Agenda:

1. Intorduction
2. Why microservices?
3. Types of integration
4. Messaging
5. RabbitMq guaranties and topology
6. Pub/Sub
7. Request/Response
8. Errors and retries
9. Scheduling
10. Saga
11. Courier
12. Riders (Kafka) and NodeJs
13. Introudction

Добрый день! Меня зовут Копосов Антон. Я .Net Engineer в компании Temabit. За время работы в компании занимался бэкенд решениями для мобильных приложений Сильпо, Фора и Фоззи а так же несколькими корпоративными приложениями. Поскольку на момент разработки приложений в компании уже было достаточно много продуктов – возникала необходимость интегрировать существующий функционал с новоиспеченными приложениями.

Так вот сегодня мы поговорим о способах интергации приложений в частности с использованием систем обмена сообщениями, а конкретно об использовании такого фреймворка как Masstransit. Как они сами себя называют A free, open-source distributed application framework for .NET.

1. Why microservices?

Прежде чем начинать разбираться в реализации тех или иных паттернов в Masstransit. Следует ответить на вопрос ЗАЧЕМ? Зачем нужно что-либо куда-либо интегрировать.

Ведь можно просто взять и написать все в одном codebase и запустить это приложение на все случаи жизни. Но как показала природа – это так не работает. Вот самое сложное существо из разряда монолитов(одноклеточных). Это самодостаточный организм, имеет множество функций, но при этом не сумевшее эволюционировать и стать на вершину пищевой цепочки.

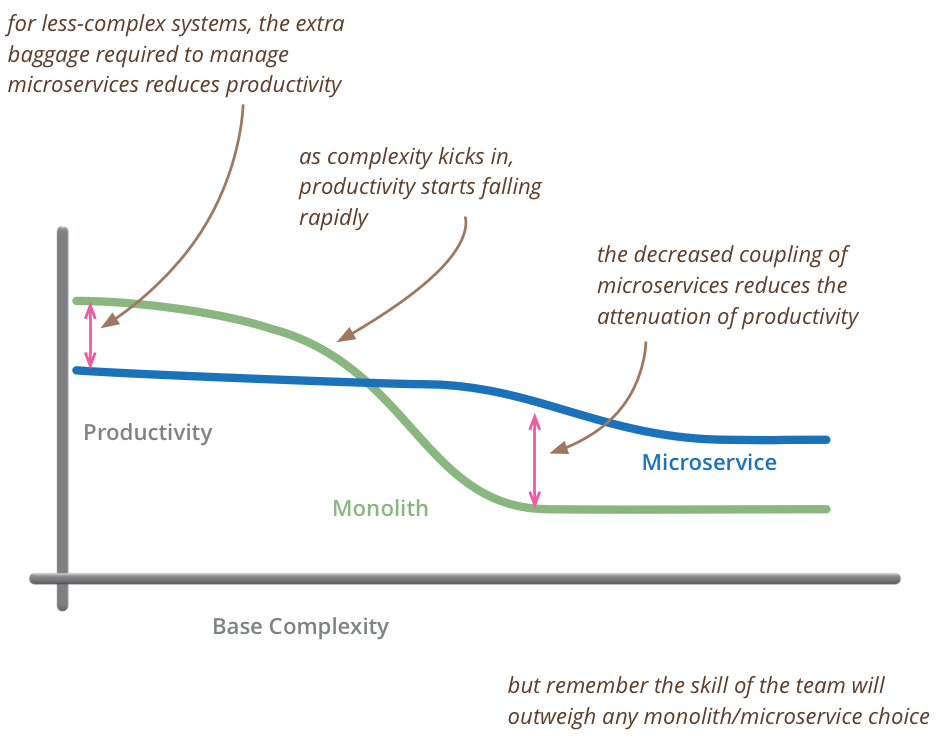


В то же время мы люди и высшие животные пошли по пути микросервисов(многоклеточночти).

А успешный организм может успешно сучествовать только при наличии специализированных клеток и налаженной коммуникации между ними – это гормоны, факторы роста, нейромедиаторы (читай messaging).

Но стойте – это ведь это презентация по разработке ПО, а не в мире животных. По сути чем плох монолит? Он не плох, он хорош до тех пор, пока не становится плох. Это как сложность алгоритмов проявляется только на больших значениях.

Вот картинка, которую вероятно много кто уже видел – продуктивность(фич в секунду) от общей сложности продукта.



А что конкретно не так монолитом. Почему получается такой график?



Разделение монолита на микросервисы помогает решить такие задачи:

1. Распеределение нагрузки (масштабировать только то, что нужно масштабировать)
2. Отложенная обработка
3. Разнесение ответственности

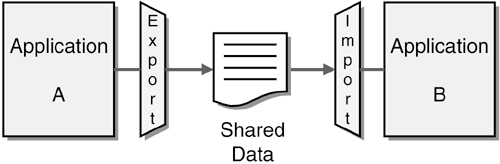
И даже для монолита интеграция скорее всего понадобится с другими приложениями.

Итак, поскольку сколько-либо ценное приложение должно быть сложным. Большая сложность в проэкте – неизбежна. А значит оптимально адоптить микросервисы, а их уже нужно каким-то образом интегрировать между собой.

Какие же виды интеграции существуют?

1. Обмен файлами (хардкор, для тех, кто не знает ни одну базу)
2. Shared Database
3. Rest/Rpc
4. Messaging

**Обмен файлами:**

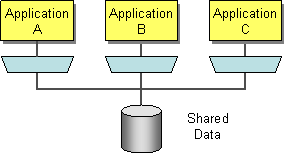


1. Дешево, сердито.
2. Небезопасно (никакой транзакционности, проблемы с concurrency, проблемы с безопасностью)
3. Несогласованность формата
4. Несогласованность во времени

Одним словом, лучше чем ничего.

**Shared Database:**

Данные, которыми нужно поделиться – лежат в общей базе данных.



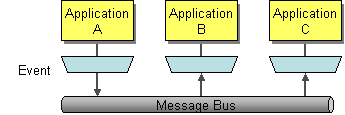
1. Популярно.
2. Транзакционно
3. Сильно связно (эволюционирование схемы базы затруднено)
4. Размазывание логики по сервисам. No cohesion

**RPC/Rest:**

1. Популярно.
2. Данные и поведение инкапсулировано в конечных сервисах
3. Необходимость обрабатывать таймауты.
4. Сложность обработки исключений при большой цепочке вызовов.

**Messaging:**

Сервисы соединены с общей системой обмена сообщений. Обмениваются данными и инициируют поведение путем отправки сообщений.



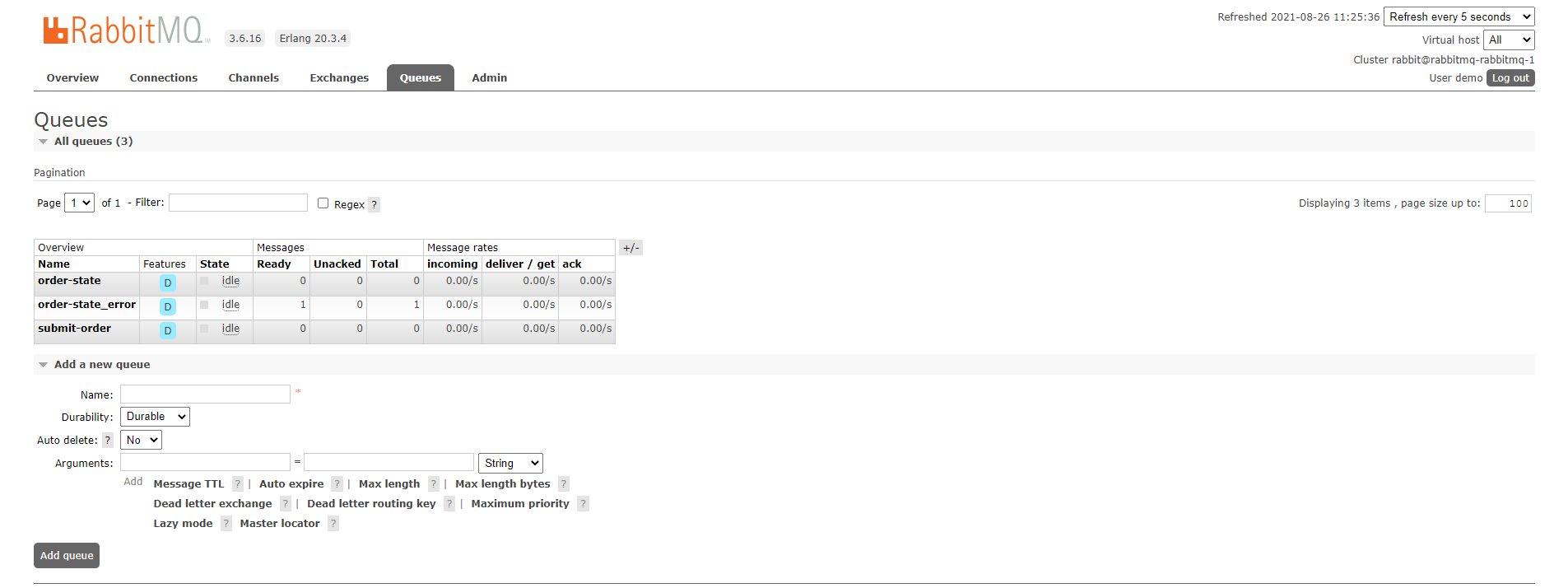
1. Ассинхронность
2. Lose coupling. Сервисы ничего не знают друг о друге, а только посылают сообщения в единую точку маршрутизации.
3. Надежность
4. Realtime

Обмен сообщениями зарекомендовал себя, как лучший способ интеграции для микросервисной архитектуры. Слабая связность и четкие границы сервисов позволяют независимо разрабатывать и развертывать сервисы. Но есть и минусы – дополнительный компонент брокер сообщений. Брокер сообщений как раз и обеспечивает основные фишки интеграции с помощью обмена сообщениями.

Одним из самых популярных брокеров является RabbitMq. И Masstransit полностью поддерживает этот брокер наряду с другими (AzureServiceBus, ActiveMQ) а так же с версии 7 имеет частичную поддержку Kafka с помощью такой штуки как Rider.

Как же работает RabbitMq?

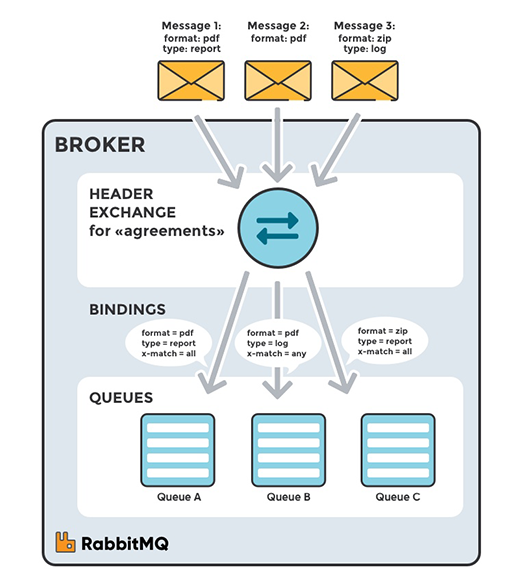
Это отдельный компонент, который можно поставить у себя на машине или развернуть в докере. Так очень удобной штукой является плагин rabbitmq\_management для просмотра ексченджей и очередей в браузере.

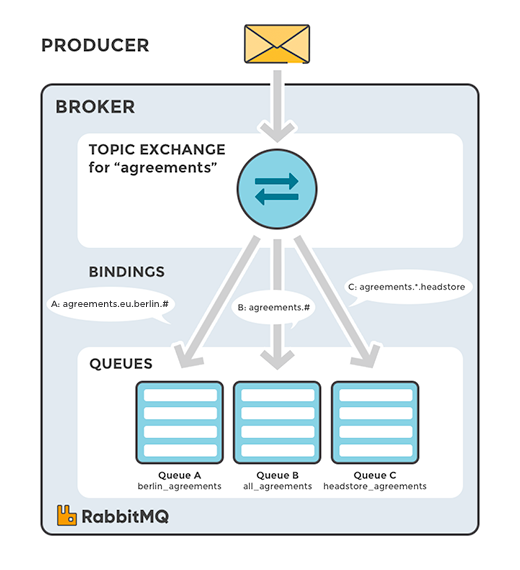
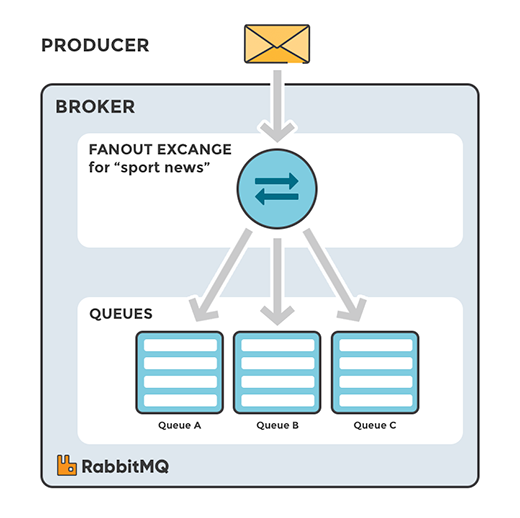
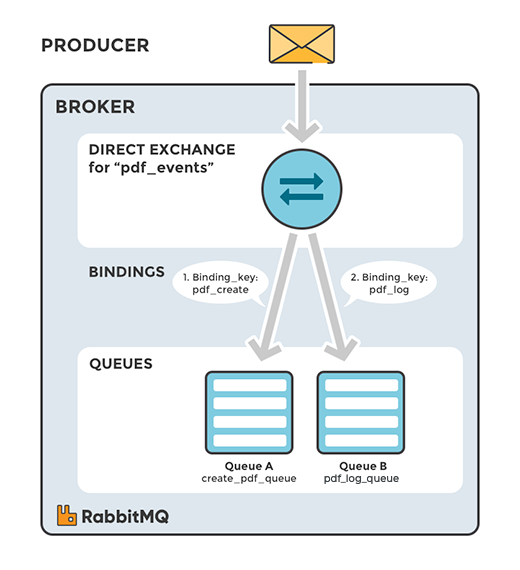


Основные примитивы в ребите – это очереди и ексченджи.

Сообщение попадает в указаный ексчендж, а дальше маршрутизируется в зависомости от типа байндинга и ключа маршрутизации в подходящую очередь(и).

Типы биндингов:





Гарантии:

The super simplified overview:

* Publishers send messages to exchanges
* Exchanges route messages to queues and other exchanges
* RabbitMQ sends acknowledgements to publishers on message receipt
* Consumers maintain persistent TCP connections with RabbitMQ and declare which queue(s) they consume
* RabbitMQ pushes messages to consumers
* Consumers send acknowledgements of success/failure
* Messages are removed from queues once consumed successfully

RabbitMQ offers "at most once delivery" and "at least once delivery" but not "exactly once delivery" guarantees.

Теперь посмотрим что же дает нам Masstransit.

**What does MassTransit add to the transport?**

MassTransit is a lightweight service bus for building distributed .NET applications. The main goal is to provide a consistent, .NET friendly abstraction over the message transport.

[#](https://masstransit-project.com/usage/transports/" \l "concurrency)**Concurrency**

MassTransit is completely asynchronous and leverages the .NET Task Parallel Library (TPL) to consume messages concurrently to achieve maximum throughput and high server utilization.

[#](https://masstransit-project.com/usage/transports/#connection-management)**Connection management**

The network is unreliable. If the application is disconnected from the message broker, MassTransit takes care of reconnecting and making sure all of the exchanges, queues, and bindings are restored.

[#](https://masstransit-project.com/usage/transports/#exception-retries-and-poison-messages)**Exception, retries, and poison messages**

Your message consumers don't need to know about broker acknowledgement protocols. If your message consumer runs to completion, the message is acknowledged and removed from the queue. If you throw an exception, MassTransit uses a retry policy to redeliver the message to the consumer. If the retries are exhausted due to continued failures or other reasons, MassTransit moves the message to an error queue. If the message did not reach a consumer due to being misrouted to the queue, the message is moved to a skipped queue.

[#](https://masstransit-project.com/usage/transports/#serialization)**Serialization**

C# is a statically typed language, and developers work with types. RabbitMQ works with bytes. So how do you format a message over the wire? How do you handle different date/time formats (local, UTC, unspecified)? How do you deal with numbers, are they integers, longs, or decimals? MassTransit has already thought about this and implemented sensible defaults for you. And there are many serializers provided out of the box, including JSON, BSON, and XML as well as the .NET binary formatter as a last resort.

You can even protect your messages using AES-256 encryption, to keep prying eyes away and to ensure the safety of private information (to meet PCI or HIPAA requirements).

[#](https://masstransit-project.com/usage/transports/#message-header-and-correlation)**Message header and correlation**

Designing a common message envelope can be a nitty-gritty affair until things stabilize. And MassTransit is already stable having been used in production since 2008. The format is [well documented](https://masstransit-project.com/architecture/interoperability) and has been tested with billions of messages. Furthermore, the envelope includes headers for tracking messages, including conversations, correlations, and requests. The address and host information in the envelope make it easy to build any messaging pattern.

[#](https://masstransit-project.com/usage/transports/#consumer-lifecycle-management)**Consumer lifecycle management**

MassTransit handles consumer creation and disposal, and integrates with most major dependency injection containers using their built-in lifetime scope management. This ensures that dependencies are created and destroyed as part of the message consumption pipeline.

[#](https://masstransit-project.com/usage/transports/#routing)**Routing**

MassTransit provides a heavily production tested convention for using RabbitMQ exchanges to route published messages to the subscribed consumers. The structure is CPU and memory friendly, which keeps RabbitMQ happy.

[#](https://masstransit-project.com/usage/transports/#easy-unit-testing)**Easy Unit Testing**

One of the first rules of unit testing is to avoid hitting infrastructure. And RabbitMQ is just that. MassTransit includes a high-performance in-memory transport for testing every consumer using the same code that would be used in production. And the MassTransit.TestFramework NuGet package includes test harnesses that handle the setup and teardown of the bus so you can easily test your message consumers and sagas.

[#](https://masstransit-project.com/usage/transports/#sagas)**Sagas**

Sagas are a powerful abstraction that supports message orchestration with durable state. Whether you use the original somewhat explicit syntax, or the powerful state machine syntax of **Automatonymous**, you can build highly available distributed workflow and coordination services easily. MassTransit supports both Entity Framework and NHibernate, using code-based mapping and migrations to simply code deployments and upgrades.

[#](https://masstransit-project.com/usage/transports/#scheduling)**Scheduling**

MassTransit has strong integration with Quartz.NET, to make it easy to schedule messages for future delivery. This brings distributed applications into the fourth dimension, making time a first-class citizen. Some incredibly powerful routing systems have been built by the authors using Quartz in combination with other MassTransit features.

There are also other scheduling providers that are supported by MassTransit, such as RabbitMQ deferred messages and Azure Service Bus scheduled enqueueing.

[#](https://masstransit-project.com/usage/transports/#monitoring)**Monitoring**

Keeping an eye on your services performance is critical, and having the right tools is a huge plus. MassTransit updates a range of performance counters as messages are processed so operations can keep an eye on message flow and compare the throughput to that of RabbitMQ.