

МОНОЛИТТІ МИКРОСЕРВИСТЕРГЕ КЕЗЕҢ-КЕЗЕҢІМЕН КӨШІРУ КЕЗІНДЕ BACKEND-ҚОСЫМШАЛАР АРХИТЕКТУРАСЫН ОҢТАЙЛАНДЫРУ: PHP ЖӘНЕ GO ИНТЕГРАЦИЯСЫ

ТИМОШЕНКО Р.А. , БИГАЛИЕВА А.З. 

Тимошенко Роман Алексеевич - Магистрант, Абылкас Сағынов атындағы Қарағанды техникалық университеті, Қарағанды қ., Қазақстан

E-mail: rombik2004r@gmail.com, <https://orcid.org/0009-0004-4099-8220>

*Бигалиева Альфия Замировна - PhD, қауымдастырылған профессор (доцент), Абылкас Сағынов атындағы Қарағанды техникалық университеті, Қарағанды қ., Қазақстан

E-mail: bigalievaaalfija@gmail.com, <https://orcid.org/0000-0002-0136-5402>

Андатпа. Мақалада монолитті backend-қосымшаны микросервистік архитектураға кезең-кезеңімен көшіру мәселесі қарастырылады. Зерттеудің өзектілігі корпоративтік ақпараттық жүйелердің функционалдық және жүктемелік өсуі жағдайында монолитті архитектураға тән тығыз байланыстылық, тұтас жүйені масштабтау қажеттілігі, компоненттерді тәуелсіз орналастырудың шектеулілігі және сыртқы сервистердің негізгі бизнес-сценарийлердің орындалуына әсер етуімен айқындалады. Зерттеудің мақсаты – монолиттен микросервистерге кезең-кезеңімен көшудің архитектуралық паттерндерін талдау және PHP/Yii2 негізіндегі монолит пен Go тілінде әзірленген бөлініп шығарылатын сервис мысалында көшіру тиімділігін бағалау әдістемесін әзірлеу. Зерттеу барысында архитектуралық және салыстырмалы талдау, эксперименттік стендті жобалау және жүктемелік тестілеу әдістері қолданылды. Көшіру стратегиясы ретінде Strangler Fig тәсілі таңдалып, оны API Gateway/Reverse Proxy, Anti-Corruption Layer, Outbox, Circuit Breaker және идемпотенттілік паттерндерімен үйлестіру ұсынылды. Бірінші кезеңде хабарламалар жіберу модулін жеке Go-сервисіне бөліп шығару қарастырылды. Тапсырысты жасау және оқиғаны Outbox кестесіне жазу бір жергілікті транзакция шеңберінде орындалып, хабарламаны нақты жеткізу асинхронды түрде жүзеге асырылады. Монолиттік және мақсатты конфигурацияларды салыстыру үшін Docker ортасында эксперименттік стенд пен k6 негізіндегі жүктемелік тестілеу әдістемесі әзірленді. Бағалау критерийлері ретінде жауап беру уақытының p50, p95 және p99 процентильдері, өткізу қабілеті, қате жауаптардың үлесі және тәуелді компонент істен шыққан кездегі жүйенің тұрақтылығы белгіленді. Зерттеу нәтижелері кезең-кезеңімен көшіруді толық қайта жазуға балама ретінде ұйымдастыруға арналған әдістемелік негіз қалыптастырады. Сандық жүктемелік эксперимент пен оның статистикалық талдауы зерттеудің келесі кезеңінде жүргізіледі.

Түйін сөздер: монолитті архитектура, микросервистік архитектура, кезең-кезеңімен көшіру, Strangler Fig, Outbox pattern, PHP, Go.

Кіріспе

Корпоративтік ақпараттық жүйелердің дамуы көбінесе монолитті архитектурадан басталады: бұл бизнес-логиканы, дерекқормен жұмыс істеуді және пайдаланушыға ұсынылатын интерфейсті бір код репозиторийі мен бір орналастыру процесі шеңберінде біріктіретін бірыңғай қосымша. Жобаны дамытудың бастапқы кезеңінде мұндай тәсіл орынды болып табылады, өйткені ол инфрақұрылымға жұмсалатын шығындарды азайтады, қателерді түзету процесін және әзірлеуді жеделдетеді. Алайда жүйе мен әзірлеушілер командасы өскен сайын монолитке тән мәселелер пайда бола бастайды [1, 17; 2, 24]: байланысы жоқ функционалдылыққа әсер ету қаупінсіз өзгерістер енгізу күрделене түседі, кез келген жаңарту қосымшаны толық қайта орналастыруды талап етеді, жүйені масштабтау тек жекелеген шамадан тыс жүктелген модульдер үшін емес, тұтастай жүзеге асырылады, ал әртүрлі әзірлеу топтары біртұтас кодтық кеңістікте жұмыс істеуге мәжбүр болады. Бұл өз кезегінде қақтығыстар санын арттырып, бағдарламалық өнімді жеткізу қарқынын төмендетеді.

Бұл мәселені шешудің бір жолы – микросервистік архитектураға көшу, мұнда жекелеген функционалдық домендер тәуелсіз түрде орналастырылатын сервистерге бөлінеді. Дегенмен қолданыстағы жүйені «нөлден» толық қайта жазу жоғары тәуекелдермен байланысты [1, 72; 2, 26; 3, 40; 4, 32]: оған айтарлықтай уақыт пен ресурстар жұмсау, қайта жазу кезеңінде функционалдылықты дамытуды уақытша тоқтату, сондай-ақ жинақталған бизнес-логиканың

және тек қолданыстағы кодта ғана көрініс тапқан жасырын талаптардың жоғалу ықтималдығы жатады. Бұған балама ретінде монолит пен жаңа сервистер өтпелі кезеңде қатар жұмыс істейтін, ал функционалдылық шағын әрі бақыланатын бөліктермен сыртқы сервистерге шығарылатын кезең-кезеңімен көшіру тәсілін қолдануға болады.

Осы зерттеудің мақсаты – монолитті backend-қосымшаны микросервистік архитектураға кезең-кезеңімен көшірудің архитектуралық паттерндерін талдау және PHP-монолитті (Yii2 фреймворкі) Go тілінде әзірленген бөлініп шығарылатын сервиспен интеграциялау мысалында мұндай көшудің әсерін бағалау әдістемесін әзірлеу.

Қойылған мақсатқа қол жеткізу үшін келесі міндеттер шешіледі:

- PHP-монолитінің типтік архитектурасын және тән әлсіз тұстарды талдау;
- кезең-кезеңімен көшіру стратегиясы мен паттерндерін таңдауды негіздеу;
- монолит пен бөлініп шығарылатын Go-сервисінің өзара әрекеттесу схемасын, оның ішінде деректерді келісімді түрде беру механизмін жобалау;
- сервисті бөліп шығаруға дейінгі және одан кейінгі жүйе сипаттамаларын салыстырмалы бағалауға арналған эксперимент әдістемесін әзірлеу;
- ішінара көшіру тәуекелдерін және микросервистерге көшудің орынсыз болатын жағдайларын анықтау.

Осы мақала зерттеудің бірінші кезеңі болып табылады: онда теориялық және әдістемелік база қалыптастырылып, эксперименттік стенд жобаланып, өлшеу әдістемесі сипатталған. Толық жүктемелік эксперименттің сандық нәтижелері және олардың статистикалық талдауы кейінгі зерттеу жұмысының нысаны болып табылады.

Зерттеу материалдары мен әдістері

Зерттеу нысаны ретінде PHP тілінде Yii2 фреймворкін пайдалана отырып және PostgreSQL реляциялық дерекқорды қолдану арқылы іске асырылған монолитті архитектуралық типтік backend-қосымшасы қарастырылады. Мұндай архитектура есепке алу және оқу-әкімшілік бағыттағы ақпараттық жүйелерде кеңінен қолданылады және пайдаланушыларды, клиенттерді, тапсырыстарды, хабарламаларды және есептілікті басқару модульдерін бірыңғай кодтық кеңістік пен дерекқордың бірыңғай схемасы шеңберінде біріктіреді.

Зерттеу барысында келесі әдістер қолданылды: бастапқы жүйенің әлсіз тұстарын анықтауға бағытталған архитектуралық талдау (модульдер арасындағы байланыстылықты, бірыңғай істен шығу нүктелерін және бірлесіп орналастыруға қойылатын шектеулерді талдау); эволюциялық архитектура паттерндері мен микросервистік интеграция паттерндері бойынша әдебиеттерді салыстырмалы талдау; оқшауланған ортада монолит пен бөлініп шығарылатын сервистің негізгі сипаттамаларын қайта жаңғыртатын эксперименттік стендті жобалау әдісі; көшіруге дейінгі және кейінгі жүйенің сандық сипаттамаларын алу мақсатында жүктемелік тестілеу әдісі.

Зерттеудің құралдық базасын PHP 8 және Yii2 фреймворкі (монолитті іске асыру және эмуляцияланатын legacy-логиканы сақтау), Go 1.2x (бөлініп шығарылатын сервисті іске асыру), PostgreSQL (деректерді, оның ішінде Outbox кестесін сақтау), Nginx (монолит пен жаңа сервис арасындағы сұрауларды бағыттауға арналған reverse proxy / API Gateway рөлінде), Docker және Docker Compose (эксперименттік стендті оқшаулау және оның қайта жаңғыртылуын қамтамасыз ету), сондай-ақ синтетикалық жүктемені генерациялау және жауап беру уақыттары мен өткізу қабілетінің метрикаларын жинау үшін кб құралы құрайды.

Бастапқы монолитті архитектура және оның әлсіз тұстары

Қарастырылып отырған бастапқы жүйе бірыңғай PostgreSQL схемасымен өзара әрекеттесетін классикалық үш деңгейлі Yii2-қосымшасы болып табылады (контроллерлер – сервистер/модельдер – деректерге қол жеткізу деңгейі). Функционалдық тұрғыдан алғанда, қосымшада бірнеше салыстырмалы түрде дербес домендерді бөліп көрсетуге болады: пайдаланушылар мен клиенттерді басқару, тапсырыстарды рәсімдеу, есептерді қалыптастыру және хабарламалар жіберу (email/SMS/push). Алайда олардың барлығы физикалық тұрғыдан біртұтас қосымша ретінде орналастырылады және іске қосылады.

Мұндай архитектураның негізгі әлсіз тұстарына мыналар жатады: қосымша процесі мен дерекқор деңгейіндегі бірыңғай істен шығу нүктесі; байланысты масштабтау – жекелеген модульге (мысалы, хабарламаларды тарату модуліне) түсетін жүктеменің артуы бүкіл қосымшаны тұтастай масштабтауды талап етеді; байланысты жүйені іске енгізу – бір модульдегі катені түзету бүкіл қосымшаны қайта жинақтап, қайта орналастыруды қажет етеді, бұл тоқтап қалу уақытын және байланысы жоқ функционалдылықтағы регрессиялар қаупін арттырады; сыртқы жүйелерге (мысалы, SMTP-серверге немесе push-хабарламалар сервисіне) пайдаланушы сұрауын өңдеу ағынының ішінде тікелей синхронды шақырулар жасау. Мұндай тәсіл жауап беру уақытын арттырып, негізгі сценарийді (мысалы, тапсырысты жасау процесін) қосалқы функционалдың (хабарлама жіберудің) қолжетімділігіне тәуелді етеді [1, 30; 3, 78; 10, 203].

Strangler Fig стратегиясы

Кезең-кезеңімен көшіру стратегиясы ретінде Мартин Фаулер ұсынған Strangler Fig қағидаты таңдалды [5]. Бұл тәсілде жаңа жүйе қолданыстағы жүйенің айналасында біртіндеп қалыптасып, жекелеген функцияларды кезең-кезеңімен өзіне алады. Нәтижесінде монолит не толықтай жаңа жүйемен алмастырылады, не көшіру экономикалық тұрғыдан тиімсіз болып табылатын негізгі ядро деңгейіне дейін қысқартылады. Бұл тәсілдің негізгі ерекшелігі — өтпелі кезең барысында монолит пен жаңа сервистердің қатар жұмыс істеуі және трафикті біртіндеп әрі бақыланатын түрде жаңа компоненттерге бағыттау.

Алғашқы болып бөлініп шығарылатын компонентті таңдау келесі критерийлерге негізделеді: модульдің жүйенің негізгі транзакциялық логикасымен байланысының төмен болуы; жауапкершілік аймағы нақты айқындалған доменнің (bounded context) болуы; негізгі деректердің дереу келісімділігін қамтамасыз ету тұрғысынан модульдің сыни маңызға ие болмауы (eventual consistency моделін қолдануға мүмкіндік болуы) [1, 84; 3, 114; 8, 100]; сондай-ақ тәуелсіз масштабтаудың өзіндік артықшылығының болуы, мысалы, жүктеменің біркелкі емес және қысқа мерзімді шарықтау сипатымен байланысты болуы.

Notification Service модулі аталған критерийлердің барлығына сәйкес келеді: ол тапсырыстар доменімен әлсіз байланысқан, тапсырыс туралы деректердің тұтастығына тікелей әсер етпейді, кейінге қалдырылған өңдеуге төзімді және жүктеме профилі жүйенің негізгі бөлігінің жүктеме профилінен айтарлықтай ерекшеленеді. Осы факторлар оны көшірудің алғашқы кезеңінде бөліп шығаруға тиімді кандидат етеді.

Монолит пен бөлініп шығарылатын сервисті интеграциялау паттерндері

API Gateway / Reverse Proxy. Бірыңғай кіру нүктесі (эксперимент барысында – Nginx) сұрауларды олардың жолына және/немесе трафик үлесіне қарай монолитке немесе жаңа сервиске бағыттайды. Бұл клиент жағында өзгерістер енгізбей-ақ жүктемені жаңа сервиске біртіндеп ауыстыруға мүмкіндік береді [5].

Anti-Corruption Layer (ACL). Legacy-жүйесінің деректер моделін жаңа сервистің домендік моделіне түрлендіретін аралық қабат. Ол ескірген деректер схемасының ерекшеліктерінің жаңа, дербес доменге енуіне жол бермейді [5; 6, 350; 7].

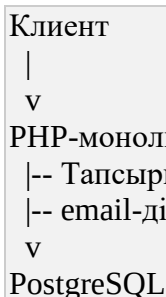
Outbox pattern. Бұл паттерн «қосарлы жазу» (Dual Write) мәселесін шешуге арналған [3, 98; 9]. Аталған мәселе бір сұрауды өңдеу барысында деректерді жергілікті дерекқорға жазу және сыртқы сервисті синхронды шақыру әрекеттері қатар орындалған жағдайда туындайды: жергілікті деректер жазылғаннан кейін, бірақ сыртқы сервис шақырылғанға дейін ақау орын алса, оқиға жоғалуы мүмкін. Outbox паттернінде бизнес-оқиғаны арнайы кестеге жазу негізгі деректерді жазумен бір жергілікті транзакция шеңберінде жүзеге асырылады. Одан кейін жеке релей-процесс оқиғаларды асинхронды түрде жариялайды. Бұл бөлінген транзакцияларды қолданбай, «кемінде бір рет» (at-least-once) жеткізу моделін қамтамасыз етуге мүмкіндік береді.

Saga pattern. Бірнеше сервисті қамтитын бизнес-транзакцияларды екі фазалы коммитті қолданбай басқару үшін пайдаланылады. Жергілікті транзакциялар тізбегі оқиғалар арқылы хореографиялық тәсілмен немесе арнайы үйлестіруші арқылы оркестрленген тәсілмен басқарылады, ал ақау орын алған жағдайда компенсаторлық әрекеттер орындалады [3, 114; 5].

Circuit Breaker, Retry және идемпотенттілік. Circuit Breaker жаңа сервистің қолжетімсіздігі кезінде шақыруларды уақытша тоқтатып, жүйенің қалпына келуіне мүмкіндік беру арқылы каскадты ақаулардың алдын алады. Қайталап орындау саясаты (Retry) операциялардың идемпотенттілігімен қатар қолданылуы тиіс. Мысалы, идемпотенттілік кілттерін пайдалану сұрауды қайта жіберу кезінде хабарламалардың немесе тапсырыстардың қайталанып жіберілуіне жол бермейді [3, 78; 8, 68].

Сервисті бөліп шығаруға дейінгі және кейінгі мысал

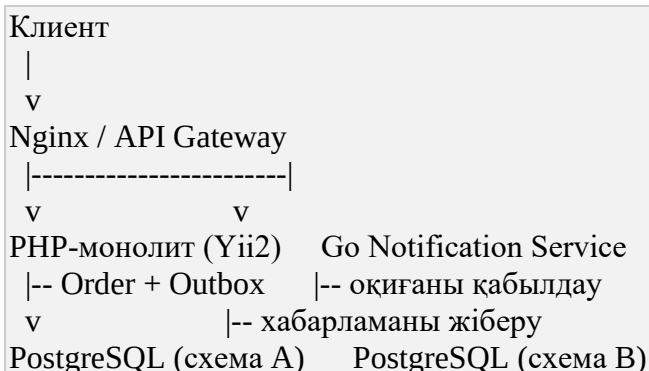
Көшіруге дейін монолитте тапсырысты жасау процесі бір HTTP-сұрау шеңберінде хабарлама жіберуді синхронды түрде шақыруды қамтиды (сурет 1):



Сурет 1. Монолитте тапсырысты жасау

Псевдокод деңгейінде бұл үдерісті ықшам түрде былай көрсетуге болады: тапсырысты жасау әдісі Order нысанын дерекқорға жазады, содан кейін сол ағынның ішінде сыртқы пошта серверіне жүгінеді. SMTP-сервер қолжетімсіз болған немесе тайм-аут мәні асып кеткен жағдайда, тапсырыстың өзі іс жүзінде дұрыс сақталған болуы мүмкін болғанымен, клиенттің сұрауы қате туралы жауаппен аяқталады немесе оның орындалуы кідіреді.

Хабарламалар сервисі бөлініп шығарылғаннан кейін өзара әрекеттесу схемасы төмендегідей өзгереді:



Сурет 2. Тапсырысты өңдеу және хабарламалар сервисінің өзара әрекеттесуі

Жаңа схемада тапсырысты жасау операциясы және оқиғаны Outbox кестесіне жазу монолиттің бірыңғай жергілікті транзакциясы шеңберінде орындалады. Жеке релей-процесс жинақталған оқиғаларды Go-сервисіне (REST-эндпоинт немесе хабарламалар кезегі арқылы) жариялайды. Go-сервис хабарламаны нақты жіберуге жауап береді және деректердің жеке схемасына ие.

Бұл қайта ұйымдастырудың негізгі нәтижесі – тапсырысты жасау процесі сыртқы хабарлама жіберу арнасының қолжетімділігі мен өнімділігіне тәуелді болмайды. Go-сервисі уақытша қолжетімсіз болған жағдайда, оқиғалар Outbox кестесінде жинақталып, байланыс қалпына келтірілгеннен кейін жеткізіледі.

Экспериментті ұйымдастыру

Салыстырмалы бағалау жүргізу мақсатында эксперименттік стендті екі конфигурацияда орналастыру ұсынылады. **А конфигурациясы (базалық)** бастапқы монолиттің жұмыс істеу

логикасын қайталайды: тапсырысты жасау сол HTTP-сұрауының шеңберінде хабарлама жіберуді синхронды түрде шақырумен қатар жүреді. **В конфигурациясы (мақсатты)** бөлініп шығарылған сервисті іске асырады: тапсырысты жасау кезінде оқиға Outbox кестесіне жергілікті түрде жазылады, ал хабарламаны нақты жіберу монолитпен REST API және Outbox паттернін іске асыратын релей-процесс арқылы өзара әрекеттесетін Go-сервисі тарапынан асинхронды түрде орындалады.

Сынақ жүргізу жағдайларының салыстырмалылығын қамтамасыз ету үшін екі конфигурацияны да CPU мен жедел жад ресурстарына қойылған бекітілген шектеулері бар оқшауланған Docker-контейнерлерінде жүйені іске енгізу көзделеді [4, 36; 11]. Тапсырысты жасау эндпоинтіне түсетін жүктеме K6 құралы арқылы виртуалды пайдаланушылар санын кезең-кезеңімен көшіру жолымен (бір мезгілде 10 → 50 → 100 клиент) әр кезеңнің ұзақтығын тұрақты мәнде сақтай отырып генерацияланады [9].

Метрикаларды жинау әдістемесі

Жүктемелік тестілеу барысында келесі көрсеткіштерді тіркеу жоспарланады: тапсырысты жасау эндпоинтіне жауап беру уақытының p50, p95 және p99 процентильдері; өткізу қабілеті (секундына сәтті өңделген сұраулар саны, RPS); қате жауаптардың үлесі (error rate) [11]; сондай-ақ негізгі бизнес-сценарийдің қосалқы функционалдылықтың істен шығуына төзімділігін бағалау мақсатында тәуелді компоненттің жасанды істен шығуы кезіндегі жүйенің мінез-құлқы – А конфигурациясында пошта серверін, ал В конфигурациясында Go-сервисін уақытша ажырату.

Бұдан басқа, жүйені пайдалану тұрғысынан маңызды көрсеткіштерді сапалық бағалау көзделеді: тек хабарламалар модуліне қатысты өзгерістерді орналастыруға кететін уақыт (А конфигурациясында – монолитті толық қайта орналастыру уақыты, В конфигурациясында – жеке сервисті орналастыру уақыты), сондай-ақ хабарламалар логикасына өзгерістер енгізу кезінде кодтық базаның ревьюден өтуге тиіс көлемі.

Нәтижелер және оларды талқылау

Осы жұмыс зерттеудің бастапқы кезеңін қамтитындықтан, төменде келтірілген кесте жүктемелік тестілеу нәтижелерін ұсыну форматын белгілейді. Нақты сандық мәндер эксперимент толыққанды орналастырылған стендте жүргізілгеннен кейін толтырылуы тиіс және оларды түпкілікті өлшенген деректер ретінде түсіндіруге болмайды.

Зерттеудің келесі кезеңінде тексерілуге тиіс гипотеза мынадай: хабарламаларды жіберу функциясын асинхронды, оқшауланған сервиске көшіру негізгі тапсырыс жасау сценарийінің кідірісін төмендетеді, өйткені синхронды сыртқы шақыру критикалық орындау жолынан шығарылады, сондай-ақ хабарламаларды жеткізу арнасы істен шыққан жағдайда жүйенің тұрақтылығын арттырады. Go-сервисі уақытша қолжетімсіз болған жағдайда да тапсырыстарды қабылдау жалғаса береді, себебі оқиғалар Outbox кестесінде буферленеді. Сонымен қатар инфрақұрылымдық күрделіліктің артуы және тапсырысты жасау мен хабарламаны нақты жеткізу арасында кідірістің пайда болуы күтіледі.

Ішінара көшіру тәуекелдері және оларды азайту тәсілдері [1, 78; 3, 78; 9]

Кесте 1. Кезең-кезеңімен көшіру тәуекелдері және оларға сәйкес архитектуралық паттерндер

Тәуекел	Тәуекелдің көрінісі	Тәуекелді азайту тәсілі (паттерн)
Қосарлы жазу мәселесі	Дерекқорға жазу мен сервисті шақыру арасындағы ақау салдарынан оқиғаның жоғалуы	Outbox pattern
Бөлінген транзакциялар	Екі сервистегі деректерді атомарлы түрде өзгерту мүмкін еместігі	Saga (хореография/оркестрация)
Сұраулардың қайталануы	Retry кезінде хабарламаның қайта жіберілуі	Идемпотенттілік, идемпотенттілік кілттері
Каскадты істен шығулар	Жаңа сервистің қолжетімсіздігі монолиттің жұмысын баяулатады немесе тоқтатады	Circuit Breaker, тайм-ауттар

Инфрақұрылымдық күрделіліктің артуы	Компоненттер, конфигурациялар және істен шығу нүктелерінің санының артуы	Observability: журналдау, трассалау, мониторинг
Логиканың уақытша қайталануы	Функционалдылықтың бір бөлігі монолитте де, сервисі де қатар іске асырылады	Feature flag, трафикті бақыланатын кезең-кезеңімен көшіру

Микросервистерге көшірудің орынсыз болатын жағдайлары

Кезең-кезеңімен көшірудің артықшылықтарына қарамастан, микросервистік архитектураға көшу барлық жағдайда бірдей негізді шешім бола бермейді [1, 26; 4, 48; 10; 11]. Сервистерді бөліп шығару мынадай жағдайларда орынсыз: жобаның ауқымы шағын және бизнес-логиканың күрделілігі төмен болса; әзірлеу командасы шағын болып, бірыңғай кодтық кеңістіктегі өзгерістерді үйлестіру мәселесіне тап болмаса; жүйеге түсетін жүктеме төмен және модульдер арасында айқын біркелкі емес сипатқа ие болмаса; жекелеген функцияларды тәуелсіз масштабтау қажеттілігі болмаса [1, 24].

Мұндай жағдайларда бөлінген инфрақұрылымға, observability құралдарына және деректердің келісімділігін қамтамасыз етуге жұмсалатын шығындар микросервистік архитектураның икемділігінен алынатын пайдадан асып түседі. Сондықтан мұндай жүйелер үшін жақсы құрылымдалған монолит (так называемый modular monolith) анағұрлым қолайлы шешім болып қала береді.

Қорытынды

Жұмыста PHP/Yii2 негізінде құрылған жүйені мысалға ала отырып, монолитті архитектурадан микросервистік архитектураға кезең-кезеңімен көшу мәселесі қарастырылды, сондай-ақ Go тілінде бірінші сервисті бөліп шығару тәсілі ұсынылды. Strangler Fig стратегиясын Outbox, Anti-Corruption Layer, Circuit Breaker паттерндерімен және идемпотенттілікті қамтамасыз етумен үйлестіру жүйені толық қайта жазумен салыстырғанда тәуекелдерді төмендете отырып, көшіруді кезең-кезеңімен жүзеге асыруға мүмкіндік беретіні көрсетілді. Хабарламалар сервисін бөліп шығаруға дейінгі және кейінгі жүйенің кідірісін, өткізу қабілетін және тұрақтылығын салыстырмалы бағалауға арналған эксперименттік стендтің архитектурасы мен жүктемелік тестілеу әдістемесі жобаланды.

Негізгі әдістемелік қорытынды мынада: монолитті микросервистерге көшіру бастапқы кодты бір бағдарламалау тілінен екіншісіне ауыстырумен ғана шектелмейді. Ол жүйенің жауапкершілік шекараларын және оның құрамдас бөліктері арасындағы деректерді келісу тәсілдерін қайта қарауды көздейді. Болашақ зерттеулердің негізгі бағыттарына: орналастырылған эксперименттік стендте толық жүктемелік эксперимент жүргізіп, сандық деректер алу және оларға статистикалық талдау жасау; Saga паттерні арқылы бірнеше сервисті үйлестіруді қажет ететін сценарийлерге ұсынылған әдістемені қолдану; сондай-ақ бөлініп шығарылатын сервистер саны артқан кезде observability құралдарын енгізуге жұмсалатын шығындарды бағалау жатады.

Әдебиеттер тізімі

1. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol: O'Reilly Media, 2021. 612 p.
2. Taibi D., Lenarduzzi V., Pahl C. Processes, Motivations and Issues for Migrating to Microservices Architectures: An Empirical Investigation // IEEE Cloud Computing. 2017. Vol. 4, № 5. P. 22–32. DOI: [10.1109/MCC.2017.4250931](https://doi.org/10.1109/MCC.2017.4250931).
3. Richardson C. Microservices Patterns: With Examples in Java. Shelter Island: Manning Publications, 2018. 520 p.
4. Faustino D., Gonçalves N., Portela M., Silva A. R. Stepwise Migration of a Monolith to a Microservice Architecture: Performance and Migration Effort Evaluation // Performance Evaluation. 2024. Vol. 164. Art. 102411. DOI: [10.1016/j.peva.2024.102411](https://doi.org/10.1016/j.peva.2024.102411).
5. Microsoft. Strangler Fig pattern [Electronic resource]. URL: Microsoft Azure Architecture Center – Strangler Fig pattern. <https://learn.microsoft.com/en->

[us/azure/architecture/patterns/strangler-fig](https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig)

6. Vernon V. Implementing Domain-Driven Design. Upper Saddle River: Addison-Wesley Professional, 2013. 656 p.

7. Richardson C. Pattern: Transactional Outbox [Electronic resource]. URL: <https://microservices.io/patterns/data/transactional-outbox.html>

8. Nygard M. T. Release It!: Design and Deploy Production-Ready Software. 2nd ed. Raleigh: Pragmatic Bookshelf, 2018. 368 p.

9. Grafana Labs. Grafana k6 Documentation [Electronic resource]. URL: <https://grafana.com/docs/k6/latest/>

10. Balalaie A., Heydarnoori A., Jamshidi P. Migrating to Cloud-Native Architectures Using Microservices: An Experience Report // Advances in Service-Oriented and Cloud Computing. 2016. P. 201–215. DOI: [10.1007/978-3-319-33313-7_15](https://doi.org/10.1007/978-3-319-33313-7_15).

11. Abgaz Y. M., McCarren A., Elger P., Solan D., Lapuz N., Bivol M., Jackson G. M., Yilmaz M., Buckley J. F., Clarke P. M. Decomposition of Monolith Applications IntoMicroservices Architectures: A Systematic Review // IEEE Transactions on Software Engineering. 2023. Vol. 49, № 8. P. 4213–4242. DOI: [10.1109/TSE.2023.3287297](https://doi.org/10.1109/TSE.2023.3287297).

References

1. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol: O'Reilly Media, 2021. 612 p.

2. Taibi D., Lenarduzzi V., Pahl C. Processes, Motivations and Issues for Migrating to Microservices Architectures: An Empirical Investigation // IEEE Cloud Computing. 2017. Vol. 4, № 5. P. 22–32. DOI: [10.1109/MCC.2017.4250931](https://doi.org/10.1109/MCC.2017.4250931).

3. Richardson C. Microservices Patterns: With Examples in Java. Shelter Island: Manning Publications, 2018. 520 p.

4. Faustino D., Gonçalves N., Portela M., Silva A. R. Stepwise Migration of a Monolith to a Microservice Architecture: Performance and Migration Effort Evaluation // Performance Evaluation. 2024. Vol. 164. Art. 102411. DOI: [10.1016/j.peva.2024.102411](https://doi.org/10.1016/j.peva.2024.102411).

5. Microsoft. Strangler Fig pattern [Electronic resource]. URL: Microsoft Azure Architecture Center – Strangler Fig pattern. <https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig>

6. Vernon V. Implementing Domain-Driven Design. Upper Saddle River: Addison-Wesley Professional, 2013. 656 p.

7. Richardson C. Pattern: Transactional Outbox [Electronic resource]. URL: <https://microservices.io/patterns/data/transactional-outbox.html>

8. Nygard M. T. Release It!: Design and Deploy Production-Ready Software. 2nd ed. Raleigh: Pragmatic Bookshelf, 2018. 368 p.

9. Grafana Labs. Grafana k6 Documentation [Electronic resource]. URL: <https://grafana.com/docs/k6/latest/>

10. Balalaie A., Heydarnoori A., Jamshidi P. Migrating to Cloud-Native Architectures Using Microservices: An Experience Report // Advances in Service-Oriented and Cloud Computing. 2016. P. 201–215. DOI: [10.1007/978-3-319-33313-7_15](https://doi.org/10.1007/978-3-319-33313-7_15).

11. Abgaz Y. M., McCarren A., Elger P., Solan D., Lapuz N., Bivol M., Jackson G. M., Yilmaz M., Buckley J. F., Clarke P. M. Decomposition of Monolith Applications IntoMicroservices Architectures: A Systematic Review // IEEE Transactions on Software Engineering. 2023. Vol. 49, № 8. P. 4213–4242. DOI: [10.1109/TSE.2023.3287297](https://doi.org/10.1109/TSE.2023.3287297).

ОПТИМИЗАЦИЯ АРХИТЕКТУРЫ BACKEND-ПРИЛОЖЕНИЙ ПРИ ПОЭТАПНОЙ МИГРАЦИИ МОНОЛИТА НА МИКРОСЕРВИСЫ: ИНТЕГРАЦИЯ PHP И GO

ТИМОШЕНКО Р.А. , БИГАЛИЕВА А.З. 

Тимошенко Роман Алексеевич – Магистрант, Карагандинский технический университет имени Абылкаса Сагинова, г. Караганда, Казахстан.

E-mail: rombik2004r@gmail.com, <https://orcid.org/0009-0004-4099-8220>

***Бигалиева Альфия Замировна** – PhD, ассоциированный профессор (доцент), Карагандинский технический университет имени Абылкаса Сагинова, г. Караганда, Казахстан

E-mail: bigaliyevaalfija@gmail.com, <https://orcid.org/0000-0002-0136-5402>

Аннотация. В статье рассматривается проблема поэтапной миграции монолитного backend-приложения на микросервисную архитектуру. Актуальность исследования обусловлена характерными для монолитной архитектуры ограничениями, возникающими по мере роста корпоративных информационных систем: высокой связанностью компонентов, необходимостью масштабирования системы в целом, ограниченными возможностями независимого развертывания отдельных компонентов и зависимостью основных бизнес-сценариев от внешних сервисов. Цель исследования заключается в анализе архитектурных паттернов поэтапной миграции с монолитной на микросервисную архитектуру и разработке методики оценки эффективности миграции на примере интеграции монолита на PHP/Yii2 с выделяемым сервисом, разработанным на языке Go. В исследовании применялись методы архитектурного и сравнительного анализа, проектирования экспериментального стенда и нагрузочного тестирования. В качестве стратегии миграции выбран подход Strangler Fig, предложено его сочетание с паттернами API Gateway/Reverse Proxy, Anti-Corruption Layer, Outbox, Circuit Breaker и обеспечением идиомпотентности. На первом этапе рассматривается выделение модуля отправки уведомлений в отдельный Go-сервис. Создание заказа и запись события в таблицу Outbox выполняются в рамках одной локальной транзакции, после чего фактическая доставка уведомления осуществляется асинхронно. Для сравнения монолитной и целевой конфигураций разработаны экспериментальный стенд на базе Docker и методика нагрузочного тестирования с использованием k6. В качестве критериев оценки определены процентильные показатели времени ответа p50, p95 и p99, пропускная способность, доля ошибочных ответов и устойчивость системы при отказе зависимого компонента. Результаты исследования формируют методическую основу для организации поэтапной миграции как альтернативы полной переработке системы. Полномасштабный нагрузочный эксперимент и его статистический анализ будут проведены на следующем этапе исследования.

Ключевые слова: монолитная архитектура, микросервисная архитектура, поэтапная миграция, Strangler Fig, Outbox pattern, PHP, Go.

OPTIMIZING BACKEND APPLICATION ARCHITECTURE THROUGH INCREMENTAL MONOLITH-TO-MICROSERVICES MIGRATION: PHP AND GO INTEGRATION

TIMOSHENKO R.A. , BIGALIYEVA A.Z. 

Timoshenko Roman Alekseevich – Master's student, Karaganda technical university named after Abylkas Saginov, Karaganda, Kazakhstan.

E-mail: rombik2004r@gmail.com, <https://orcid.org/0009-0004-4099-8220>

***Bigaliyeva Alfiya Zamirovna** – PhD, associate professor, Abylkas Saginov Karaganda technical university, Karaganda, Kazakhstan

E-mail: bigaliyevaalfija@gmail.com, <https://orcid.org/0000-0002-0136-5402>

Abstract. The paper examines the problem of gradually migrating a monolithic backend application to a microservice architecture. The relevance of the study is determined by the limitations inherent in monolithic architectures that emerge as enterprise information systems grow, including high component coupling, the need to scale the system as a whole, limited opportunities for independent deployment of individual components, and the dependence of core business scenarios on external services. The aim of the study is to analyze architectural patterns for gradual migration from a monolithic to a microservice architecture and to develop a methodology for evaluating migration effectiveness using the integration of a PHP/Yii2-based monolith with a service developed in Go as an example. The study employs architectural and comparative analysis, experimental testbed design, and load testing methods. The Strangler Fig approach was selected as the migration strategy, combined with the API Gateway/Reverse Proxy, Anti-Corruption Layer, Outbox, and Circuit Breaker patterns, as well as idempotency mechanisms. The first migration stage involves extracting the notification module into a separate Go service. Order creation and event recording in the Outbox table are performed within a single local transaction, while the actual delivery of notifications is carried out asynchronously. To compare the monolithic and

target configurations, an experimental testbed based on Docker and a k6-based load testing methodology were developed. The evaluation criteria include p50, p95, and p99 response-time percentiles, throughput, error rate, and system resilience in the event of a dependent component failure. The results of the study provide a methodological basis for organizing gradual migration as an alternative to a complete system rewrite. A full-scale load experiment and its statistical analysis will be conducted at the next stage of the research.

Key words: monolithic architecture, microservice architecture, gradual migration, Strangler Fig, Outbox pattern, PHP, Go.