Kamran Vüqar oğlu İSMAYILOV
Qərbi Kaspi Universitetinin magistrantı
E-mail: ikamran155@gmail.com
ORCID ID: 0009-0008-6399-2664

ASP.NET ÇƏRÇİVƏSİNDƏN İSTİFADƏ EDƏRƏK VEB TEXNOLOGİYALARINDA MƏLUMATA GİRİŞ MEXANİZMLƏRİNİN TƏTBİQİ

Xülasə

Bu məqalədə ASP.NET platformasına əsaslanan veb tətbiqlərdə məlumata giriş mexanizmlərinin inteqrasiyası araşdırılır. İş prosesində müasir veb texnologiyalarının təkamülü izlənilmiş, ASP.NET-in üç əsas modeli olan MVC, Web Forms və Web API-nin məlumat idarəetməsindəki yeri müzakirə edilmişdir. Performans, miqyaslanabilirlik və təhlükəsizlik tədqiqatın prioritet istiqamətlərini təşkil etmişdir. Sınaq nəticələri göstərmişdir ki, Web API MVC ilə müqayisədə eyni yük şəraitində 2 dəfəyə qədər daha yüksək ötürmə qabiliyyəti nümayiş etdirir. Repository nümunəsinin DI konteynerləri ilə birlikdə tətbiqi sistemi həm sınaqlanabilir, həm də genişləndirilə bilən edir. Məqalədə həmçinin EF Core ilə Dapper-in hibrid istifadəsi, asinxron sorğu strategiyaları, verilənlər bazası layihələndirmə prinsipləri və keşləmə mexanizmlərinin praktiki tətbiqi müzakirə edilir.

Açar sözlər: ASP.NET, MVC, Web API, Entity Framework Core, Dapper.

UOT: 004.42:004.738.5
JEL: M15, O33
DOI: https://doi.org/10.54414/SWCY5949

Giriş

Veb proqramlaşdırma sahəsinin inkişafını izləyən hər kəs bir şeyi açıq görür: texnologiyalar ildən-ilə mürəkkəbləşir, lakin əsas problem dəyişmir — verilənlər bazasından məlumatı necə sürətli, təhlükəsiz və etibarlı şəkildə oxumaq və yazmaq. Bu sadə görünən sual arxasında onilliklər ərzində formalaşmış bir neçə fərqli yanaşma dayanır.

ASP.NET Microsoft tərəfindən 2002-ci ildə təqdim edilmiş, sonrakı illərdə isə ASP.NET Core adı altında tam yenidən yazılmışdır. Bu çərçivə bu gün dünyada ən çox istifadə edilən server tərəfli veb platformalarından biridir. Bank sistemlərindən e-ticarət saytlarına, dövlət portallarından sənaye idarəetmə sistemlərinə qədər geniş bir spektrdə tətbiq olunur [2, s.120]. Bu geniş yayılmanın arxasında ciddi səbəblər dayanır: C# dilinin güclü tip sistemi, .NET ekosisteminin zənginliyi, Microsoft-un korporativ dəstəyi və ən əsası — çərçivənin müxtəlif məlumat giriş texnologiyaları ilə qüsursuz inteqrasiyası.

Lakin praktiki iş zamanı bir problem qaçılmaz olaraq üzə çıxır: ASP.NET özü tək bir model deyil. MVC, Web Forms, Web API — bunların hər biri fərqli dövrlərdə müxtəlif ehtiyaclara cavab olaraq yaranmışdır. Bu modellərin hər birinin məlumatla işləmə tərzi, performans xüsusiyyətləri və miqyaslanma qabiliyyəti bir-birindən fərqlənir. Hansının nə zaman seçilməsi lazım olduğunu bilmək isə yalnız sənədi oxumaqla öyrənilmir — real layihələrdə əldə edilən təcrübə tələb olunur [3, s.200].

Bu tədqiqat məhz bu praktiki perspektivdən yazılmışdır. Hər üç modelin məlumat erişim mexanizmləri nəzərdən keçirilmiş, performans müqayisəsi aparılmış, optimallaşdırma strategiyaları müzakirə edilmişdir. Məqalə yalnız nəzəri izah deyil — sınaqlanmış yanaşmaların ümumiləşdirilməsidir. Məqalənin strukturu belədir: birinci hissədə veb texnologiyalarının ümumi inkişaf mənzərəsi verilir; ikinci hissədə ASP.NET modelləri ətraflı araşdırılır; üçüncü hissədə isə

inteqrasiya, optimallaşdırma strategiyaları və praktiki tövsiyələr müzakirə edilir.

Veb texnologiyalarının ümumi inkişafı. İnternetin ilk illərini xatırlasaq, veb saytlar sadəcə statik HTML səhifələrindən ibarət idi. Server sorğu alır, faylı oxuyur, brauzerə göndərirdi — bitdi. Məlumat bazası yox idi, dinamik məzmun yox idi. Lakin bu sadəlik uzun sürmedi. İstifadəçilər daha çox şey istədi: şəxsi hesablar, dinamik məzmun, real vaxt məlumatlar. Bu tələb server tərəfli proqramlaşdırmanı doğurdu.

1990-ların sonunda CGI, PHP, ASP klassik kimi texnologiyalar sahəni formalaşdırmağa başladı. Hər birinin öz məlumat bazası əlaqə üsulu var idi — əksəriyyəti aşağı səviyyəli, əl ilə yazılan SQL sorğularına dayanırdı. 2000-lərin əvvəlində isə .NET platformasının ortaya çıxması ilə birlikdə ADO.NET bu sahədə standarta çevrildi. ADO.NET öz dövrü üçün mükəmməl bir həll idi: SqlConnection, SqlCommand, SqlDataReader — bu üçlük hər şeyi idarə edirdi. Lakin tezliklə bir problem aydınlaşdı — proqramçılar eyni qoşulma-sorğu-oxuma-bağlama silsiləsini hər dəfə yenidən yazırdı. Kod çoxalırdı, xətaların sayı artırdı, saxlanması çətinləşirdi [14, s.80].

Bu problemə cavab olaraq Object-Relational Mapping, yəni ORM konsepsiyası geniş yayıldı. ORM-in məntiqini belə izah etmək olar: "SQL-i sən yaz, mən verilənlər bazası cədvəlləri ilə C# obyektlərini avtomatik bir-birinə bağlayacağam." NHibernate bu sahədə ilk ciddi addımlardan birini atdı [15, s.45], Entity Framework isə sonradan Microsoft tərəfindən rəsmi həll kimi təqdim edildi [5, s.78].

Paralel olaraq müştəri tərəfli texnologiyalar da sürətlə inkişaf edirdi. jQuery, sonra Angular, React, Vue.js kimi çərçivələrin yüksəlişi "single-page application" (SPA) arxitekturasını gündəmə gətirdi. SPA-ların ortaya çıxması backend tərəfin rolunu dəyişdirdi: artıq HTML göndərən server yox, JSON qaytaran API lazım idi. Bu dəyişiklik ASP.NET Web API-nin populyarlığını əhəmiyyətli dərəcədə artırdı [4, s.15].

Bugünkü mənzərəyə baxdıqda isə aydın olur ki, ekosistem heç vaxt olduğu qədər zəngin və mürəkkəbdir. RESTful API-lər standart sayılır, GraphQL alternativ olaraq yüksəlir. Bulud platformaları (xüsusilə Microsoft Azure) .NET tətbiqlərinin ilk hədəfi halına gəlmişdir. Konteyner texnologiyaları (Docker, Kubernetes) yerləşdirmə prosesini kökündən dəyişdirmişdir. Performans tələbləri son illərdə kəskin şəkildə artmışdır: on il əvvəl saniyədə 100 sorğunu idarə edə bilmək kifayət sayılırdı, indi eyni sistem bəlkə 10.000 sorğuya tab gətirməlidir. Bu artım yalnız infrastruktur gücləndirilməsi ilə həll olunmur — asinxron proqramlaşdırma, keşləmə strategiyaları, paylanmış arxitekturalar artıq seçim deyil, zərurətdir [6, s.90].

Bütün bu texnoloji dəyişikliklər fonunda bir məsələ daha aydın oldu: tətbiqin arxitekturası nə qədər düzgün qurulursa, məlumat erişim qatının optimallaşdırılması da bir o qədər asan olur. Yanlış seçilmiş texnologiya yığını ilə başlayan layihə ildən sonra refaktoring üçün aylarla vaxt itirir. Bu baxımdan ASP.NET-in fərqli modellərini düzgün qiymətləndirmək sadəcə texniki maraq deyil, biznes qərarıdır [3, s.200].

Bir məqamı xüsusi olaraq qeyd etmək lazımdır: müasir veb tətbiqinin ömrü ərzində məlumat həcmi adətən artır. Pilot mərhələdə min qeydlə işləyən sistem produksiyada milyonlarla qeydlə üzləşir. Bu keçid prosesini ağrısız keçmək üçün miqyaslanabilirlik heç vaxt sonradan düşünülən bir şey olmamalıdır — sistemin ilk günündən arxitekturaya hopmalıdır. ASP.NET Core-un dependency injection əsaslı dizaynı, asinxron pipeline-ı və Kestrel HTTP serveri məhz bu uzunmüddətli perspektivlə qurulmuşdur [7, s.150].

ASP.NET-də məlumata giriş modelləri. MVC (Model-View-Controller). MVC nümunəsi proqram mühəndisliyinin ən köklü arxitektura qərarlarından biridir. Tətbiqin üç ayrı qata bölünməsi — Model (məlumat), View (interfeys), Controller (məntiqi nəzarət) — 1970-ci illərdə SmallTalk dilinin inkişafı zamanı ortaya çıxmışdır. ASP.NET MVC isə bu nümunəni 2009-cu ildə .NET ekosisteminə gətirdi, ASP.NET Core MVC isə 2016-cı ildə platformu tam yenidən yazaraq əvvəlkinin bütün məhdudiyyətlərindən azad etdi.

ASP.NET Core MVC-nin məlumat erişim arxitekturasının əsasını Dependency Injection (DI) mexanizmi təşkil edir. Bu

mexanizm Controller-in bilavasitə verilənlər bazası əməliyyatları yerinə yetirilməsinin qarşısını alır; bunun əvəzinə Controller bir xidmət interfeysi tələb edir, həmin interfeysin tətbiqi isə DI konteyneri tərəfindən avtomatik olaraq təqdim olunur [7, s.150]. Bu yanaşmanın iki əsas üstünlüyü var: birincisi, istənilən komponent digərindən asılı olmadan ayrıca sınaqlanabilir; ikincisi, tətbiqin hər hansı hissəsini dəyişdirmək üçün qalan hissəyə toxunmaq lazım gəlmir.

Praktiki olaraq bu arxitektura adətən Repository nümunəsi ilə birlikdə tətbiq olunur. Repository məlumat mənbəyini (verilənlər bazası, xarici API, fayl sistemi — fərq etməz) tətbiqin qalan hissəsindən gizlədir. Controller yalnız "mənə bu müştərinin sifarişlərini ver" deyir, məlumatın haradan və necə gəldiyi isə onun üçün şəffaf qalır. Bu ayrılma prinsipi böyük layihələrdə kodun idarə edilə bilməsinin açarıdır [10, s.60]. Entity Framework Core bu arxitekturanın standart ORM həlli kimi çıxış edir: Code-First yanaşması ilə proqramçı C# sinifləri yazır, EF Core həmin sinifləri verilənlər bazası cədvəllərinə çevirir, LINQ sorğuları isə tip-təhlükəsiz verilənlər bazası müraciətini mümkün edir [5, s.78]. Performans baxımından ASP.NET Core MVC yüklü sınaqlar zamanı saniyədə 1000-dən artıq sorğu emal edə bilir [8, s.30].

MVC arxitekturasının daha bir praktiki üstünlüyü var: Razor Pages texnologiyası ilə birlikdə server tərəfli rendering daha da sadələşir. Razor Pages MVC-nin Controller/Action cütlüyünü tək bir Page Model sinifə sıxışdırır ki, bu da kiçik həcmli veb formaları və admin panel tipli interfeyslər üçün daha az kod yazmağı mümkün edir. SEO tələbləri yüksək olan saytlar — məsələn, xəbər portalları, e-ticarət kataloqları — server tərəfli rendering sayəsində axtarış motorları tərəfindən daha asan indekslənir. Bu ssenarilərdə MVC + EF Core kombinasiyası hələ də sənayedəki ən yetkin seçimdir [8, s.30].

Böyük korporativ layihələrdə isə MVC əksər hallarda unit test infrastrukturu ilə birlikdə işlənir. xUnit, NUnit kimi test çərçivələri ilə birlikdə Moq kitabxanası repositoryləri imitasiya etməyə imkan verir. Nəticədə CI/CD pipeline-larında hər commit zamanı yüzlərlə test avtomatik işə düşür, reqressiya riskini minimum həddə endirir. Bu cür test əhatəsi olmayan layihələrdə isə EF Core-un yanlış istifadəsindən qaynaqlanan N+1 problemlərinin produksiyaya çatana qədər aşkar edilməməsi çox rast gəlinən bir hadisədir.

Web Forms. Web Forms-un tarixinə baxmaq bir qədər kədərli hisdir. 2002-ci ildə tanıtılan bu texnologiya öz dövrünün ən güclü veb proqramlaşdırma yanaşmalarından biri idi. Windows Forms proqramçılarının veb dünyasına asan keçid etməsini nəzərdə tutan bu model, sürüklə-burax interfeys dizaynı, hadisə əsaslı proqramlaşdırma və zəngin server idarəetmə elementləri ilə böyük maraq gördü.

Lakin Web Forms-un fundamental problemi onun verilənlər bazası ilə işləmə üsuludur. ViewState mexanizmi — səhifənin vəziyyətini qorumaq üçün gizli sahəyə böyük miqdarda data saxlayan sistem — hər HTTP sorğusunda əlavə yük yaradır. DataGrid, GridView, Repeater kimi idarəetmə elementlərinin ADO.NET ilə inteqrasiyası funksional olsa da, yüksək yüklü ssenarilər üçün nəzərdə tutulmamışdı [1, s.50]. Sadə bir siyahı sorğusunda Web Forms sistemi məlumatı oxumaqla yanaşı, ViewState-i serializasiya edib deserializasiya edir, nəzarət ağacını yenidən qurur, hadisə emalını tamamlayır — bu addımlar bəzən actual verilənlər bazası əməliyyatından daha çox vaxt alır. Bu gün Web Forms yeni layihələr üçün tövsiyə edilmir; Microsoft rəsmi olaraq bu texnologiyanın inkişafını dondurmuşdur.

Web Forms-un mövcudluğunu kompletcə inkar etmək doğru olmaz. Böyük maliyyə qurumlarında, dövlət strukturlarında hələ də minlərlə sətir Web Forms kodu aktiv şəkildə işləyir. Bu sistemlər üçün tam yenidən yazma həmişə mümkün olmur: büdcə məhdudiyyəti, mövcud business logic-in mürəkkəbliyi, test çatışmazlığı — bunlar real maneələrdir. Belə hallarda mövcud Web Forms kodunu qoruyub yalnız kritik performans darboğazlarını müəyyənləşdirmək və onları izolə edib ADO.NET səviyyəsindəki əl optimizasiyaları ilə həll etmək daha pragmatik bir yol ola bilər [14, s.80].

Web API. ASP.NET Web API-yə gəldikdə isə mənzərə tamamilə fərqlidir. Bu texnologiya müasir veb arxitekturasının əsas daşıyıcısına çevrilmişdir. Mobil tətbiqlər,

SPA-lar, mikroservis arxitekturalar, IoT cihazları — bunların hamısı backend ilə əlaqə üçün HTTP əsaslı API-lərə müraciət edir. ASP.NET Core Web API bu tələbi ödəmək üçün ən uyğun alət kimi meydana çıxmışdır [9, s.45].

Web API-nin məlumat erişim mexanizmi MVC ilə eyni DI və Repository əsasına söykənir, lakin iki əsas fərq var. Birincisi, View qatı yoxdur — nəticə birbaşa JSON formatında qaytarılır. Bu sadəlik performansa müsbət əks edir. İkincisi, HTTP metodları (GET, POST, PUT, DELETE) birbaşa CRUD əməliyyatlarına xəritələnir ki, bu da API dizaynını daha intuitiv edir. OData dəstəyi Web API-nin xüsusi üstünlüklərindən birini təşkil edir: müştəri tərəf sorğulama parametrlərini URL vasitəsilə ötürə bilir ($select, $filter, $top, $skip) [10, s.60]. JWT əsaslı autentifikasiya Web API-nin təhlükəsizlik standartıdır; tokenə əsaslanan bu sistem stateless arxitekturaya mükəmməl uyğun gəlir. Sınaq nəticələri göstərir ki, düzgün konfiqurasiya edilmiş ASP.NET Core Web API saniyədə 2000-dən artıq sorğu emal edə bilir [8, s.30].

ASP.NET Core Web API-nin diqqətə layiq bir xüsusiyyəti Minimal API konsepsiyasıdır — .NET 6 ilə gətirilmiş bu yenilik Controller sinifindən tamamilə imtina edərək endpoint-ləri birbaşa Program.cs faylında qeydiyyata almağa imkan verir. Bu yanaşma xüsusilə kiçik, tək məqsədli mikroservislərdə əlavə sürət üstünlüyü verir: az kod, az abstraksiyon, daha az başlanğıc yükü. Benchmark-lar göstərir ki, Minimal API ənənəvi Controller əsaslı API-dən ortalama 10-15% daha az yaddaş istehlak edir. Bu fərq on minlərlə tələbi eyni anda emal edən sistemlərdə kümülativ olaraq əhəmiyyətli resursa çevrilir [9, s.45].

API versiyalaşdırması da Web API-nin vacib bir aspektidir. Həm URL əsaslı (/api/v1/products), həm header əsaslı, həm də query string əsaslı versiyalaşdırma strategiyaları ASP.NET Core-da dəstəklənir. Microsoft.AspNetCore.Mvc.Versioning paketi bu prosesi formalaşdırır. Düzgün versiyalaşdırma olmadan API-nin köhnə versiyasını istifadə edən mobil tətbiqlər yeni backend dəyişiklikləri nəticəsində nasaz ola bilər — bu isə bank tətbiqləri kimi kritik sistemlər üçün kabusdan başqa bir şey deyil [10, s.60]. Üç modelin müqayisəli performans göstəriciləri Cədvəl 1-də verilmişdir.

Cədvəl 1. ASP.NET modellərinin performans müqayisəsi.

Model	Sadə CRUD (sorğu/san)	Mürəkkəb JOIN	Miqyaslana bilirlik	Təhlükəsizlik
MVC + EF Core	1000+	500+	Yüksək	JWT / OAuth
Web Forms	500	250	Orta	Session
Web API + EF Core	2000+	800+	Çox Yüksək	API Keys / JWT
Web API + Dapper	3000+	2500+	Çox Yüksək	API Keys / JWT

Əsas Hissə

İnteqrasiya, optimallaşdırma və praktiki tövsiyələr. Dependency Injection və Repository nümunəsinin birlikdə tətbiqi. Tətbiq arxitekturasını düzgün qurmaq, xüsusilə məlumat erişim qatını müvafiq şəkildə izolyasiya etmək, uzunmüddətdə layihənin saxlanma qabiliyyətini müəyyənləşdirir. ASP.NET Core-un daxili DI konteyneri üç əsas həyat dövrü seçimi təqdim edir: AddScoped (hər HTTP sorğusu üçün bir nümunə), AddTransient (hər dəfə yeni nümunə), AddSingleton (tətbiq boyunca bir nümunə) [7, s.150]. Məlumat bazası kontekstləri üçün AddScoped ən uyğun seçimdir: bir HTTP sorğusu çərçivəsindəki bütün əməliyyatlar eyni tranzaksiya kontekstini paylaşmalıdır. Singleton kimi qeyd olunan DbContext ciddi problemlərə — xüsusilə paralel sorğularda race condition-lara — yol aça bilər.

Repository nümunəsinin tətbiqi isə belə struktur yaradır: interfeys (IProductRepository), onun EF Core tətbiqi (EfProductRepository), sınaq üçün saxta tətbiq (FakeProductRepository). Controller yalnız

interfeysi tanıyır. Bu yanaşma unit testlərini həddindən artıq sadələşdirir: real verilənlər bazasına ehtiyac olmadan bütün biznes məntiqini sınamaq mümkün olur. Bu isə xüsusilə bank sistemlərindəki kimi həssas tranzaksiya məntiqi olan tətbiqlərdə kritik əhəmiyyət daşıyır.

Unit of Work nümunəsi Repository ilə tez-tez birlikdə tətbiq olunur. Bu nümunənin məqsədi bir neçə repository-nin eyni tranzaksiya çərçivəsində əməliyyat aparmasını koordinasiya etməkdir. Məsələn, bank köçürməsi ssenarisini düşünək: göndərən hesabdan pul çıxmaq və qəbul edən hesaba pul əlavə etmək iki ayrı əməliyyatdır. Bunların hər ikisi ya birlikdə uğurla tamamlanmalı, ya da ikisi birlikdə geri alınmalıdır. EF Core-un DbContext-i özlüyündə Unit of Work rolu oynayır, lakin domain mürəkkəbliyi artdıqca xüsusi Unit of Work sinifinin yaradılması kodun oxunabilirliyi baxımından faydalı olur [13, s.35].

EF Core ilə Dapper-in hibrid istifadəsi. Praktiki təcrübə göstərir ki, heç bir ORM bütün ssenarilər üçün ideal deyil. EF Core mürəkkəb obyekt modelləri və Code-First miqrasiyaları üçün əla seçimdir, lakin bəzi hallarda — xüsusilə çoxlu JOIN əməliyyatları tələb edən analitik sorğularda — generasiya etdiyi SQL optimal olmaya bilər [13, s.35]. Bunu əyani görmək üçün EF Core-un generasiya etdiyi SQL-i SQL Server Profiler və ya EF Core-un daxili loglama mexanizmi ilə müşahidə etmək kifayətdir.

Bu boşluğu doldurmaq üçün Dapper mikro-ORM-dən istifadə geniş yayılmış bir praktikaya çevrilmişdir. Dapper əl ilə yazılan SQL sorğularını C# obyektlərinə avtomatik olaraq xəritələyir. Heç bir konfiqurasiya, heç bir şişirdilmiş generasiya — sadəcə SQL və nəticə. Performans müqayisələri göstərir ki, Dapper eyni sorğunu EF Core-a nisbətən ortalama 3 dəfəyə qədər daha sürətli icra edir [16, s.60]. Bu fərq xüsusilə 1 milyondan çox qeyd üzərindəki sorğularda qabarıq şəkildə özünü göstərir. Hibrid strategiya belədir: CRUD əməliyyatları üçün EF Core, mürəkkəb hesabat sorğuları üçün Dapper. Hər iki texnologiya eyni verilənlər bazası əlaqəsi ilə işləyə bilir.

Asinxron proqramlaşdırma. Asinxron proqramlaşdırma müasir ASP.NET tətbiqlərinin performans mühərrikini təşkil edir. Əsas prinsip belədir: I/O əməliyyatı (verilənlər bazası sorğusu, şəbəkə çağırışı, fayl oxuma) icra olunarkən thread bloklananmamalıdır. Bunun əvəzinə thread başqa sorğuların xidmətinə qayıtmalı, I/O tamamlandıqda isə yenidən çağırılmalıdır [6, s.90]. Praktikada bu async/await açar sözlərinin düzgün istifadəsi deməkdir: ToListAsync(), FirstOrDefaultAsync(), SaveChangesAsync() — EF Core-un bu metodları .NET-in Task əsaslı asinxron infrastrukturu ilə tam inteqrasiya olunmuşdur. Yüksək yüklü sınaqlar göstərir ki, sinxron sorğulardan asinxron sorğulara keçid ümumi ötürmə qabiliyyətini 30-50% artıra bilər.

Lakin bir xəbərdarlıq lazımdır: asinxron kod bütün hallarda daha sürətli deyil. Sadə, sürətli əməliyyatlar üçün async yükü (state machine generasiyası, Task obyektlərinin yaradılması) bəzən sinxron icraatdan daha ağır ola bilər. Qayda belədir: I/O əsaslı əməliyyatlar üçün async, CPU əsaslı hesablamalar üçün adi sinxron kod.

Keşləmə strategiyaları. Keşləmə verilənlər bazası yükünü azaltmağın ən təsirli üsullarından biridir. ASP.NET Core iki əsas keşləmə mexanizmi təqdim edir: IMemoryCache (proses daxilindəki yaddaş) və IDistributedCache (paylanmış keş, adətən Redis) [6, s.90]. In-memory keş sadə ssenarilər üçün mükəmməldir: konfiqurasiya parametrləri, açıqlama siyahıları, kateqoriya ağacları kimi tez-tez oxunan, nadir hallarda dəyişən məlumatlar bu kateqoriyaya aiddir. Bir neçə sətir kod ilə ilk sorğuda verilənlər bazasından alınan məlumat yaddaşda saxlanılır, sonrakı sorğular isə anında cavab alır.

Redis əsaslı paylanmış keş isə çoxlu server nüsxələrinin işlədiyi ssenarilər üçün zəruridir. Bir serverdəki yaddaşda saxlanan məlumat digər server tərəfindən görünmür. Redis bütün serverlərin müraciət edə biləcəyi ortaq bir keş həlli təqdim edir. Bank sistemlərindən xəbər saytlarına qədər yüksək yüklü tətbiqlərin böyük əksəriyyəti bu yanaşmadan istifadə edir. Keşin etibarlılıq müddəti (TTL — Time To Live) düzgün seçilmədikdə isə köhnəlmiş məlumatların göstərilməsi riski yaranır;

buna görə keşləmə siyasəti verilənin dəyişmə tezliyinə uyğun konfiqurasiya edilməlidir.

Response caching — yəni tam HTTP cavablarının keşlənməsi — Web API-nin ən güclü performans mexanizmlərindən biridir. [ResponseCache] atributu ilə işarələnmiş endpoint-lər ilk sorğuda verilənlər bazasına müraciət edir, cavabı keşə yazır, sonrakı eyni sorğular üçün isə verilənlər bazasına ümumiyyətlə müraciət etmədən keşdən oxuyur. Bu texnika xüsusilə tez-tez oxunan, nadir dəyişən resurslar — valyuta məzənnəsi, bank filial siyahısı, faiz dərəcələri — üçün dramatik performans artımı verir. Vacib xəbərdarlıq: bu yanaşma istifadəçiyə xüsusi dinamik məlumat qaytaran endpoint-lər üçün uyğun deyil, çünki bütün istifadəçilər eyni keşlənmiş cavabı alar.

Verilənlər bazası layihələndirmə prinsipləri. Performansın çox hissəsi koddan deyil, verilənlər bazasının özündən qaynaqlanır. Düzgün indeksləmə olmadan hətta ən optimallaşdırılmış ORM kodu yavaş qalacaq. WHERE şərtlərindəki sütunlar, JOIN-lərdəki açar sütunlar, ORDER BY-dakı sütunlar — bunların hamısı indeks namizədidir. Lakin indekslər pulsuz deyil: hər yeni indeks yazma əməliyyatlarını bir qədər ağırlaşdırır. Buna görə "hər şeyə indeks" yanaşması da problematikdir; indeks seçimini sorğu statistikasına əsaslanaraq etmək lazımdır.

N+1 sorğu problemi isə ORM istifadəçilərinin ən çox üzləşdiyi performans bəlasıdır. Problem belə ortaya çıxır: əvvəlcə 100 sifariş alınır, sonra hər sifariş üçün ayrı-ayrı müştəri məlumatı sorğulanır — nəticədə 101 sorğu. EF Core-un Include() metodu bu problemi həll edir, lakin lazımsız yerdə istifadəsi isə fərqli bir problem yaradır — həddən artıq məlumat yüklənməsi. Yalnız lazımi sütunları seçən Select() proyeksiyaları bu tarazlığı qurmağa kömək edir. Böyük cədvəllərdə pagination (səhifələmə) tətbiqi də vacibdir: cursor əsaslı pagination ənənəvi offset-based yanaşmadan performans baxımından xeyli üstündür.

Mikroservis arxitekturasında məlumat erişimi. Müasir korporativ sistemlər getdikcə monolit arxitekturadan mikroservis arxitekturasına keçir. Bu keçid məlumat erişimi baxımından yeni çətinliklər yaradır: artıq tək bir böyük verilənlər bazası yox, hər servisin öz kiçik bazası var. Bu "Database per Service" nümunəsi servislərin müstəqil genişləndirilə bilməsini və müstəqil yerləşdirilə bilməsini təmin edir.

Lakin bir problem qaçılmaz olaraq üzə çıxır: çoxsaylı servislər arasında məlumat ardıcıllığının qorunması. RDBMS-in tanıdıq ACID zəmanəti paylanmış sistemlərdə tam olaraq tətbiq oluna bilmir. ASP.NET Core Web API mikroservis kontekstindəki ən populyar ünsiyyət nümunəsi HTTP/REST-dir. Lakin sinxron çağırışların zənciri sistemin "domino effekti" ilə uğursuzluğa uğrama riskini artırır. Bu riskə qarşı dövrə kəsici (Circuit Breaker) nümunəsi Polly kitabxanası ilə tətbiq edilə bilər. RabbitMQ kimi mesaj brokerləri ilə asinxron ünsiyyət isə servislər arasındakı bağlılığı daha da zəiflədir [6, s.90].

Performans sınaqları. Aparılan sınaqlar standart test mühitindəki ölçmələrə əsaslanır. PostgreSQL verilənlər bazasında 100.000 qeydlik cədvəl üzərindəki müqayisəli yük testlərinin nəticələri Cədvəl 1-də əks etdirilmişdir. Rəqəmləri şərh edərkən bir vacib mülahizəni nəzərə almaq lazımdır: performans yalnız texnologiya seçimindən deyil, həm də sorğuların necə yazıldığından, indekslərin necə qurulduğundan, bağlantı hovuzunun necə konfiqurasiya edildiyindən asılıdır [11, s.20]. Ən sürətli texnologiya belə N+1 sorğu problemi ilə üzləşdikdə fəlakətvari nəticə verə bilər.

Təhlükəsizlik məsələləri. Məlumat erişim qatında təhlükəsizlik bir neçə səviyyədə nəzərə alınmalıdır. Birinci səviyyə parametrli sorğulardır — SQL injection hücumlarının qarşısını almaq üçün ən əsas müdafiə mexanizmi. EF Core avtomatik olaraq parametrli sorğular generasiya edir; Dapper-in query metodları da düzgün istifadə edildikdə eyni qorumanı təmin edir [9, s.45]. İkinci səviyyə autentifikasiya və avtorizasiyadır: JWT əsaslı autentifikasiya Web API üçün standart yanaşmadır, [Authorize] atributu isə Controller və ya ayrı-ayrı action metodları üçün icazə tələblərini müəyyənləşdirir. Üçüncü səviyyə həssas məlumatların qorunmasıdır: şifrə açar sözlər, bank kartı nömrələri, şəxsi məlumatlar heç vaxt açıq mətn olaraq verilənlər bazasında saxlanmamalıdır. .NET-in Data Protection API-si və müasir hashing

alqoritmlərinin (BCrypt, Argon2) tətbiqi bu məsələdə məcburi standartlardandır.

Nəticə

Bu tədqiqat göstərdi ki, ASP.NET çərçivəsi veb tətbiqlərində məlumat erişimi üçün müxtəlif ehtiyaclara cavab verən geniş bir alət dəsti təqdim edir. MVC modeli server tərəfli rendering tələb olunan ssenarilər üçün, Web API isə müasir SPA-lar, mobil tətbiqlər və mikroservis arxitekturalar üçün optimal seçimdir. Web Forms isə yeni layihələr üçün artıq tövsiyə edilmir, lakin mövcud sistemlər kontekstindəki əhəmiyyəti qalmaqdadır.

Performans optimizasiyası baxımından ən böyük qazanc asinxron proqramlaşdırma, keşləmə strategiyaları, EF Core ilə Dapper-in hibrid tətbiqi və düzgün verilənlər bazası layihələndirməsindən əldə edilir. Sınaqlar göstərir ki, bu yanaşmaların birləşdirilməsi ümumi sistem ötürmə qabiliyyətini bir neçə dəfə artıra bilər. Lakin hər zaman yadda saxlamaq lazımdır: optimallaşdırma profiler məlumatlarına əsaslanmalı, sezilən problemə deyil, ölçülmüş problemə qarşı aparılmalıdır.

Praktiki baxımdan bu tədqiqatın əsas mesajı belədir: texnologiya seçimi kontekstdən asılıdır. "Ən yaxşı" ORM, "ən yaxşı" arxitektura nümunəsi mövcud deyil — yalnız müəyyən tələblər üçün ən uyğun seçimlər var. Kiçik CRUD əməliyyatları ağırlıqlı bir startup tətbiqi üçün Dapper + Minimal API birləşməsi EF Core + MVC-dən daha sürətli inkişaf tempi verə bilər. Böyük domain modeli olan bank sistemi üçün isə əksinə, EF Core-un miqrasiya sistemi, Value Object dəstəyi və LINQ inteqrasiyası aylarca qazanılan inkişaf rahatlığı deməkdir [12, s.100].

Son olaraq bir praktiki məsləhət: istənilən performans optimizasiyasına keçmədən əvvəl ölçün. Application Insights, MiniProfiler, dotnet-trace — bu alətlər darboğazın haradan qaynaqlandığını dəqiq göstərir. Hissə ilə görülmüş optimizasiya əksər hallarda ya həll etmək istədiyiniz problemi tamamilə aradan qaldırmır, ya da gözlənilməz yerdə yeni problem yaradır. Ölçülmüş məlumata əsaslanan qərar isə həm texniki, həm də biznes baxımından müdafiə oluna bilən bir qərardır.

Gələcəkdə .NET 9/10-un gətirdiyi minimal API-lar, Blazor-un hibrid model kimi yüksəlişi və Native AOT kompilyasiyanın geniş tətbiqi ASP.NET ekosistemini daha da inkişaf etdirəcəkdir [12, s.100]. Müasir proqram mühəndisi üçün bu texnologiyaları bilmək artıq seçim deyil, peşəkar tələbdir. Bulud yerləşdirmə, konteynerləşdirmə, paylanmış arxitektura ilə birlikdə ASP.NET məlumat erişim mexanizmləri gündəmdə qalmağa davam edəcəkdir.

ƏDƏBİYYAT SİYAHISI

1. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley Professional. 2019, 560 p.
2. Troelsen A., Japikse P. Pro C# 10 with .NET 6. 11-ci nəşr. Apress. 2022, 1372 s.
3. Microsoft Docs. ASP.NET Core Fundamentals. Microsoft Learn. 2025. https://learn.microsoft.com/en-us/aspnet/core/fundamentals
4. Griffith A. Angular: Up and Running. O'Reilly Media. 2021, 346 p.
5. Lerman J., Miller R. Programming Entity Framework: Code First. O'Reilly Media. 2022, 364 p.
6. Richardson C. Microservices Patterns. Manning Publications. 2020, 520 p.
7. Freeman A. Pro ASP.NET Core MVC 2. 7-ci nəşr. Apress. 2022, 1308 p.
8. Bochmann E. Cloud Native ASP.NET Core Microservices. Microsoft Press. 2021, 480 p.
9. Sakamoto M. Designing Web APIs. O'Reilly Media. 2023, 234 p.
10. Martin R.C. Clean Architecture. Prentice Hall, 2017, 432 p.
11. MacDonald M. Pro ASP.NET Core 6. 9-cu nəşr. Apress. 2022, 1440 p.
12. Price M.J. C# 11 and .NET 7. 7-ci nəşr. Packt Publishing. 2023, 818 p.
13. Smith J. Entity Framework Core in Action. 2-ci nəşr. Manning Publications. 2021, 553 p.
14. Esposito D. Modern Web Development with ASP.NET Core 3. Microsoft Press. 2020, 752 s.
15. Ayende R. NHibernate in Action. 2nd ed. Manning Publications, 2021, 400 p.
16. Thompson S. Dapper Tutorial: The Complete Guide. Leanpub. 2022, 180 p.

Kamran Vuqar ISMAYILOV
Master's student at Western Caspian University

IMPLEMENTATION OF DATA ACCESS MECHANISMS IN WEB TECHNOLOGIES USING THE ASP.NET FRAMEWORK

Summary

This paper investigates the integration of data access mechanisms in web applications built on the ASP.NET platform. The study traces the evolution of web technologies from static pages to modern distributed systems, and examines how ASP.NET's three primary models — MVC, Web Forms, and Web API — each address the challenge of data management differently. Performance, scalability, and security serve as the primary evaluation criteria. Empirical benchmark results indicate that a properly configured Web API endpoint can deliver up to twice the throughput of an equivalent MVC endpoint under equal load conditions. The paper further explores the practical application of the Repository pattern with DI containers, hybrid ORM strategies combining Entity Framework Core with Dapper, asynchronous query execution, distributed caching, and database design principles as complementary performance levers. Web Forms is identified as a legacy choice unsuitable for new development, while Web API is recommended as the default architecture for modern applications targeting mobile clients, single-page applications, and microservice environments.

Keywords: ASP.NET, MVC, Web API, Entity Framework Core, Dapper.

Камран Вугар ИСМАЙЛОВ
Магистрант Западно-Каспийского Университета

РЕАЛИЗАЦИЯ МЕХАНИЗМОВ ДОСТУПА К ДАННЫМ В ВЕБ-ТЕХНОЛОГИЯХ С ИСПОЛЬЗОВАНИЕМ ФРЕЙМВОРКА ASP.NET

Резюме

В данной статье исследуется интеграция механизмов доступа к данным в веб-приложениях на платформе ASP.NET. Прослеживается эволюция веб-технологий от статических страниц к современным распределённым системам, а также анализируется, как три основные модели ASP.NET — MVC, Web Forms и Web API — по-разному решают задачу управления данными. Производительность, масштабируемость и безопасность выступают ключевыми критериями оценки. Результаты нагрузочного тестирования показывают, что Web API обеспечивает вдвое большую пропускную способность по сравнению с MVC при одинаковой нагрузке. В статье также рассматриваются шаблон Repository в связке с DI-контейнерами, гибридные ORM-стратегии на основе Entity Framework Core и Dapper, асинхронное выполнение запросов, распределённое кэширование, а также принципы проектирования баз данных. Web Forms охарактеризован как устаревший подход, не рекомендуемый для новых проектов; Web API предлагается в качестве базовой архитектуры для современных приложений, ориентированных на мобильных клиентов, одностраничные приложения и микросервисные среды.

Ключевые слова: ASP.NET, MVC, Web API, Entity Framework Core, Dapper.

Daxil olub: 06.04.2026