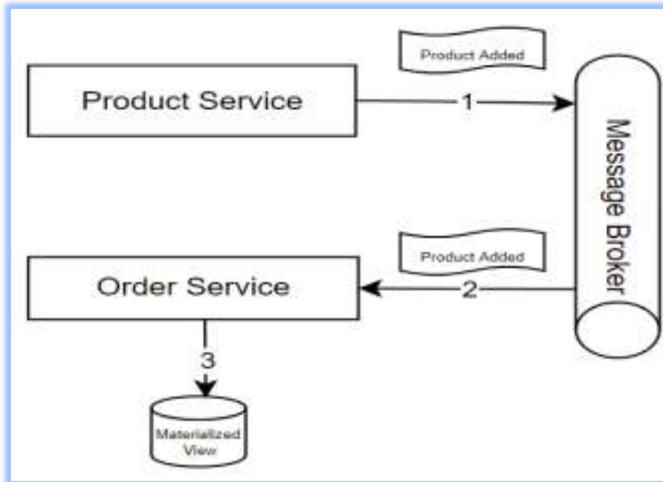
მიკროსერვისი, რომელიც მისთვის მორგებული სტრუქტურით ინახავს სხვა მიკროსერვისის მართვის ქვეშ მყოფ მონაცემს (მაგალითად, შეკვეთების *Order-მიკროსერვისი* ზემოხსენებული მაგალითიდან), ამუშავებს მოვლენას შეტყობინების სისტემიდან და მიღებული ინფორმაციის საფუძველზე, ახდენს პროექციულ ცხრილში მონაცემის მისთვის სასურველი ფორმით განლაგებას (ნახ.7.1).



ნახ. 7.1. პროდუქტის მონაცემების გარდაქმნა როგორც Materialized View *შეკვეთების* მიკროსერვისში

შეკვეთების მიკროსერვისის (Order Microservice) მიერ, პროდუქტების ცვლილების ინფორმაციის უწყვეტ რეჟიმში დამუშავება ხდება *ასინქრონულად*, რომელიც სერვისებს შორის ქორეოგრაფიაზე დაფუძნებული კომუნიკაციის მაგალითია (და სერვისებს შორის სინქრონულ კომუნიკაციას თავიდან გვარიდებს (ნახ. 7.2).

იმისათვის, რომ მონაცემთა სინქრონიზაციის პროცესი იყოს ბოლომდე გამართული, სისტემას უნდა ჰქონდეს მონიტორინგის მექანიზმები, რომლებიც პერიოდულად მოახდენს ძირითად მონაცემთა მდგომარეობის შედარებას Materialized View-ებთან (მონაცემთა რაოდენობების დადარება და ა.შ.).



ნახ. 7.2. პროდუქტის დეტალების სინქრონიზაცია შეკვეთების მიკროსერვისში

სხვაობის შემთხვევაში, დააგენერირებს შესაბამის შეტყობინებას ადმინისტრირების ჯგუფთან, რათა დროულად მოხდეს სისტემაში არსებული ხარვეზის იდენტიფიცირება და აღმოფხვრა.

მსგავსი მექანიზმები ძირითადად წარმოდგენილია რეკონსილაციის პროცესების (Jobs) პერიოდული შესრულებით. მონიტორინგის სისტემის ნაწილია ასევე შეტყობინებათა სისტემის (Message Broker) მონაცემთა რიგების (Queue) დატვირთულობის და მონაცემთა რაოდენობის კონტროლი.

დღეისათვის, პრაქტიკაში გავრცელებულია განსხვავებული Message Broker სისტემები (RabbitMq, Apache Kafka და ა.შ.) [81,82]. მათ შორის ვხვდებით ტექნოლოგიურ და ფუნქციონალურ სხვაობებს, თუმცა, *მთავარი კრიტერიუმი,*

რომ სისტემა უნდა იყოს *ასინქრონულ კომუნიკაციაზე* მორგებული (Command/Event Driven Communication), არის სისტემის საფუძველი.

ჩვენ მიერ დაპროექტებული არქიტექტურა, თავისი სპეციფიკით ერგება ნებისმიერი ტიპის შეტყობინებების სისტემას, მიუხედავად ტექნოლოგიისა და ფუნქციონალური შესაძლებლობებისა. მაგალითად, თუ შეტყობინებების სისტემად (Message Broker) განვიხილავთ Apache Kafka-ს, მას აქვს სხვადასხვა ტიპის Plugin-ები [81]. ზოგიერთი მათგანი მესიჯების ნაკადებთან (Message Streams) მუშაობის მოქნილ ინსტრუმენტებს გვთავაზობს (მაგალითად, Kafka Streams). იგი გვაძლევს საშუალებას, გავაერთიანოთ მესიჯების რამდენიმე ნაკადი, მონაცემები გავამდიდროთ ბაზიდან.

Kafka Streams-ის საშუალებით, შესაძლებელია იმის მიღწევა, რომ მიკროსერვისებმა საკუთარი ბაზიდან აირიდონ პროექციული ცხრილები (Materialized Views), თუმცა, RabbitMq-ში იგივე პრინციპით მონაცემთა დამუშავება, Message Broker-ის დონეზე შესაძლებელი არ არის.

გამომდინარე იქიდან, რომ ჩვენ განვიხილავთ სისტემის ზოგად არქიტექტურულ თარგს (Enterprise Architecture) და არ გამოვყოფთ კონკრეტულ ტექნოლოგიებს, რომლებზეც უნდა შეიქმნას სისტემა აუცილებლად (ჩვენ მათ წარმოვად-გენთ რეკომენდაციის სახით), სისტემის კომპონენტებში განვიხილავთ ისეთ დეტალებს, რომ არქიტექტურა გამოყენე-ბადი იყოს ნებისმიერი ტექნოლოგიური და ინფრასტრუქ-ტურული მოცემულობისათვის.

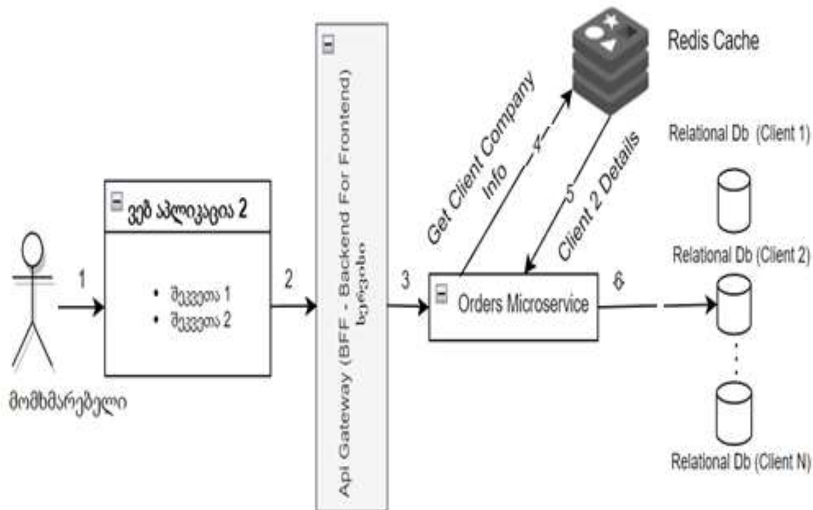
7.2. კლიენტი კომპანიის მონაცემთა საცავის იდენტიფიცირების პროცესი

ჩვენს მიერ დაპროექტებულ სისტემაში, მოთხოვნის დამუშავების დროის ოპტიმიზირება მნიშვნელოვან საკითხს წარმოადგენს. მომხმარებლის მხრიდან ინიცირებული ყველა მოთხოვნა საჭიროებს შესაბამისი კომპანიის მონაცემთა ბაზის იდენტიფიცირებას, სადაც სრულდება მოთხოვნა (ბიზნეს ოპერაცია). მომხმარებელი კომპანიის იდენტიფიცირება, ინფრასტრუქტურული მისამართების გარჩევა და ინფორმაციის დამუშავება უნდა გამოირჩეოდეს მაღალი წარმადობით.

მიკროსერვისები, რომლებზეც ნაწილობრივ უკვე ვისაუბრეთ, თავის მხრივ იყენებენ სისტემის მომწოდებლის მიერ რეგისტრირებული კომპანიების დენორმალიზებულ მონაცემებს (ქეშს – Cache), რომელიც ინახება In Memory მონაცემთა საცავში (ყველაზე გავრცელებული პაკეტებია: MongoDb, Redis და სხვ.). კეშირების საშუალებით, მონაცემების დამუშავება ბევრად სწრაფი/ოპტიმალურია ვიდრე რეალაციური SQL-ბაზებისა.

მაგალითის სახით, განვიხილოთ, მომხმარებლის მხრიდან შეკვეთის დეტალების მოთხოვნა (Request) (ნახ.7.3). პროცესი გამოიყურება შემდეგნაირად: კლიენტი აპლიკაცია Api Gateway სერვისს მიმართავს შესაბამისი პროდუქტის დეტალების მოთხოვნით (ვებ-სერვისის სინქრონული გამოძახება - Api Call), Api Gateway სერვისი თავის მხრივ მოთხოვნას ამისამართებს შეკვეთების მიკროსერვისში,

სინქრონული Api Call-ის საშუალებით. შეკვეთების მიკრო-სერვისი (Restful Web Api) მოთხოვნის დამუშავებისას პირველ ეტაპზე ამოწმებს კლიენტი კომპანიების კეშს, შემდეგ მიღებული ინფორმაციის საფუძველზე, ხდება კომპანიის და მისი მონაცემთა ბაზის ინფრასტრუქტურული მისამართის იდენტიფიცირება და მოთხოვნის დამუშავების პროცესის გაგრძელება. თუ კომპანიასთან *უქმდება ხელშეკრულება*, მაშინ კომპანიების ადმინისტრირების მოდულიდან ხდება კომპანიის *აქტიური სტატუსის შეჩერება.* შედეგად, ნებისმიერი გამოძახება (Api Call), რომელსაც პარამეტრად მოყვება არააქტიური კომპანიის იდენტიფიკატორი, Back_End სერვის-ების დონეზე დაიბლოკება და მომხმარებელი ვერ შეძლებს ვერანაირი ინფორმაციის დამუშავებას ამ კომპანიის ჭრილში.



ნახ. 7.3. კომპანიის მონაცემთა ბაზის იდენტიფიცირების პროცესი კეშის გამოყენებით

კომპანიის ქეშის განახლება ხდება *კომპანიების ადმინისტრირების* მოდულიდან, კლიენტი კომპანიის ნებისმიერი სახის ცვლილების ჭრილში. ასევე, პერიოდულად Company.BackgroundTasks Windows Service-ის დავალების (Job) საშუალებით.

7.3. სერვისების ჰორიზონტალური მასშტაბირება მაღალი დატვირთულობის პირობებში

კლიენტი კომპანიების ჭრილში მონაცემთა ბაზის იზოლირება (1:1 კავშირი), სერვისების ვერსიის ცენტრალიზებული შენარჩუნება და დატვირთვის მიხედვით ჰორიზონტალური მასშტაბირება გვაძლევს უპირატესობას, რომ თუ სისტემაში ხდება პროგრამული ხარვეზის გამოსწორება ან ფუნქციონალის სრულყოფა, მაშინ სერვისების დანერგვა არ ხდება თითოეული კლიენტისთვის ცალკ-ცალკე.

სერვისების დატვირთვის განაწილებისთვის, ჰორიზონტალური მასშტაბირება კონტეინერიზაციის მექანიზმით წარმოებს, ხოლო მონაცემთა ბაზის დატვირთულობას კრიტიკული და ხშირად მოთხოვნადი ფუნქციონალისათვის (მაღალი დატვირთვის მქონე წერტილები) ასინქრონული კომუნიკაციით წყდება. (მაგალითი: ჩვენ მიერ განხილული ორკესტრირებაზე დაფუძვნებული საგა, პროდუქციის რეალიზაციის ბიზნეს პროცესში).

ჩვენ სისტემაში სერვისებთან შედარებით, მონაცემთა ბაზის დატვირთულობა ბევრად ნაკლებია, რადგან ბაზების დატვირთვა კომპანიის ჭრილში ბალანსდება (თითოეულ

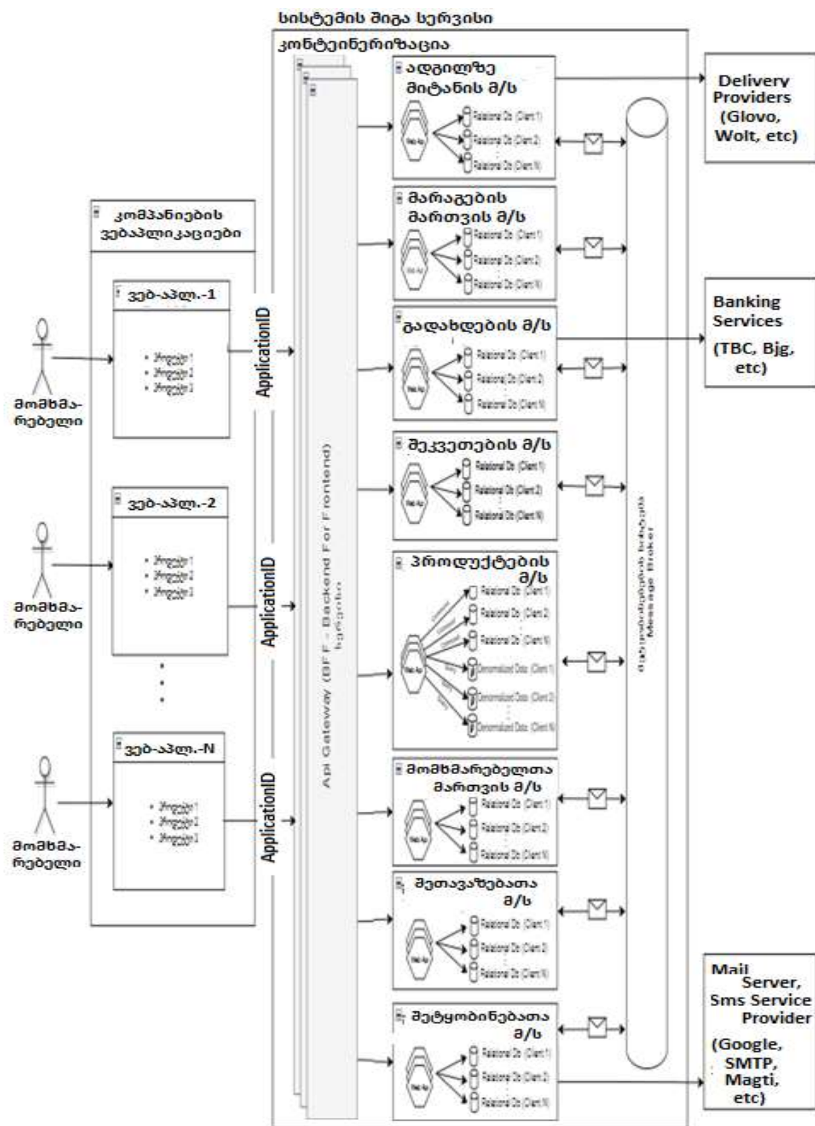
მომხმარებელ კომპანიას ცალკეული იზოლირებული მონაცემთა ბაზა აქვს).

ჩვენ მიერ აღწერილი მიკროსერვისები, კონტეინერიზაციის მექანიზმით, განთავსებულია Kubernetes კლასტერზე და ავტომატურად ხდება სერვისების დატვირთულობის მეტრიკების პერიოდული გადამოწმება – უწყვეტ რეჟიმში.

თითოეული სერვისისათვის, Kubernetes ტექნოლოგიის საშუალებით, განსაზღვრულია რესურსების მოხმარების ოპტიმალური მდგომარეობა, ე.წ. ,,Desired State". ეს კონფიგურაცია გვაძლევს საშუალებას განვსაზღვროთ სერვისის მუშაობის ოპტიმალური რესურსების რაოდენობა, როგორიცაა *პროცესორის დატვირთულობა, მეხსიერების მოხმარების დონე* და ა.შ.

თუ სისტემაში ფიქსირდება მაღალი დატვირთულობა, მაშინ სერვისის მიერ რესურსების მოხმარების დონე იწევს მაღლა, შესაბამისად სერვისის 1 Instance-სთვის თუ მოხდება რესურსების მეტი რაოდენობით მოხმარება, რაც დაკონფიგურირებულია საშუალო დატვირთულობის პირობებისთვის, მაშინ Kubernetes კლასტერი ავტომატურად ახდენს სერვისების Instance-ების გამრავლება/მასშტაბირებას (ჰორიზონტალური სკალირება) (ნახ.7.4) [94, 110].

მაღალი დატვირთულობის მქონე ბიზნეს პროცესებისთვის (მაგალითად, პროდუქციის ელექტრონულად შესყიდვის ბიზნეს პროცესი), სისტემაში ვიყენებთ მოთხოვნის ასინქრონულად დამუშავების გზას.



ნახ. 7.4. სერვისების ჰორიზონტალური მასშტაბირება (1)

ეს ნიშნავს იმას, რომ ყოველი მოთხოვნის (Command Message) განთავსება ხდება შეტყობინებების Message Broker სისტემის რიგში (Queue) მიმდევრობის დაცვით და სისტემა თავად ახდენს მოთხოვნის დამუშავებას მაშინ, როგორც კი ამის შესაძლებლობა ექნება.

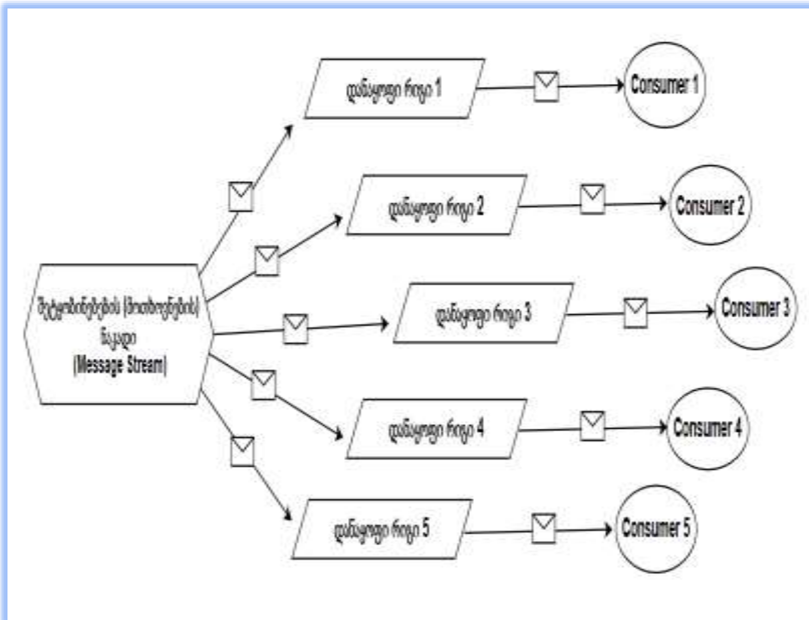
მიუხედავად იმისა, რომ მოთხოვნის ასინქრონული დამუშავება ამცირებს სერვისების დატვირთვას სისტემის მაღალი დატვირთვის პირობებში, იმისათვის, რომ შეტყობინებების სისტემის რიგებში არ დაგროვდეს ბევრი მოთხოვნა (Message) (შემთხვევა, როდესაც სისტემა ვერ ასწრებს იგივე სისწრაფით მოთხოვნის დამუშავებას, რა სიხშირითაც ხდება მოთხოვნის შემოდინება), ვიყენებთ *მოთხოვნების პარალელურად დამუშავების სტრატეგიას*, ისე, რომ შენარჩუნებული იქნეს რიგითობა (თანმიმდევრობა) კონკრეტული მომხმარებლის მოთხოვნებს შორის.

ამ შემთხვევაში ხდება შეტყობინების დამუშავების წერტილების – Consumer-ების (მომხმარებლების) გამრავ-ლება, ანუ ჰორიზონტალური მასშტაბირება, რათა მოხდეს მოთხოვნის ნაკადის პარალელურად დამუშავება.

პროცესი გამოიყურება შემდეგნაირად: მოთხოვნების სრული ნაკადი (Message Stream) ნაწილდება შესაბამის დანაყოფ რიგებში (Partition Queue) შეტყობინების კონკრე-ტული მნიშვნელობის მიხედვით (ჩვენ შემთხვევაში, *პროდუქციის შესყიდვის ბიზნეს პროცესში* ეს მნიშვნელობა არის მომხმარებლის იდენტიფიკატორი, რათა ერთი კონკრე-ტული მომხმარებლის მოთხოვნის დამუშავება მოვახდინოთ

რიგითობის დაცვით, ხოლო სხვა მომხმარებლის მოთხოვნა დავამუშავოთ პარალელურად).

შეტყობინებების Message Broker სისტემაში იქმნება დანაყოფი (Partition) რიგები, სადაც ნაწილდება მოთხოვნის ნაკადი მომხმარებლების ჭრილში. თითოეულ რიგს ყავს ექსკლუზიური მომხმარებელი (Consumer) (ნახ.7.5).



ნახ. 7.5. სერვისების ჰორიზონტალური მასშტაბირება (2)

Consumer-ების სკალირების გზით, ვაღწევთ მესიჯების პარალელურად დამუშავებას, ისე რომ საჭიროებისამებრ ვიცავთ რიგირთობას და არ ვახდენთ მომხმარებლის შეკვეთების არათანმიმდევრულ დამუშავებას.

მომხმარებლის ჭრილში მოთხოვნის არათანმიმდევრულად დამუშავება დაკავშირებულია მრავალ პრობლემასთან. თითოეული შეკვეთისათვის მოწმდება ხელმისაწვდომი ნაშთი და აღიძვრება მრავალი სხვა პროცესი, იმ შემთხვევაში თუ მომხმარებელს არ აქვს საკმარისი თანხა 2 შეკვეთის განსახორციელებლად, შესაძლოა პირველად დამუშავდეს ბოლოს განთავსებული შეკვეთა, რაც გამოიწვევს მომხმარებლის უკმაყოფილებას და სისტემაში მონაცემთა არასწორ მდგომარეობას.

რიგითობის დაცვა ასევე კრიტიკულია შეკვეთის რედაქტირების პროცესშიც, თუ ჯერ დავამუშავებთ ბოლო ცვლილების შეტყობინებას და შემდგომ – წინა ცვლილებას, მივიღებთ მონაცემის არასწორ მდგომარეობას, რაც ძალიან დიდი ფუნქციონალურ და რეპუტაციულ რისკებთან არის დაკავშირებული.

7.4. ადმინისტრირების მოდული

სისტემის ადმინისტრირებისათვის გვაქვს 2 დამოუკიდებელი მოდული:

- პირველი, რომელიც გამოიყენება უშუალოდ სისტემის მიმწოდებელი, პროვაიდერი კომპანიის მიერ და წვდომადია მხოლოდ სისტემის მიმწოდებელი კომპანიისათვის და
- მეორე, ადმინისტრირების მოდული მომხმარებელი კომპანიებისათვის, საკუთარი ბიზნესის სამართავად.

7.4.1. მიმწოდებლის ადმინისტრირების მოდული

სისტემის ადმინისტრირების მოდულიდან ხორციელდება კლიენტი კომპანიების მართვა და მასთან დაკავშირებული ფუნქციონალი.

კომპანიების ადმინისტრირების მოდული ხელმისაწვდომია მხოლოდ სისტმის მიმწოდებელი პროვაიდერისათვის და წვდომადია VPN (Virtual Private Network) -ის საშუალებით.

კომპანიების ადმინისტრირების მოდულს აქვს ცალკე მდგომი, იზოლირებული CompanyAdministration.Identity მიკროსერვისი, რომლის გავლითაც, Api Gateway web სერვისიდან, ადმინისტრირების მოდულში ხდება სისტემის მიმწოდებლის წარმომადგენლობის აუტენტიფიკაცია/ავტორიზაცია.

7.4.1.1. კლიენტი კომპანიის რეგისტრაციის პროცესი

მას შემდეგ, რაც კომპანია მისთვის სასურველი პირობით და პაკეტის ზომით შეიძენს საკუთარი ბიზნესის მართვის ფუნქციონალს – როგორც სერვისი, კომპანიის რეგისტრირება ხდება პლატფორმის მიმწოდებლი სისტემის მხარეს კომპანიების მონაცემთა ბაზაში.

თითოეული კომპანიის ჭრილში ხდება კომპანიის დეტალების: *საიდენტიფიკაციო ნომრის, გაფორმებული ხელშეკრულების, პაკეტის ზომის და კომპანიის წარმომადგენელი საკონტაქტო პირების* შესახებ ინფორმაციის განსაზღვრა

(შენახვა). კომპანიის რეგისტრაცია ხორციელდება Company მიკროსერვისის გავლით და წარმატებით რეგისტრაციის შემთხვევაში კომპანიის წარმომადგენლობას ეგზავნება SMS/Email შეტყობინებები მხარეთა შორის გაფორმებული კონტრაქტის და გამოწერილი პირობების დეტალური ინფორმაცის შესახებ.

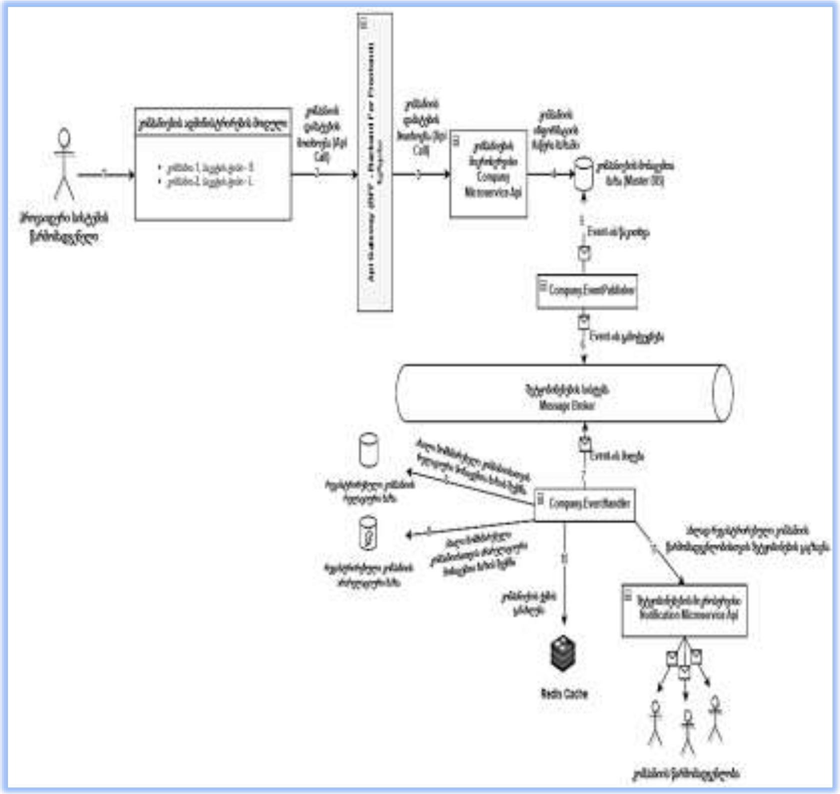
Company მიკროსერვისი წვდომადია მხოლოდ პლატ-ფორმის მიმწოდებელი პროვაიდერისათვის და შედგება შემდეგი კომპონენტებისაგან: Company Web Service, Company.BackgroundTasks Windows Service, Company.EventPublisher Windows Service, Company.EventHandler Windows Service.

სისტემის მიმწოდებელი პროვაიდერის ადმინისტრი-რების მოდულს აქვს იზოლირებული Api Gateway web სერვისი (BFF– Backend For Frontend), რომლის გავლითაც, კომპანიების ადმინისტრირების პორტალი (SPA– Single Page Application) ახდენს კომუნიკაციას Company მიკროსერვისთან.

BackgroundTasks Windows Service უზრუნველყოფს პერიოდული სამუშაოების (Jobs) შესრულებას, რომელიც ამოწმებს კომპანიის მიერ შეძენილი პაკეტის აქტიურობის პირობებს და, საჭიროებისამებრ, ავტომატურად უცვლის კომპანიას აქტიურობის სტატუსს. მის მიხედვით ხდება სისტემის მიწოდება მომხმარებლისათვის, კონკრეტული კლიენტი კომპანიის ჭრილში.

კომპანიის რეგისტრაციის პროცესში, იმისათვის რომ ინფრასტრუქტურის ადმინისტრატორებს არ მოუწიოთ სკრიპტების და ტექნიკური გარემოს ხელით გამართვა

(Manual Deployment), სისტემაში ავტომატურად ხდება (ტრიგ-ერდება) კლიენტის შესაბამისი მონაცემთა ბაზის გენერირება როგორც რელაციური, ასევე არარელაციური მონაცემთა ბაზების დონეზე (CQRS მიდგომის რეალიზაციისათვის, ElasticSearch სერვისში ხდება შესაბამისი მონაცემთა ინდექსების შაბლონის შექმნა – Indices) [106]. პროცესი ხორ-ციელდება ასინქრონულად (ნახ.7.6).



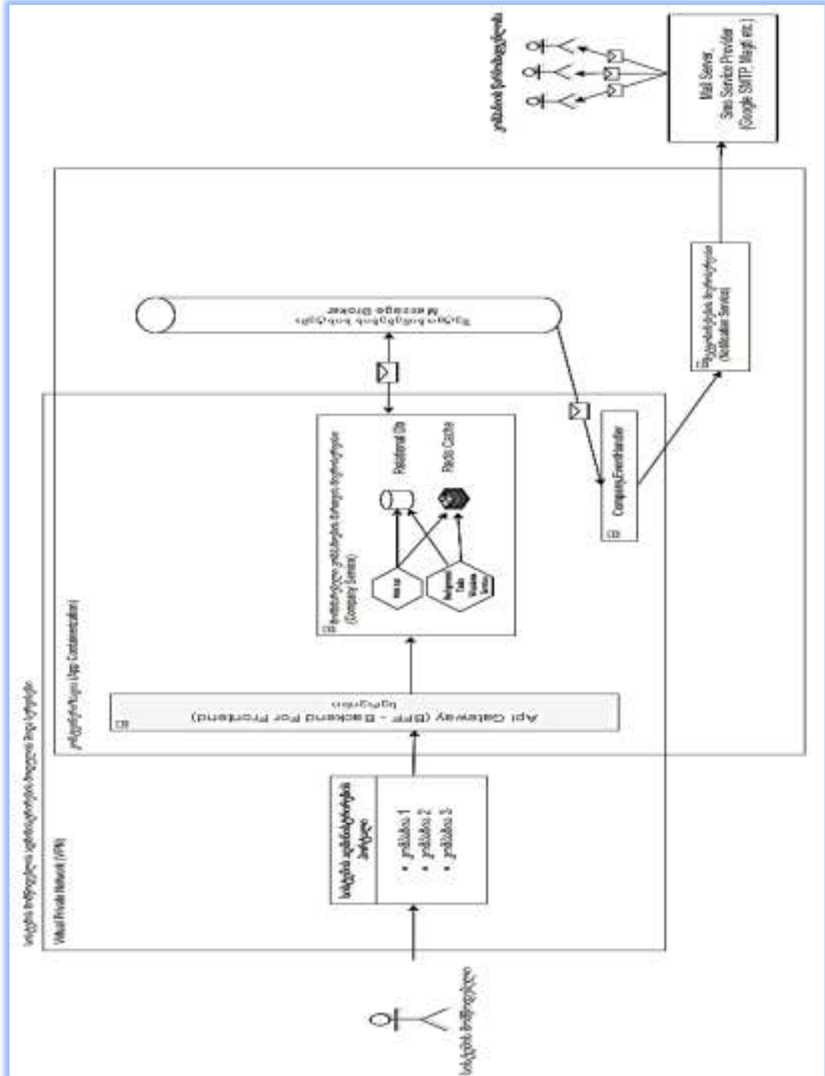
ნახ. 7.6. ახალი კლიენტი კომპანიის დამატების პროცესი

სისტემის მიმწოდებელი პროვაიდერის ადმინისტრირების მოდულიდან ხდება კლიენტი კომპანიების შეტყობინებების მართვა. გამოწერილი პაკეტის ვალიდურობის ამოწურვიდან გარკვეული პერიოდით ადრე, სისტემის პროვაიდერი მომხმარებელი კომპანიების წარმომადგენლობას უგზავნის eMail/Sms შეტყობინებას გამოწერის (Subscription) განახლების ან გაუქმების შესახებ.

შეტყობინების გაგზავნა ხდება ყოველდღიური ღამის სამუშაოს (job-ის) შესრულების საფუძველზე შემდეგნაირად:

„ყოველ ღამით ეშვება სამუშაო, რომელიც ამოწმებს გამოწერილი პაკეტის ვალიდურობის პირობებს და იმ შემთხვევაში თუ კომპანიის მხრიდან არ მომხდარა ისეთი გამოწერის განახლება, რომელსაც 4 კვირის განმავლობაში ეწურება ვალიდურობის ვადა, კლიენტი კომპანიის საკონტაქტო პირებს ეგზავნებათ შეტყობინება გამოწერის განახლების ან, სურვილისამებრ, შეწყვეტის შესახებ“ (ნახ.7.7).

ზემოაღწერილი ჯობი ეშვება Company.BackgroundTasks Windows სერვისის ჭრილში. გამოწერის განახლება მომხმარებელ კომპანიას შეუძლია კომპანიების მართვის ადმინისტრირების მოდულიდან, რომლის შედეგადაც ხდება კომპანიების კეშის განახლება.



ნახ. 7.7. კომპანიების წარმომადგენლობის ინფორმირება სისტემის მიმწოდებელი კომპანიის მხრიდან

7.4.1.2. კლიენტი კომპანიის ადმინისტრირების მოდული

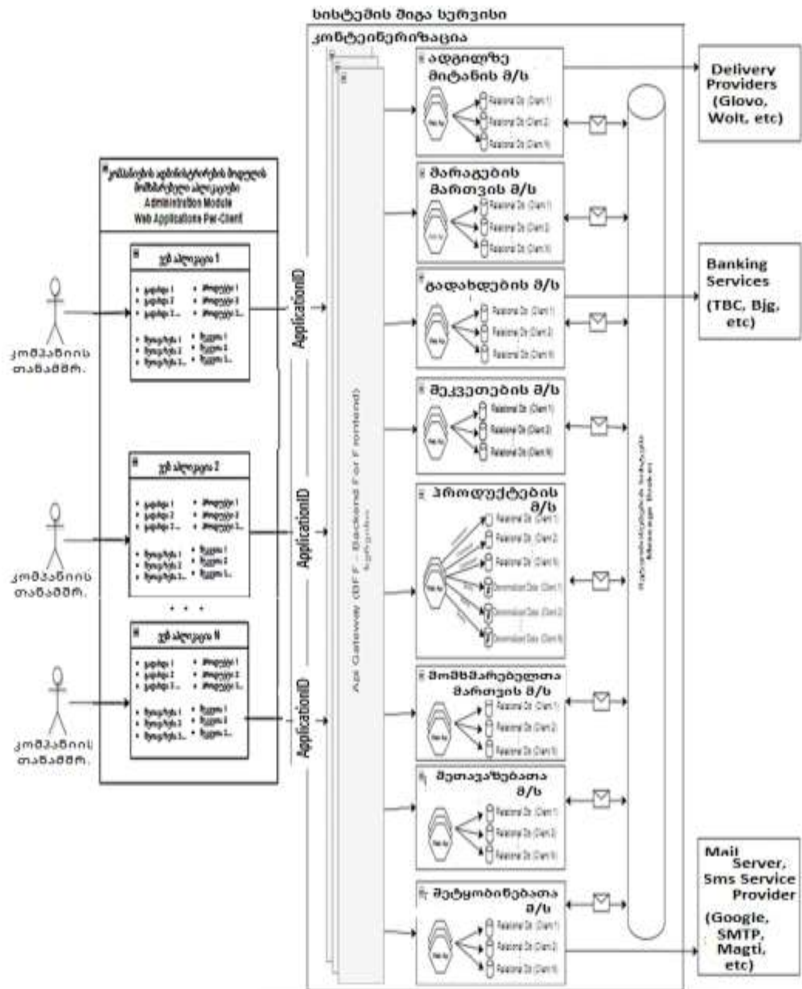
კლიენტი კომპანიის თანამშრომლებს აქვთ ცალკე მდგომი მოდული საკუთარი ბიზნესის ადმინისტრირებისათვის. ისევე, როგორც მომხმარებლის პორტალის შემთხვევაში, ადმინისტრირების მოდულის შემთხვევაშიც, სისტემის მიმწოდებელი პროვაიდერი სრულად უზრუნველყოფს კომპანიის ჭრილში ადმინისტრირების პორტალის კომპანიისთვის სასურველ ბრენდინგზე მორგებას. ადმინისტრირების მოდულს აქვს ცალკე მდგომი მომხმარებლების მართვის ფუნქციონალი, რომელსაც არ აქვს თანაკვეთა სისტემის გარე მომხმარებლებთან და უზრუნველყოფს კომპანიის თანამშრომლების იდენტიფიცირებას.

ამისათვის, ადმინისტრირების მოდულში გვაქვს იზოლირებული Administration.Identity მიკროსერვისი, რომლის საშუალებითაც ხდება კომპანიის თანამშრომლების სისტემაში რეგისტრაცია და მათთვის შესაბამისი უფლებების განსაზღვრა უშუალო მენეჯერული რგოლის მიერ.

კომპანიის თანამშრომლის ავტორიზაციის პროცესი ტექნიკურად ხორციელდება იგივე პრინციპით, როგორც მომხმარებლის პორტალის შემთხვევაში. ადმინისტრირების მოდულში უფლებების მართვა ბევრად კომპლექსურია ვიდრე მომხმარებლებისთვის განკუთვნილი პორტალის შემთხვევაში.

ადმინისტრირების მოდულში ავტორიზაციის შემდგომ, თანამშრომელს პორტალზე უჩანს მხოლოდ მისთვის განსაზღვრული ფუნქციონალი. კომპანიის თანამშრომლებს აქვთ განსხვავებული უფლებები, მათთვის საჭირო ფუნქციონალის მიხედვით. მარკეტინგის თანამშრომლებს აქვთ უფლება დაამატონ/დააკორექტირონ/დროებით გააუქმონ ან გააქტიურონ მათი კომპანიის შეთავაზებები, ხოლო საწყობის თანამშრომლებს აქვთ შესაბამისი უფლება, პროდუქციის აღწერის, მიღებისა და გაცემის ინფორმაციის განთავსებისათვის.

7.4.2.1. მომხმარებელი კომპანიების ადმინისტრირების მოდულის არქიტექტურული მოდელი



ნახ. 7.8. მომხმარებელი კომპანიების ადმინისტრირების მოდულის არქიტექტურული მოდელი

7.4.2.2. რეპორტინგის ფუნქციონალი

კომპანიების ადმინისტრირების მოდულში აქტიურად გამოიყენება *რეპორტინგის ფუნქციონალი.* რეპორტინგის მოქნილი ფუნქციონალი კომპანიას აძლევს საშუალებას, ნებისმიერ დროს აწარმოოს საკუთარი ბიზნესის მონიტორინგი და, სურვილისამებრ, დააგენერიროს სხვადასხვა ტიპის/პერიოდის *რეპორტი* (ამონაწერი) სისტემიდან.

Reporting მიკროსერვისი შედგება შემდეგი კომპონენტებისაგან: Reporting.WebApi ვებ სერვისი, Reporting.EventHandler Windows სერვისი, Reporting.EventPublisher Windows სერვისი, Reporting.BackgroundTasks Windows სერვისი.

რეპორტინგის ფუნქციონალი გამოირჩევა მაღალი დატვირთულობით. კომპანიის თანამშრომლებს ხშირად უწევთ რამდენიმე წლის, მილიონობით მონაცემის დამუშავება, იმისათვის, რომ მიიღონ სასურველი ინფორმაციის შემცველი რეპორტი.

გარდა თანამშრომლების მხრიდან მოთხოვნილი მყისიერი რეპორტინგის ფუნქციონალისა, სისტემა ავტომატურ რეჟიმში აგენერირებს რეპორტებს სხვადასხვა პერიოდის და ფუნქციონალის მიხედვით.

მაგალითად, ყოველთვიურად ხდება პროდუქციის რეალიზაციის სტატისტიკური მონაცემების დაჯამება და წინა პერიოდთან შედარება, რომლებიც გენერირდება სტატისტიკური დიაგრამების სახით და ავტომატურ რეჟიმში ეგზავნება კომპანიის მენეჯმენტის რგოლის წარმომადგენლობას.

გარკვეული პერიოდულობით, რეპორტების გენერირე-ბას უზრუნველყოფს Reporting.BackgroundTasks Windows სერვისი. კომპანიის მომხმარებლის მხრიდან, სასურველი პერიოდის რეპორტის დაგენერირება ხდება ადმინისტრი-რების პორტალიდან Reporting ვებ სერვისის (მიკროსერვისის) საშუალებით. კომპანიის თანამშრომლებისათვის განკუთ-ვნილ ადმინისტრირების მოდულს აქვს ცალკე მდგომი Api Gateway Web სერვისი (BFF), რომლის გავლითაც ხდება ადმი-ნისტრირების პორტალის მიკროსერვისებთან კომუნიკაცია.

იმ შემთხვევაში, თუ კომპანიის თანამშრომელს ჭირდება დიდი პერიოდის მონაცემების ანალიზი და დამუშავება რეპორტის დასაგენერირებლად, *სინქრონული* კომუნიკაციის გზით მოთხოვნის დამუშავება არ არის კარგი გამოსავალი. ამისათვის გვაქვს ალტერნატიული გადაწყვეტა, რომლის საშუალებითაც რეპორტის მყისიერად დაგენერირე-ბის ნაცვლად, კომპანიის თანამშრომელს შეუძლია რეპორტის შეკვეთა სასურველი პარამეტრებით და დასახელებით.

სისტემა შეინახავს რეპორტის მოთხოვნას და *ასინქრო-ნულად* განაგრძობს მის დამუშავებას. შეკვეთის დაფიქსი-რების შემდეგ, სისტემა ამუშავებს მოთხოვნას და მითითე-ბულ მეილზე თანამშრომელს ავტომატურად უგზავნის დაგენერირებულ რეპორტს. მეილზე გაგზავნასთან ერთად, სისტემა დაგენერირებულ ფაილს ინახავს ფაილ-სერვერზე, რათა ახლიდან მოთხოვნის შემთხვევაში არ დაგვჭირდეს დიდი მონაცემების თავიდან დამუშავება და ფაილის ახლიდან გენერირება. ადმინისტრირების პორტალში,

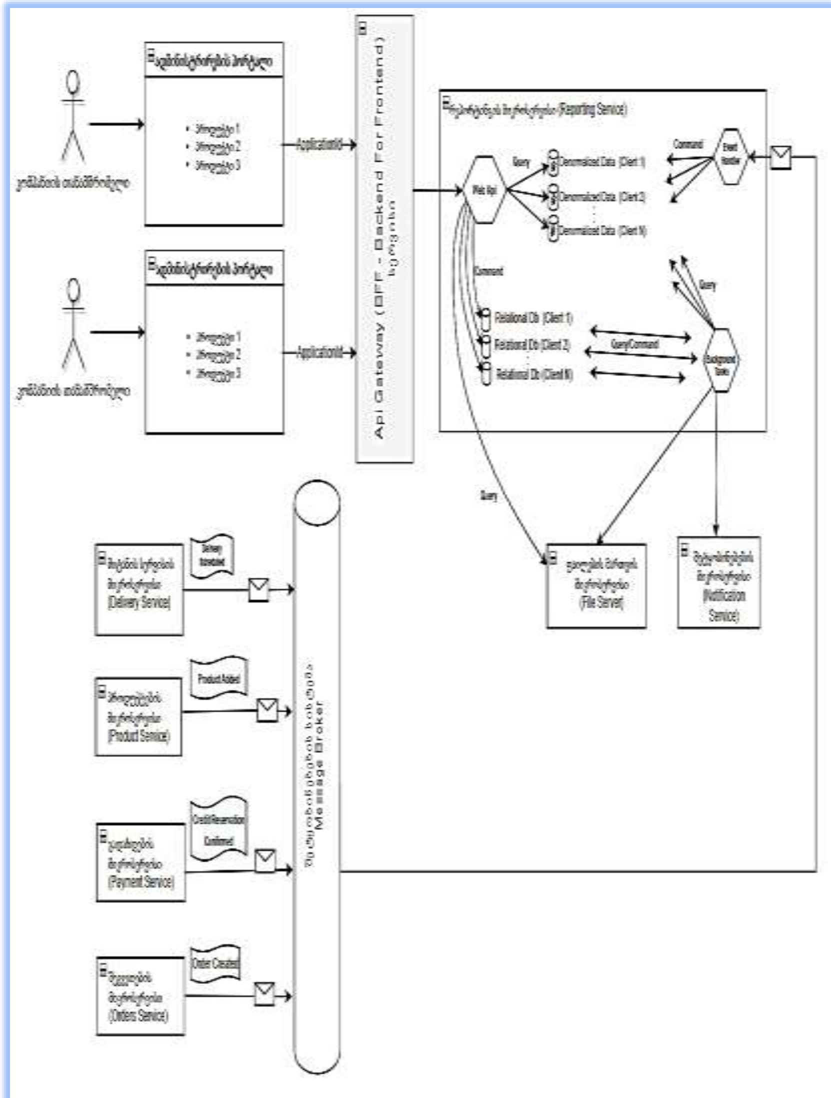
კომპანიის თანამშრომელს უჩანს მის მიერ მოთხოვნილი რეპორტების სია და სასურველ დროს შეუძლია ჩამოტვირთოს დაგენერირებული დოკუმენტი ფაილ - სერვერიდან.

რეპორტის მყისიერად დაგენერირება/ჩამოტვირთვის ფუნქციონალი ხელმისაწვდომია მხოლოდ გარკვეული პერიოდის – დასამუშავებლად ოპტიმალური რაოდენობის მონაცემების მოთხოვნის შემთხვევაში.

კომპანიის თანამშრომელს, მინიჭებული უფლებების მიხედვით, სისტემიდან სხვადასხვა ტიპის რეპორტის ამოღება შეუძლია, როგორებიცაა: პროდუქციის მიღება/ჩაბარების რეპორტი თარიღის და/ან პროდუქტის ჭრილში, პროდუქციის რეალიზაციის რეპორტი (თარიღის, პროდუქტის, სააქციო შეთავაზების ჭრილში), განთავსებული შეკვეთების რეპორტი შესაბამისი თარიღის და შეკვეთის სტატუსების მიხედვით, გადახდის სტატუსების და უკან დაბრუნებული თანხის ამონაწერის რეპორტი და მრავალი სხვ.

გამომდინარე იქიდან, რომ რეპორტინგის ფუნქციონალი პროცესინგის მხრივ დიდ რესურსებს მოითხოვს, საოპერაციო მონაცემთა ბაზის (ძირითადი მონაცემთა სამართავი მიკროსერვისების რელაციური მონაცემთა ბაზა) გამოყენება რეპორტინგის პროცესებისთვის არასწორი ტექნიკური გადაწყვეტაა, რადგან ამით საგრძნობლად იზრდება დატვირთვა საოპერაციო მონაცემთა ბაზებზე, რაც სისტემის წარმადობას დროთა განმავლობაში მკვეთრად გააუარესებს.

7.9 ნახაზზე მოცემულია რეპორტინგის მიკროსერვისის საილუსტრაციო სქემა.



ნახ. 7.9. რეპორტინგის მიკროსერვისი

კონკრეტულ მოცემულობაში მოქნილად ვიყენებთ მიდგომას – სახელწოდებით CQRS [106]. რეპორტინგის მიკროსერვისი შეტყობინებების Message Broker სისტემის საშუალებით ახდენს ძირითადი სამართავი (Master) სერვისების მიერ გამოქვეყნებული მოვლენების (Event) დამუშავებას და შენახვას არარელაციურ (NoSQL) მონაცემთა ბაზაში *დენორმალიზებული* სახით (ჩვენ შემთხვევაში გამოყენებული ტექნოლოგიაა ElasticSearch, თუმცა ზოგადი სისტემის არქიტექტურა მორგებადია ნებისმიერ სხვა ტექნოლოგიაზე, რომელიც ორიენტირებულია მონაცემთა ოპტიმალურ წაკითხვა/დამუშავებაზე) [98, 102].

7.5. მეშვიდე თავის დასკვნა

წიგნის ბოლო თავში წარმოდგენილია ჩვენ მიერ შემუ-შავებული *სისტემის ადმინისტრირების ფუნქციონალი* და შესაბამისი *არქიტექტურული მოდელი.* გადმოცემულია დაპროექტებული სისტემის მონაცემთა საცავების აგების მეთოდოლოგია მიკროსერვისული არქიტექტურის ბაზაზე. აღწერილია სისტემის დინამიკური, ჰორიზონტალური მასშტაბირების სტრატეგიები და ძირითადი მიდგომები, სისტემის ფუნქციონირების დროს მაღალი დატვირთვის პირობებში.

დასკვნა

ჩატარებული კვლევებისა და ექსპერიმენტული საპრო-ექტო სამუშაოების პროგრამული რეალიზაციის საფუძველზე Web-აპლიკაციებისა და მიკროსერვისული დინამიკური არქიტექტურების დეველოპმეტის საკითხებზე შეიძლება ჩამოვაყალიბოთ შემდეგი კონკრეტული დასკვნები:

1. ნაშრომში გაანალიზებული და კლასიფიცირებულია პროგრამული აპლიკაციების შექმნის მეთოდები, მეთოდო-ლოგიები და ინსტრუმენტული საშუალებები კომპანია Microsoft-ის პროგრამული პლატფორმების საფუძველზე: ASP.NET Web Forms, ASP.NET MVC და Silverlight. გამო-კვეთილია დაპროგრამების ტექნოლოგიების ძირითადი მახასიათებლები, მათი დადებითი და უარყოფითი მხარეები. შემოთავაზებულია რეკომენდაციები – რეალური ამოცანების გადაწყვეტისას პროგრამირების მისაღები ტექნოლოგიის გამოყენების მიზანშეწონილობის შესახებ;

2. გამოყენებითი პროგრამული უზრუნველყოფის არქიტექტურის შემუშავების მიზნით, დაპროექტების ეტაპზე ზოგადი სტანდარტების გათვალიწინებით, Ms_Silverlight ფრეიმვორკის სამუშაო გარემოში შემუშავდა მდიდარი (მრავალფუნქციური) ინტერნეტ აპლიკაციის (RIA) არქიტექ-ტურა, რომელიც გულისხმობს MVVM სტანდარტზე აგებულ პროგრამულ ლოგიკას. მის საფუძველზე განისაზღვრა აპლი-კაციის ლოგიკურ კომპონენტებს შორის კომუნიკაციები;

3. არქიტექტურული სტანდარტების გამოყენება მაღალი ხარისხის შედეგს იძლევა Silverlight პროექტებში,

სერვისის დონეზე რეალიზებულია WCF ტექნოლოგიისა და პროქსი კლასებით. ასინქრონული მეთოდების გამოძახებები და Command არქიტექტურა წარმატებით მუშაობს პრეზენტაციის დონეზე. ასევე, დადებით შედეგს იძლევა კომპოზიტური გვერდების გამოყენება (ინდივიდუალური View გვერდების გაერთიანება კომპოზიტურ View-ში);

4. სამომხმარებლო გარემოს გამდიდრება Silverlight პლატფორმის კონტროლებით. Silverlight ტექნოლოგიის მნიშვნელოვანი უპირატესობაა მონაცემებთან ბმის მძლავრი მექანიზმი და კონვერტაციის საშუალება. Value Converter ობიექტი სამომხმარებლო ინტერფეისზე გამოტანილი ინფორმაციის ფორმატირებისა და ტრანსფორმირების საშუალებას იძლევა. აგრეთვე, რეპორტების მოდულის ჩაშენება პროგრამულ აპლიკაციაში, მონაცემთა უსაფრთხოების მიზნით მხოლოდ ავტორიზებული მომხმარებლების დაშვება ინფორმაციაზე, ანგარიშგებათა ექსპორტირება სხვადასხვა ფაილურ ფორმატში და დოკუმენტების გენერაცია შაბლონის საფუძველზე – მეტად მნიშვნელოვანი ფუნქციებია;

5. Silverlight ტექნოლოგიით შესაძლებელია სამომხმარებლო ინტერფეისის სრულყოფა ვებაპლიკაციებში *ანიმაციების* დამატებით. Expression Blend ინსტრუმენტში გათვალისწინებულია ანიმირებისა და ტრანსფორმაციის ეფექტების შექმნა, შეიცავს შესაბამის ბიბლიოთეკებს, რომლებიც მემკვიდრეობით მოდის Timeline კლასიდან. იგი განთავსებულია System.Windows.Media.Animation ბიბლიოთეკაში.

6. განხორციელდა მცირე და საშუალო ბიზნესის მენეჯმენტის პროცესების ანალიზი, პრობლემების გამოვლენა და მათი გადაჭრის გზების მოძიება ახალი ციფრული ტექნოლოგიების, პროგრამული პლატფორმების და სისტემის სუფთა არქიტექტურის საფუძველზე;

7. შემუშავებული ინტეგრირებული ბიზნეს-ეკოსისტემა დინამიკური სტრუქტურაა და წარმოადგენს მოქნილ გადაწყვეტას მცირე და საშუალო ზომის კომპანიების ერთობლივი მოღვაწეობისა და განვითარებისათვის, სწრაფადცვლად გარემოში. განხორციელდა ტექნიკური გადაწყვეტილებების მორგება ისეთი სისტემის მოდელზე, რომელიც პრაქტიკულად გაამარტივებს კვლევის ობიექტის ანუ დამკვეთის წარმომადგენლების საქმიანობას;

8. შეიქმნა ტექნოლოგიებისგან დამოუკიდებელი კორპორატიული აპლიკაციის (Technology Agnostic, Enterprise) არქიტექტურული მოდელი, რომლის პროგრამული რეალიზაციის შემდგომ, ბიზნესის წარმომადგენლობას ექნება საშუალება სისტემა მოიხმაროს როგორც სერვისი და ტექნიკური ცოდნის და პროგრამული აპლიკაციის შექმნა/განვითარების გარეშე, მცირე დანახარჯებით, ავტომატიზებულ რეჟიმში მართოს საკუთარი ბიზნესი;

9. აგებული სისტემის ინფრასტრუქტურა მორგებულია კომპანიის ბრენდინგზე და მარკეტინგული ბიზნესმოთხოვნების თვალსაზრისით, საკმაოდ მოქნილი და ეფექტიანი გადაწყვეტაა არატექნიკური კომპეტენციის მქონე

კომპანიებისათვის. კონკრეტული ბიზნესპროცესების თავისებურებებზე დაყრდნობით, გაანალიზებულ იქნა სინქრონული და ასინქრონული პროცესების მართვის სტრატეგიები, წარმოდგენილია შესაბამისი ალგორითმული სქემები და სტრუქტურები, აგრეთვე მათი გამოყენების რეკომენდაციები და საილუსტრაციო მაგალითები;

10. მიზნობრივი სისტემის ბიზნესპროცესების მართვის სრულყოფისათვის შემუშავებულია მიკროსერვისული დინამიკური არქიტექტურა, მისი კომპლექსურობის კონკრეტული ასპექტები და პრაქტიკული მაგალითების საფუძველზე ნაჩვენებია რეკომენდებული გადაწყვეტის გზები;

11. მიკროსერვისული სისტემის კომპონენტებს შორის საზღვრებისა და მათი ლოგიკური დეკომპოზიციისათვის, შესაბამისი პროგრამული უზრუნველყოფის დასაპროექტებლად გამოყენებულია დომენ-ორიენტირებული დიზაინის (DDD) მიდგომა. შემუშავებულია დასმული პრობლემების მოქნილად გადაწყვეტის გზები დისტრიბუციული სისტემების არქიტექტურის შემუშავების პროცესში;

12. მცირე და საშუალო ბიზნესის მართვის ციფრული პლატფორმის დინამიკური არქიტექტურის დაპროექტებისას, ნაშრომში ყურადღება გამახვილდა სისტემის კომპონენტების და მონაცემთა დეცენტრალიზებულად მართვის მოქნილ სტრატეგიებზე, რათა მომხმარებელი კომპანიების რაოდენობის ზრდასთან ერთად, სისტემა გამოირჩეოდეს მდგრადობით, მაღალი წვდომადობით, მონაცემთა დამუშავების სისწრაფით და მოქნილი მასშტაბირებადობით.

გამოყენებული ლიტერატურა

1. ჩოგოვაძე გ., ფრანგიშვილი ა., სურგულაძე გ., მართვის საინფორმაციო სისტემების დაპროგრამების ჰიბრიდული ტექნოლოგიები და მონაცემთა მენეჯმენტი. ISBN 978-9941-20-790-7. სტუ. „ტექნიკ.უნივერსიტეტი“, თბ., 2017. -1001 გვ

2. სურგულძე გ., ქრისტესიაშვილი ხ., სურგულაძე გიორგი. საწარმოო რესურსების მენეჯმენტის ბიზნეს-პროცესების მოდელირება და კვლევა. ISBN 978-9941-20-557-6. სტუ. „ტექნიკ.უნივერსიტეტი“, თბ., 2015 -216 გვ.

3. სურგულაძე გ., პაპავაძე ს., მაჩალაძე ო. ინფორმაციული საზოგადოება და ინფორმატიკის დიდაქტიკა. ISBN 978-9941-8-5443-9. მონოგრაფია. სტუ. „IT-კონსალტინგ.სამეცნ. ცენ-ტრი”, თბ., 2023. –260 გვ

4. სურგულაძე გ., პეტრიაშვილი ლ. კორპორაციული მართვის სისტემების პროგრამული დეველოპმენტი (WCF/WPF, SOA). ISBN 978-9941-8-2725-9. საკურსო პროექტის მეთოდური მითითებანი. სტუ-ს „IT-კონსალტინგის სამეცნიერო ცენტრი”, თბ., 2021. -65 გვ.

5. ჩოგოვაძე გ., სურგულაძე გ., გულიტაშვილი მ., დოლიძე ს. პროგრამული აპლიკაციების ხარისხის მართვა: ტესტირება და ოპტიმიზაცია. ISBN 978-9941-20-629-2. სტუ. „IT-კონსალტინგ ცენტრი“. თბ., 2020. -363 გვ.

6. www.seguetech.com/blog/2013/06/07/desktop-vs-web-applications-deeper-comparison გადამოწმ. 22.05.16.

7. https://webapphuddle.com/web-application-vs-website-design-a-fundamental-difference/ გადამოწმ. 22.05.16.

8. http://www.seguetech.com/blog/graphic/2013/04/16/website-vs-web-application-difference გადამოწმ. 22.05.16.

9. Lair R. Beginning Silverlight 4 in C# ISBN-13: 978-1-4302-2988-9, Copyright © 2010

10. Moroney L. Microsoft® Silverlight® 4 Step by Step. ISBN: 978-0-735-63887-7, Copyright © 2010

11. Cleeren G., Dockx K. Microsoft Silverlight 5 Data and Services Cookbook. ISBN 978-1-84968-350-0 Copyright © 2012.

12. Maan J., Mantha N. Rich Internet Applications, Platforms and Tools - A Paradigm Shift in Web User Experience. The Fourth Intern. Conf. "Computer Science, Engineering and Applications". 2014, 4(3): pp. 121-129, DOI:10.5121/csit.2014.4312. https://www.researchgate.net/publication/269198905_Rich_Internet_Applications_Platforms_and_Tools_-_A_Paradigm_Shift_in_Web_User_Experience. (13.07.24)

13. Churchill A.S., Henderson D. Rich Internet Applications Using SAS/IntrNet® and Microsoft Silverlight. Henderson Consulting Services, Colorado-Olney. 2008. https://support.sas.com/resources/papers/proceedings/pdfs/sgf2008/395-2008.pdf (13.07.24)

14. სურგულაძე გ., პეტრიაშვილი ლ. ვიზუალური დაპროგრამება C# ენის ბაზაზე ინფორმაციულ სისტემებისათვის (Visual StudioNET 2019 პლატფორმაზე). ISBN 978-9941-8-1708-3. სტუ. „IT-კონსალტინგ ცენტრი“. თბ., 2019. -200 გვ.

15. Adding UI for Silverlight to Visual Studio 2015 Toolbox. Copyright © 2024 Progress Software Corporation. Progress. https://docs.telerik.com/devtools/silverlight/integration/installation-adding-to-vs-toolbox-silverlight (20.05.24)

16. Walkthrough: Create a Button by Using Microsoft Expression Blend. 2022. https://learn.microsoft.com/en-us/dotnet/desktop/wpf/controls/walkthrough-create-a-button-by-using-microsoft-expression-blend?view=netframeworkdesktop-4.8 (20.05.24)

17. Galloway J., Wilson B., Scott K.A., Matson D. Professional ASP.NET MVC 5. ISBN 978-1118794753 Copyright © 2014, John Wiley & Sons, Inc.

18. Freeman A. Pro ASP.NET MVC 5. ISBN 978-1430265290 Copyright © 2013, Apress.

19. https://msdn.microsoft.com/en-us/silverlight/bb187358.aspx (11.04.24)

20. Microsoft.NET Microservices: Architecture for Containerized .NET Applications. https://raw.githubusercontent. com/dotnet-architecture/eBooks/master/current/microservices/ NET-Microservices-Architecture-for-Containerized-NET-Applications.pdf 2020. (20.05.24)

21. სურგულაძე გ., მღებრიშვილი გ. საფინანსო ორგანიზაციის მართვა მიკროსერვისული არქიტექურის გამოყენებით. სტუ შრ.კრ. „მას“ N2(29), 2019. გვ.117-123

22. Monolithic architecture vs microservices choosing the right architecture for the project. 2019. https://ka.mort-sure.com/

blog/monolithic-architecture-vs-microservices-choosing-the-right -architecture-for-the-project-288872

23. სურგულაძე გ., ფხაკაძე ც., კეკენეძე ა. ორგანიზაციული მართვის ბიზნესპროცესების მოდელირება და დაპროექტება. ISBN 978-9941-0-8259-7. სტუ. „IT-კონსალტინგ ცენტრი". თბ., 2016. -204 გვ.

24. ფრანგიშვილი ა., სურგულაძე გ., ვაჭარაძე ი. ბიზნეს-პროგრამების ექსპერტულ შეფასებებში გადაწყვეტილებათა მიღების მხარდამჭერი მეთოდები და მოდელები. ISBN 978-9941-14-450-9. სტუ. „ტექნიკ.უნივერს.", თბ., 2009 -200 გვ.

25. Martin R. Clean Architecture: A Craftsman's Guide to Software Struc¬tu¬re and Design. 2017

26. Pavlutin, D. Clean Architecture and Design: Understanding The Basics. 2020

27. Introduction.Welcome to the OWASP API Security Top 10 - 2023. https://owasp.org/API-Security/editions/2023/en/0x03-introduction/ (25.06.24)

28. Microsoft Corporation. Microsoft Application Architecture Guide. ISBN: 9780735627109 Copyright © 2009. USA

29. Meier J.D., Vasireddy S., Babbar A., Mackman A. Improving .NET Application Performance and Scalability. ISBN 0-7356-1851-8 © 2004 Microsoft Corporation

30. სურგულაძე გ., კივილაძე ნ., კივილაძე გ. კორპორაციული აპლიკაციების აგება დაპროგრამების სერვის-ორიენტირებული ტექნოლოგიით. სტუ-ს შრ.კრ. „მას". No 1(21), თბილისი, 2016. გვ.215-220

31. კივილაძე ნ. პროგრამული დანართის არქიტექტურა და RIA-აპლიკაციების დაპროექტება. სტუ-ს შრ.კრ. „მას“. No 1(21), თბილისი, 2016. გვ.243-248

32. სურგულაძე გ., გულიტაშვილი მ., ჩერქეზიშვილი. Web აპლიკაციების დამუშავების პროცესის მოდელირება UML/2 ტექნოლოგიით. Intern. Science Conf.“ Automated Control Systems & new IT”, 20-22 May. GTU, Tbilisi, 2011

33. Thomas Erl. Service-Oriented Architecture (SOA): Concepts, Technology, and Design. ISBN 007-6092038498. CopyR-© 2005.

34. Douglas K. Barry. Web Services, Service-Oriented Architectures, and Cloud Computing, Second Edition: The Savvy Manager's Guide (The Savvy Manager's Guides) 2nd Edition. ISBN 978-0123983572. Copyright © 2013.

35. Marks E.A. Service-Oriented Architecture (SOA): A Planning and Implementation Guide for Business and Technology 1st Edition. ISBN 858-0000163933. Copyright © 2006.

36. Dino Esposito, Andrea Saltarello. Architecting Applications for the Enterprise, Second Edition. ISBN: 978-0-7356-8535-2. Copyright © 2014

37. არქიტექტურული სტანდარტები http://www.codeproject.com/Articles/430590/Design-Patterns-of-Creational-Design-Patterns. (21.05.24)

38. http://www.dofactory.com/net/design-patterns. (21.05.24)

39. http://www.go4expert.com/articles/design-pattern-simple-examples-t5127/ (30.05.24)

40. ლომინაძე თ., ჟვანია თ., პეტრიაშვილი ლ. NoSQL მონაცემთა ბაზის ავტომატური კლასტერირება ხელოვნური ინტელექტის ალგორითმის გამოყენებით. საერთაშ. სამეც.-პრაქტ.კონფ. „*ინოვაციები და თანამედროვე გამოწვევები*" შრ.კრ., თბ., 2022, 18–19 ნოემბ. DOI.org/10.36073/1512-3979

41. სურგულაძე გ., ჟვანია თ., პეტრიაშვილი ლ., კაპანაძე დ. კომპიუტინგის ფაკულტეტის სტრუქტურის და აკადემიური პროცესის მოდელის ევოლუციური განვითარება. სტუ შრ.კრ. „მას" N1(37), 2024. გვ.9-16. DOI.org/10.36073/1512-3979

42. Gasitashvili Z., Kiknadze M., Zhvania T., Kapanadze D. Factors Affecting Sustainable Development and Modelling. Publisher: Springer International Publishing Published in: Recent Developments in Mathematical, Statistical and Computational Sciences, 2021, pp. 669-679

43. Todua T., Kiknadze, Zhvania T., Kapanadze D. Pattern recognition reliability prediction for compact patterns. Materials of International scientific practical conference: Modern trends in science and education. Sofia, 2021, pp.68-73

44. Gary McLean Hall, Pro WPF and Silverlight MVVM, Effective Application Development with Model-View-ViewModel. ISBN-13 (pbk): 978-1-4302-3162-2 Copyright © 2010

45. Anderson Ch. Pro Business Applications with Silverlight 4. ISBN-13 (pbk): 978-1-4302-7207-6, Copyright © 2010

46. სურგულაძე გ., თოფურია ნ., ბერულავა ა. პროგრამული პროდუქტების დეველოპმენტი. ISBN978-9941-8-3810-1. სტუ-ს „IT-კონსალტინგ სამეცნიერო ცენტრი". თბ.,2022, 250 გვ

47. სურგულაძე გ., კივილაძე ნ. Web-აპლიკაციაში რეპორტების ინტეგრაციის მეთოდები. VII საერთაშორისო სამეცნიერო და პრაქტიკული კონფერენცია "ინტერნეტი და საზოგადოება" (INSO2015). ქუთაისი. 2015. გვ.92-96

48. Robin Dewson, Beginning SQL Server for Developers. ISBN-13: 978-1430237501 Copyright © 2012.

49. Krishnaswamy J. Learning SQL Server Reporting Services 2012. ISBN 978-1-84968-992-2 Copyright © 2013.

50. Brimhall J., Dye D., Gennick J., Roberts A. Wayne Sheffield, SQL Server 2012 T-SQL Recipes. ISBN13: 978-1-4302-4200-0. Copyright © 2012.

51. https://visualstudiomagazine.com/articles/2013/07/23/async-actions-in-aspnet-mvc-4.aspx. (27.05.24)

52. სურგულაძე გ., ბულია დ., თურქია ე. Web-აპლიკაციების დამუშავება მონაცემთა ბაზების საფუძველზე (ADO.NET, ASP.NET, C#). ISBN 978-9941-14-289-5. სტუ. „ტექნიკ.უნივერს.", თბ., 2009 -189 გვ.

53. http://csb.gov.ge/uploads/gaertianebuli.pdf (12.06.24)

54. http://www.csb.gov.ge/uploads/HR_Standard_in_English.pdf (12.06.24)

55. http://www.csb.gov.ge/uploads/HRM_Systems_in_Civil_Service._geo.pdf (14.06.24)

56. ადამიანური რესურსების მართვის ავტომატიზებული სისტემის მომხმარებლის სახელმძღვანელო. საქ.ფინანსთა სამინისტრო, 2013. http://www.fas.ge/images/file/eHRMS_Manual.pdf (16.06.24)

57. Bychkov D. Desktop vs. Web Applications: A Deeper Look and Comparison. 2013. http://www.wikiwand.com/en/Thin_client#/overview (20.05.24)

58. სურგულაძე გ., კორპორაციული მენეჯმენტის სისტემების Windows დეველოპმენტი (WCF ტექნოლოგია). ISBN 978-9941-0-7878-1. სტუ. „IT-კონსალტ. ცენტრი“. თბ.,2015. -154 გვ.

59. სურგულაძე გ., კორპორაციული მენეჯმენტის სისტემების Windows დეველოპმენტი (WF ტექნოლოგია). ISBN 978-9941-0-7520-9. სტუ. „IT-კონსალტ.ცენტრი“. თბ.,2015. -136 გვ.

60. სურგულაძე გ., კორპორაციული მენეჯმენტის სისტემების Windows დეველოპმენტი (WPF ტექნოლოგია). ISBN 978-9941-0-7103-4, 978-9941-0-7104-1. სტუ. „IT-კონსალტ.ცენტრი“. თბ.,2014. -202 გვ.

61. Gorbatov V.A. Theory of Ordered Systems. "Sovetskoe Radio", -Moscow, 1976. 336 p.

62. ჩოგოვაძე გ., სურგულაძე გ., ქაჩიბაია ვ. შესავალი მონაცემთა ბაზების სისტემების თეორიაში (სისტემების ზოგადი თეორია). საქ. პოლიტექნიკური ინსტიტუტი, „გამომც. სპი", თბ., 1985, 60 გვ.

63. სურგულაძე გ. კორპორაციის ავტომატიზებული სამუშაო ადგილების ქსელის აგების ტექნოლოგია (პირველი ქართული ERP სისტემა). ISBN 978-9941-8-5109-4, მონოგრაფია. სტუ-ს „IT-კონსალტინგ სამეცნიერო ცენტრი". თბ., 2023, - 331 გვ.

64. სურგულაძე გ., გულუა დ., კახელი ბ. პროგრამული აპლიკაციების აგება ვირტუალიზაციის პირობებში. ISBN 978-9941-8-0627-4. სტუ. „IT-კონსალტინგ ცენტრი“. თბ.,2019. 159 გვ.

65. Richardson Ch. 2020. https://microservices.io/ (20.04.24)

66. Domain-Driven Design Tackling Complexity in the Heart of Software. https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/032112¬ 5215. (14.02.23)

67. გიუტაშვილი მ., თურქია ე. კორპორაციულ სისტემებში ინტელექტუალური რესურსების მენეჯმენტი, მონოგრაფია. სტუ, თბ., 2008.

68. გიუტაშვილი მ. ორგანიზაციული მართვის სისტემის სრულყოფა BI ტექნოლოგიით. სტუ-ს შრ.კრებ. „მართვის ავტომატიზებული სისტემები“, №2(3), 2007. გვ. 73-78

69. Feras Taleb. Microservices Architecture: To Be Or Not To Be. 2019. https://feras.blog/microservices-architecture-to-be-or-not-to-be/ (05.06.24)

70. Rag Dhiman. Microservices Architecture. 2015. https://app.pluralsight.com/course-player?clipId=11b9a9b1-e922-48c0-9041-1d2ffdca3d9c (17.09.23)

71. სურგულაძე გ., გულუა დ. ქსელური არქიტექტურა ბიზნესისთვის. ISBN 978-9941-0-9842-0. სტუ. „IT კონსალტ. ცენტრი”, თბ., 2017. -270 გვ.

72. თურქია ე. ბიზნესპროექტების მართვის ტექნოლოგიური პროცესების ავტომატიზაცია. ISBN 978-9941-14-784-5. სტუ. „ტექნიკური უნივერსიტეტი“, თბ., 2010. -311 გვ.

73. სურგულაძე გ., კაკაშვილი გ., მარტიაშვილი გ. მობილური აპლიკაციების დეველოპმენტის საფუძვლები (Java, Android). ISBN978-9941-8-2488-3. სტუ. „IT-კონსალტინგ ცენტრ“. თბ., 2020. -176 გვ.

74. სურგულაძე გ., პეტრიაშვილი ლ. მონაცემთა მენეჯ-მენტის თანამედროვე ტექნოლოგიები (Oracle, MySQL, MongoDB,Hadoop). ISBN 978-9941-27-176-2. სტუ. „IT-კონსალტინგ ცენტრი“. თბ., 2017. -202 გვ.

75. სურგულაძე გ., ურუშაძე ბ. საინფორმაციო სისტემების მენეჯმენტის საერთაშორისო გამოცდილება: BSI, ITIL, COBIT. ISBN 978-9941-20-458-6. სტუ. „ტექნიკ.უნივერს.“, თბ., 2014 - 345 გვ.

76. სურგულაძე გ., ბულია ი. კორპორაციულ Web-აპლიკაციათა ინტეგრაცია და დაპროექტება. მონოგრ., ISBN 978-9941-20-165-3. სტუ. თბ., 2012. -324 გვ.

77. Sommerville I. Software Engineering 10th Edition. Copyright Pearson Education, Inc., publishing as Addison-Wesley. 2016

78. ISO/IEC 15504-5 oder SPICE (Software Process Improvement and Capability Determination) ist ein internationaler Standard. https://de.wikipedia.org/wiki/ISO/IEC_15504 (1.05.24)

79. V-Model. https://de.wikipedia.org/wiki/V-Modell (11.3.24)

80. 7 Principles of Software Testing: Defect Clustering and Pareto Principle. 2019. https://www.softwaretestinghelp.com /7-principles-of-software-testing/ (29.03.23)

81. Twine H., Green G. Understanding the Differences Between RabbitMQ and Kafka. 2023. https://tanzu.vmware.com/content/blog/understanding-the-differences-between-rabbitmq-vs-kafka (17.07.24)

82. Levy E. Kafka vs. RabbitMQ: Architecture, Performance & Use Cases. 2022. https://www.upsolver.com/blog/kafka-versus-rabbitmq-architecture-performance-use-case (9.04.24)

83. ქობულაშვილი ს., თურქია ე., სურგულაძე გ. დისტრიბუციული ტრანზაქციის მართვა მიკროსერვისულ არქიტექტურაში შაბლონური მიდგომების გამოყენებით („Saga Pattern“). სტუ-ს შრ.კრ. „მართვის ავტომატიზებული სისტემები“, N 1(33), vol.1., 2022. გვ. 58-65. DOI.org/10.36073/1512-3979

84. Comartin D. Event Choreography & Orchestration (Sagas). https://codeopinion.com/event-choreography-orchestration-sagas/ (09.11.23)

85. Turkia E., Bulia I., Giutashvili M. Management of Horizontal and Vertical Integration of Intercorporate Applications on the Basis of Service-oriented Architecture. Transact. of Georgian TU "*Automated Control Systems*", No 1(12), 2012. pp. 57–62

86. Compensating Transaction pattern. Microsoft. 2021. https://docs.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction (17.10.23)

87. McCaffrey C. Applying the Saga Pattern. Goto Conferences 2015. https://youtu.be/xDuwrtwYHu8 (11.10.23)

88. Distributed Tracing. Elasticsearch. 2022. https://www.elastic.co/guide/en/apm/guide/current/apm-distributed-tracing.html (06.06.24)

89. The Relationship between Risk and Continuous Testing. 2014. https://www.stickyminds.com/interview/relationship-between-risk-and-continuous-testing-interview-wayne-ariola (19.08.23)

90. სურგულაძე გ. კომპიუტერული პროგრამირების მეთოდები და მეთოდოლოგიები (SP, OOP, VP, Agile, UML). ISBN 978-9941-1900-1. სტუ. „IT-კონსალტინგ ცენტრი". თბ., 2019. -200 გვ.

91. UI vs UX. Internet resource: https://uxplanet.org/what-is-ui-vs-ux-design-and-the-difference-d9113f6612de (30.04.24)

92. Оптимизация программного кода. 2019. Internet resource: https://techrocks.ru/2019 /01/25/code-optimization-tips/ (3.05.24)

93. სურგულაძე გ., დოლიძე ს. მომხმარებლის ინტერფეისის დაპროგრამება (AngularJS, ReactJS). სტუ. „IT კონსალტ. სამეცნ.ცენტრი", თბ., 2019. -106 გვ.

94. Mukaj J., Draçi I. Containerization: Revolutionizing Software Development and Deployment Through Microservices Architecture Using Docker and Kubernetes. 2023. https://www.researchgate.net/publication/372501148_Containerization_Revolutionizing_Software_Development_and_Deployment_Through_Microservices_Architecture_Using_Docker_and_Kubernetes (10.07.24)

95. Docker, Kubernetes, Helm into DevOps methodologies. https://web.dmi.unict.it/sites/default/files/documenti_sito/BAXM

A200118%20BaxEnergy%20-%20Docker%20-%20Kubernetes%20-%20Helm%20ridotte.pdf (10.07.24)

96. OpenShift Container Platform overview. https://docs.openshift.com/container-platform/4.12/getting_started/openshift-overview.html (12.07.24)

97. Application Logs from OpenShift into Graylog. By Mehmet-nuricetin. https://community.graylog.org/t/application-logs-from-openshift-into-graylog/30301 (20.12.23)

98. სურგულაძე გ., კივილაძე გ. შესავალი NoSQL მონაცემთა ბაზებში. ISBN978-9941-0-9642-6. სტუ. „IT-კონსალტ.სამეცნ. ცენტრი“. თბ., 2017. -152 გვ.

99. Brewer E. CAP twelve years later: How the "rules" have changed. Computer. 45 (2). Institute of Electrical and Electronics Engineers (IEEE). 2012, pp.23–29. Doi:10.1109/mc.2012.37. https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/ (9.07.23)

100. Чоговадзе Г., Качибая В., Сургуладзе Г. Теория реляционных зависимостей и проектирование логической схемы баз данных. Моногр., ISBN 5-511-00072-8. Тбилисский Гос. Университет, Грузия. 1988. 230 ст.

101. სურგულაძე გ., პეტრიაშვილი ლ. მონაცემთა საცავის დაპროექტება და აგება ინტერნეტული ბიზნესისათვის. ISBN 978-9941-8-0623-0, მონოგრ., სტუ-ს „IT-კონსალტინგ სამეცნ. ცენტრი". თბ., 2022, - 200 გვ.

102. სურგულაძე გ., პეტრიაშვილი ლ. აპლიკაციების დაპროგრამება და მონაცემთა მენეჯმენტი. ISBN 978-9941-8-

3810-1, სტუ-ს „IT-კონსალტინგ სამეცნ.ცენტრი". თბ., 2022, -135 გვ.

103. ვედეკინდი ჰ., სურგულაძე გ., თოფურია ნ. განაწილებული ოფის-სისტემების მონაცემთა ბაზების დაპროექტება და რეალიზაცია UML-ტექნოლოგიით. ISBN 99940-57-17-0. სტუ. „ტექნიკ.უნივერს.", თბ., 2006 -237 გვ.

104. Kizilpinar D. Data Consistency in Microservices Architecture. 2021. https://medium.com/garantibbva-teknoloji/data-consistency-in-microservices-architecture-5c67e0f65256 (08.03.23)

105. მეიერ-ვეგენერი კ., სურგულაძე გ., ბასილაძე გ. საინფორმაციო სისტემების აგება მულტიმედიური მონაცემთა ბაზებით. ISBN 978-9941-20-468-5. სტუ. „ტექნიკური უნივერსიტეტი", თბ., 2014 -345 გვ.

106. CQRS (Command-Query Responsibility Segregation). Internet resource: https://www.eventstore.com/cqrs-pattern (12.07.24)

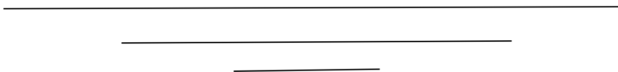
107. Microservices Architecture for Modern Digital Platforms. A white paper by Dr. Shailesh Kumar Shivakumar, Mindtree Limited. https://www.mindtree.com/sites/default/ files/2019-11/Microservices+Architecture+for+Modern+Digital+Platforms+-+WP_V2.pdf

108. ქობულაშვილი ს. კომპლექსური კორპორაციული სისტემების მონაცემთა მართვის სტრატეგია და შეფერხებების მინიმიზაციის გზები მიკროსერვისული არქიტექტურის ბაზაზე. სტუ-ის 100 და იმს ფაკ-ის დაარსების 65 წლისთ.

II სტუდ. სამეცნ.-პრაქტ. კონფ. *„ციფრული ტრანსფორმაცია - გამოწვევები და პროგრესი*“. თეზ.კრებ. გვ. 130-131

109. ქობულაშვილი ს. დეცენტრალიზებული მონაცემთა მართვა მიკროსერვისულ არქიტექტურაში თანამედროვე შაბლონური მიდგომების გამოყენებით. საერთაშ. პერიოდ. სამეცნ. ჟურნ. ,,ინტელექტი”, N1(71), 2022, თბ. გვ. 107-110

110. Buskermolen J. Kubernetes Autoscaling: Horizontal and Vertical explained. 2023. https://www.fullstaq.com/knowledge-hub/blogs/autoscaling-in-kubernetes (19.06.24).



რედაქცია - პროფ. გ. სურგულაძე
კომპიუტერული დაკაბადონება - ავტორების მიერ.
სტამბური გამოცემა - გოჩა დალაქიშვილი

(იბეჭდება ავტორთა ხარჯით)

გადაეცა წარმოებას 2.09 2024. ოფსეტური ქაღალდის ზომა 60X84 1/16. პირობითი ნაბეჭდი თაბახი 13.2, ტირაჟი 50 ეგზ.

სტუ-ს „IT კონსალტინგის სამეცნიერო ცენტრი˝,
თბილისი, მ.კოსტავას 77

ISBN 978-9941-8-6333-2