**ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ**

**Факултет „Компютърни системи и технологии”**

КУРСОВ ПРОЕКТ
ПО РАЗПРЕДЕЛЕНИ ПРИЛОЖЕНИЯ И ЗАЩИТА

НА ТЕМА

# „EventRegistry – Дистрибутирано приложение за обработване на събития“

**Студент:** Никита Радославов Койнов
**ФАК. №** 381222002 **Група:** 91

## Съдържание

*Въведение* ..... *3*
*Теоретична част* ..... *3*
*Описание на системата EventRegistry* ..... *5*
*Организиране на Local Development среда* ..... *6*
*Работа със средата и библиотеките* ..... *7*
*Проблеми и тяхното решение* ..... *8*
*Заключение* ..... *9*

# Въведение

В този курсов проект се реализира асинхронна комуникация между микроуслуги чрез опашки от съобщения, използвайки Azure Service Bus и неговия локален емулатор. Целта на проекта е да се демонстрира как отделни компоненти на разпределена система могат да обменят информация помежду си чрез message queue вместо чрез директни повиквания, което осигурява по-слаба обвързаност и по-голяма гъвкавост. Избраната тема – „Реализация на messaging комуникация чрез Azure Service Bus Emulator“ – акцентира върху изграждането на локална среда за разработка с емулатор на Azure Service Bus, позволяваща разработчиците да тестват и прототипират микросервисна архитектура без необходимост от връзка с облака.

Настоящата документация описва теоретичните основи на микроуслужната архитектура и асинхронната обработка на съобщения, след което подробно разглежда изградената система EventRegistry – нейната архитектура, използвани технологии (.NET, Azure Service Bus, Docker), конфигуриране на локална среда с емулатор, имплементация на обмена на съобщения, реализираните HTTP крайни точки, както и срещнатите технически предизвикателства и решения. Накрая са обобщени постигнатите резултати, изводите от работата по проекта и възможностите за бъдещо развитие.

# Теоретична част

Microservices (микроуслуги) представляват архитектурен стил, при който приложението е изградено като съвкупност от малки, самостоятелно разгърнати услуги, всяка от които реализира точно определена бизнес функционалност. Микроуслугите са слабо свързани помежду си (loosely coupled) и комуникират през дефинирани интерфейси, най-често по мрежови протоколи. Това позволява независимо разработване, разгръщане и скалиране на отделните компоненти. За разлика от монолитната архитектура, където всички функционалности са в едно общо приложение, микроуслугите разделят системата на модули (услуги) около бизнес възможности, което улеснява поддръжката и ускорява разработката при по-големи екипи. Основни предимства на микроуслужната архитектура са повишената гъвкавост, по-лесното добавяне на нови функции и устойчивостта – повреда в една услуга не срива цялото приложение.

Един от ключовите подходи за свързване на микроуслуги е чрез асинхронна комуникация, използваща message queue (опашка за съобщения). При този модел услугите обменят съобщения посредством посредник (брокер) на съобщения, вместо да извършват директни REST или RPC повиквания помежду си. Основно предимство на опашките е времевата разкавказаност (temporal decoupling) – подателят и получателят не е нужно да работят по едно и също време. Съобщенията се съхраняват надеждно в опашката, докато получателят бъде готов да ги обработи. Това означава, че дадена микроуслуга може да изпрати съобщение и веднага да продължи работа, без да изчаква другата страна да отговори, което повишава ефективността. Освен това опашките позволяват балансиране на натоварването – ако входящият трафик временно превишава възможностите на обработващите услуги, съобщенията се натрупват в опашката и се обработват постепенно, когато ресурсите са свободни. Този шаблон е известен като конкурентни консуматори (competing consumers) –

множество работещи услуги могат паралелно да четат от една опашка, като всяко съобщение се доставя за обработка само на един от тях. Асинхронната комуникация чрез message queue не само осигурява по-добра скалируемост, но и добавя устойчивост – ако една услуга временно е недостъпна, съобщенията няма да се загубят, а ще изчакат опашката.

Azure Service Bus е напълно управляван клауд брокер за съобщения, предоставящ надеждна доставка на съобщения между отделни приложения и услуги. Service Bus поддържа два основни модела на съобщения: - Queues (опашки): Позволяват point-to-point комуникация – един подател изпраща съобщения в опашката, откъдето едно или повече конкурентни потребители могат да ги получават, но всяко съобщение се обработва само от един получател. Доставката е по принцип FIFO (First-In-First-Out) освен ако не са настроени правила за пренареждане. Както споменахме, опашките осигуряват времево разделяне и възможност за изглаждане на пикове в натоварването. Подателите и получателите работят независимо – подателят не изчаква обработката, което прави архитектурата по-слабо обвързана. - Topics & Subscriptions (топици и абонаменти): Позволяват publish/subscribe комуникация – подателят публикува съобщение към topic (топик), а всички абонирани subscriptions получават копие от съобщението. Това реализира модел "един към много". При публикуване на съобщение в даден топик, всяка активна абонамент (subscription) към този топик получава съобщението самостоятелно, сякаш има собствена виртуална опашка. Този модел е подходящ, когато различни услуги трябва да реагират на едно и също събитие. Например, една услуга може да обработва събитието бизнес-логически, докато друга изпраща нотификации – и двете получават съобщението чрез отделни абонаменти на един топик.

Други важни концепции в Azure Service Bus са правилата за доставка (Receive Modes) – Peek-lock срещу Receive-and-delete. В peek-lock режим Service Bus заключва съобщението при четене и изисква изрично потвърждение за завършена обработка (complete), преди да го премахне от опашката. Това гарантира, че при неуспех съобщението може да бъде доставено повторно (ат-лийст-уанс семантика). При receive-and-delete режим съобщението се счита за обработено в момента на изчитане, без допълнително потвърждение, което е по-просто, но крие риск от загуба при срив (ат-мост-уанс семантика). В нашия проект използваме peek-lock чрез високото ниво ServiceBusProcessor API, което автоматично заключва съобщенията и изисква да ги маркираме като обработени.

Тъй като Azure Service Bus е облачна услуга, директната работа с него изисква активна интернет връзка и може да генерира разходи. За целите на локалната разработка Microsoft предоставя Azure Service Bus Emulator – локален емулатор, който имитира поведението на Service Bus и се разпространява като Docker контейнер. Емуляторът позволява на разработчиците да разработват и тестват своите приложения изолирано, без да зависят от облачната среда. Основните предимства на използването на емулатора включват: Локална работа офлайн: Всички компоненти могат да се изпълняват на локалния компютър, което премахва мрежовата латентност и позволява работа без интернет. Без разходи за облак: Тестването с емулатор не генерира облачни такси, което е особено ценно за академични проекти или етап на разработка. Изолирана тестова среда: Емуляторът осигурява изолирана среда, където експерименти и тестове могат да се извършват без риск да повлияят реални услуги или да бъдат повлияни от външни фактори. Бърз цикъл на разработка: Промени в кода и конфигурацията могат да се тестват веднага на локално ниво, което ускорява цикъла на разработване (inner development loop).

Емуляторът на Service Bus се стартира като контейнер и включва също така контейнер с SQL сървър, който служи за вътрешно хранилище на съобщенията. Следва да се отбележи, че емулаторът има някои ограничения спрямо облачната услуга – например липсва поддръжка на Azure портал, автоматично скалиране, интеграция с Azure Active Directory и др., и не е предназначен за продукционна среда. Също така, данните и опашките създадени в емулатора не персистират през рестартиране на контейнера (те се инициализират наново при всяко стартиране). В контекста на нашия проект, емулаторът ни позволи да разработваме и тестваме EventRegistry изцяло локално, без да е нужна регистрация на истински Azure Service Bus namespace.

# Описание на системата EventRegistry

EventRegistry представлява примерна разпределена система, състояща се от няколко отделни микроуслуги, които комуникират помежду си чрез съобщения в Azure Service Bus. Приложението онагледява сценарий на регистриране на събития и изпращане на нотификации, като използва асинхронна обработка, за да постигне по-добра мащабируемост и надеждност. В общ план, системата позволява чрез публичен REST API да се регистрира дадено събитие (например нова заявка или действие), което след това се обработва във фонов режим и при необходимост се изпраща уведомление до заинтересованите страни. Всички междусервисни комуникации са реализирани чрез Azure Service Bus опашки/топици, което освобождава компонентите от директни зависимости един от друг.

Системата е изградена на принципа на микроуслуги, като всеки основен компонент е отделен проект (и потенциално отделно разгърнат сервиз). EventRegistry.Api – уеб услуга (REST API), която предоставя външните крайни точки. Тя приема входящи HTTP заявки от клиенти. Основната ѝ роля в нашия сценарий е да валидира и приеме заявка за ново събитие, след което да публикува съобщение за това събитие в Service Bus, вместо директно да извършва всички действия синхронно. По този начин API-то бързо отговаря на клиента, че заявката е приета, а по-нататъшната обработка се извършва асинхронно. EventRegistry.Business – модул, капсулиращ бизнес логиката на системата. Той съдържа правилата за обработка на събитията. В нашата реализация Business слоят се използва например за валидиране на данните, трансформиране на обекти и съдържа логиката какво да се направи, когато пристигне дадено събитие (напр. запис в база, изпращане на уведомление). Този проект е структуриран като библиотека, която може да бъде използвана от API-то и/или от фоновите услуги. EventRegistry.Data – модул за достъп до данни. Тук се намират моделите на данните (например класове, представящи "Event") и логиката за работа с хранилище – това може да е имплементация на repository, ORM (напр. Entity Framework) или просто в памет за целите на демонстрацията. В рамките на проектът, Data слоят предоставя на останалите части унифициран начин за запис, чете и обновяване на информацията за събития. EventRegistry.Messaging – отделен модул, отговарящ за интеграцията с Azure Service Bus. Тази библиотека капсулира работата с Azure Service Bus SDK-то – съдържа код за изпращане и получаване на съобщения, настройка на клиенти и опашки, и обща логика по обмена на събития. Идеята е останалите микроуслуги да не работят директно с ниско нивото на Azure.Messaging.ServiceBus, а чрез този модул, което намалява дублирането на код. EventRegistry.EventProcessor – фонов микро-сервис, който се

занимава с обработка на събитията, постъпили от опашката. Това е конзолно приложение или Worker Service, което при стартиране се свързва към Azure Service Bus, слуша за нови съобщения (събития) на определена опашка или топик и при получаване ги обработва. В контекста на нашата система, EventProcessor получава съобщения за новорегистрирани събития (публикувани от API-то), след което изпълнява нужните бизнес стъпки – например извиква EventRegistry.Business логиката за запис в база данни чрез EventRegistry.Data. След успешна обработка, този компонент може да генерира ново съобщение (например събитие "EventProcessed" или "NotificationRequested"), което да предизвика следващи действия. EventRegistry.NotificationService – микроуслуга, отговаряща за изпращане на нотификации. Тя също работи като фонов работник, който слуша за определен тип съобщения – например заявки за нотификация. Когато получи съобщение (напр. че дадено събитие е обработено и трябва да се уведоми потребител). emulator/ – тази папка съдържа конфигурацията за Azure Service Bus Emulator, използван в локална среда. Обикновено това включва Docker Compose YAML файл и допълнителни настройки (като файл config.json за емулатора). Конфигурацията определя как да се стартират контейнерите за емулатора и SQL сървъра, както и параметри като портове и ключове за достъп.

# Организиране на Local Development среда

За да се разработва и стартира системата локално, бяха необходими следните инструменти и настройки. Необходим е .NET (.NET 10) SDK, за да се компилират и изпълняват проектите EventRegistry. Всички микроуслуги (API, EventProcessor, NotificationService) са .NET приложения, написани на C#, така че инсталиран .NET е предпоставка. Понеже използваме Azure Service Bus Emulator, който се доставя като Docker образ, е нужно инсталиране на Docker Engine. Docker ни позволява да стартираме емулатора локално в изолиран контейнер. Уверяваме се, че Docker е настроен и работи (може да се провери с команда docker info). Самият емулатор се тегли като image от Microsoft Container Registry. В нашия проект предоставихме конфигурация (в папка emulator/) за лесно стартиране. Това включва docker-compose.yml, който дефинира два контейнера: servicebus-emulator (с образ mcr.microsoft.com/azure-messaging/servicebus-emulator:latest) и mssql (образ на SQL Server за хранилище). С един Docker Compose файл се стартират и двата. Преди първото пускане е необходимо да се създаде файл с променливи на средата .env, където се попълват:

- ACCEPT_EULA=Y (съгласие с лицензионните условия за емулатора и SQL сървъра),
- MSSQL_SA_PASSWORD=<парола> (силна парола за SA потребителя на SQL, изисква се от образа),
- SAS_KEY_VALUE=<ключ> (опционално, може да се зададе стойността на ключа за достъп до емулатора; ако не се зададе, ще се използва такава по подразбиране от конфигурацията).

След като Docker е настроен, в терминал в директория emulator/ изпълняваме команда: docker-compose up -d. Това изтегля нужните образи (ако ги няма) и стартира контейнерите във фонов режим. По подразбиране емулаторът слуша на порт 5672 (AMQP). След успешен старт, с docker ps може да се провери, че контейнерите servicebus-emulator и mssql са Up. В Docker Desktop също ще видим двата контейнера, като емулаторът ще пише логове (които могат да се разгледат за debug). За да свържем .NET приложенията към локалния емулатор, използваме специален connection string.

Всеки микросервис (API, EventProcessor, NotificationService) трябва да знае този connection string, за да се свърже към Service Bus. Това е реализирано чрез конфигурационни файлове (например appsettings.Development.json) или чрез добавяне на настройка в appsettings.json под ключ, например "ServiceBusConnection": "<connection_string>". По време на разработка сме ползвали user-secrets или директно appsettings за задаване на connection string-а. В продукционна среда би било чрез Secret Manager или Key Vault, но тук е локално. В Startup/Program кода на всеки сервис се зарежда този connection string и се използва за инициализиране на Service Bus клиента.

Сега можем да изпращаме заявки към API-то, които ще задействат обмен на съобщения в емулатора.

# Работа със средата и библиотеките

В разработката на проекта бяха извършени редица настройки и използвани библиотеки, които не се виждат директно в горния код, но са критични за успешна имплементация. Тази секция описва тази "невидима" работа – инсталиране и конфигуриране на нужните пакети, интегриране с .NET средата, срещнати проблеми и направени решения, както и обосновка на избора на емулатор.

За да работим с Azure Service Bus в .NET, избрахме да използваме най-новия официален клиентски пакет Azure.Messaging.ServiceBus. Той предоставя модерни асинхронни API-та и е препоръчителният пакет, за разлика от по-стария Microsoft.Azure.ServiceBus. Инсталацията стана чрез NuGet конзола или Package Manager UI, като във всеки проект, който комуникира с Service Bus (EventRegistry.Messaging, EventProcessor, NotificationService, и Api) беше добавен референтният пакет. С изпълнение на команда като Install-Package Azure.Messaging.ServiceBus или чрез вписване в .csproj, пакетът бе възприет. Допълнително, за административните операции (създаване на опашки) използвахме Azure.Messaging.ServiceBus.Administration, който обаче в новите версии е обединен с основния или достъпен през ServiceBusClient.

.NET платформата (конкретно ASP.NET Core за API и Generic Host за работните услуги) предоставя вграден механизъм за Dependency Injection. В нашия проект се възползвахме от това, за да конфигурираме веднъж Service Bus клиент и да го преизползваме.

Това създава един-единствен екземпляр на ServiceBusClient, който се използва през целия живот на приложението – както препоръчва документацията (клиентът е създаден да бъде thread-safe и да се преизползва, вместо да се създава при всяко изпращане). Аналогично, бихме могли да регистрираме и ServiceBusSender за конкретна опашка, но в нашия случай предпочетохме да създаваме sender динамично при нужда. За EventProcessor и NotificationService, които са фонови процеси, използвахме HostBuilder за .NET Generic Host. В тях, в метода ConfigureServices, регистрирахме нужните услуги: Business логиката (EventService), Data контекста (напр. EventRepository), и ServiceBusClient. Също така, регистрирахме HostedService клас (например EventProcessingWorker), който съдържа описания в секция 5 код за стартиране на ServiceBusProcessor. Този клас имплементира BackgroundService и неговият метод ExecuteAsync стартира processor.StartProcessingAsync().

Благодарение на DI, в конструктора му получава нужните зависимости (_eventService, logger, ServiceBusClient и пр.).

# Проблеми и тяхното решение

Първоначално срещнахме затруднение да свържем приложението към емулатора. Оказа се, че е задължително в connection string-а да присъства UseDevelopmentEmulator=true. Без него, при опит за ServiceBusClient.Connect се хвърля изключение за неправилен адрес или оторизация. Решението беше ясно от официалната документация – добавихме този параметър. Също така уверихме се, че ползваме правилния SharedAccessKey. В емулатора по подразбиране може да се зададе ключ чрез SAS_KEY_VALUE или да вземем стойността от Config.json. За да избегнем несъответствие, копирахме стойността от config файла на емулатора и я поставихме като ключ в нашата конфигурация. Така приложението успя да се оторизира (емулаторът използва същия механизъм на SAS токени както истинския Service Bus). 2. Service Bus Explorer несъвместимост: За да преглеждаме на живо съобщенията в опашките, опитахме да използваме популярния инструмент Service Bus Explorer. Той обаче не успя да се свърже към емулатора, дори с правилния connection string. При проучване установихме, че понастоящем емулаторът не е съвместим с този инструмент. Вместо това, за дебъг целите се наложи да логваме съобщенията при получаване (както се вижда от кода Console.WriteLine("Received: "+body) в примерите). Това беше достатъчно, макар и не толкова удобно колкото графичен инструмент. 3. Инициране на ServiceBusProcessor: Един тънък момент беше да се гарантира, че фоновые услуги (EventProcessor, NotificationService) стартират слушане след като опашките са налични. При първоначално стартиране на цялата система, API-то или EventProcessor може да тръгне, преди емулаторът да е готов или преди опашките да са създадени. Това водеше до грешки при CreateProcessor или при първи опит за получаване. За решение въведохме малко изчакване/retry логика при старта на EventProcessor – ако QueueExistsAsync("events") върне грешка, изчаква няколко секунди и опитва пак, докато емулаторът се инициализира. С Docker Compose профила, алтернативно бихме могли да зададем зависимост: да не стартира EventProcessor контейнер преди емулатора. Но тъй като разработвахме чрез IDE на локалната машина, просто следвахме правилния ред: първо Docker (емулатор), после EventProcessor. 4. Избор на библиотека: Разгледахме дали да ползваме стария пакет Microsoft.Azure.ServiceBus (който има синхронен API и малко по-различен модел). Оказа се, че новият Azure.Messaging.ServiceBus не само е препоръчителен, но и поддържа емулатора напълно. Документация от 2025 г. показва, че Azure SDK е актуализиран да работи с емулатора без проблеми. Затова избрахме него. Това ни позволи да ползваме и улеснения като ServiceBusProcessor и Azure.Extensions (за DI), което ускори разработката.

Защо Azure Service Bus Emulator вместо реален Azure? Основната причина е практичност и спестяване на ресурс при разработката. В университетска или обучителна среда не винаги е възможно студентите да имат Azure акаунти с необходимите права или бюджет. С емулатора можем да разработваме офлайн, бързо и безплатно. Освен това, емулаторът позволява да се тестват и гранични случаи локално – например как системата реагира при недостъпен брокер (можем просто да спрем контейнера, за да симулираме outage). Друга причина е, че при разработка често се налага да пускаме и спираме средата множество пъти – ако това беше реална облачна услуга, бихме натрупали нежелани опашки, съобщения и разход. Емулаторът нулира състоянието си при всеки рестарт, което в случая е удобно за

чисти тестове. Накрая, научихме се да работим с точно същия код, който би работил и срещу истински Azure Service Bus – миграцията към облака би била смяна на connection string (и премахване на UseDevelopmentEmulator флага). Така постигаме среда, максимално близка до реалната, но контролируема локално.

# Заключение

В рамките на този курсов проект успешно разработихме и демонстрирахме система от микроуслуги, комуникиращи помежду си чрез Azure Service Bus съобщения, използвайки локален емулатор. Постигнатите резултати включват: - Реализирана е асинхронна обработка на заявките – вместо директни операции, API модулът публикува събития в опашка, които се обработват от отделни фонови услуги. Това осигурява по-добра мащабируемост и устойчивост на системата. - Изградена е локална среда с Azure Service Bus Emulator, което позволи тестване на функционалността без реална облачна инфраструктура. Научихме се да конфигурираме и използваме емулатора ефективно, включително създаване на опашки, настройка на connection string и ограниченията на емулатора. - Използван е модерен подход с Azure.Messaging.ServiceBus библиотеката и Dependency Injection в .NET, което направи кода по-организиран и модулен. Микроуслугите са ясно разграничени по отговорности (API, обработка, нотификации), а общият код за Service Bus е централизиран. - Системата постигна целта да изпрати автоматично известие след обработка на събитие, без да блокира потребителското изживяване. Времето за отговор на API остава кратко, дори ако реалната обработка (и изпращане на имейл) отнема повече време.

Проектът EventRegistry изпълни поставените цели, като предостави работещ пример за messaging комуникация в .NET среда. Чрез него затвърдихме теоретичните познания за микроуслуги и асинхронни съобщения с едно практическо приложение.

Сорс кодът на приложението е публикуван на: github.com/NovaBG03/EventRegistry