Webhooks applied to Microservices

Abstract

Microservices are a software architecture inspired by service-oriented computing that has recently started gaining popularity and has been showing remarkable results in the development of large-scale systems. Contrarily, webhooks have been around for some time but they do not have a big adoption for the amount of potential they have, mainly because solutions use Application Programming Interfaces (API) to do their job, which is inefficient and does not give the same result. APIs have natural limitations to receive real-time events that webhooks solve and when embedded together, allow providers to offer a better and richer service to the customers. Microservices can also benefit from the low latency, lightweight, event-driven communication that webhooks provide. This allows microservices, that know little or nothing about each other, to communicate and act when an event happens and to act upon that without having to use heavy third-party technologies and dependencies.

Keywords: Software Architecture; Software Development; Software Engineering; Distributed Systems;

1. Introduction

The technological advance that has been witnessed in the last decades has led to a great change in the way we think and develop systems and software. Nowadays, any organization, independent of its business area, needs to realize that to better serve and deliver more value to its customers in a competitive way, it must embrace the world of information technologies. However, this is not enough to keep up, because if organizations want more growth and to stay relevant, they have to invest in technologies that give them a market advantage. This led to the demand for new technologies to increase, which also elevated the number of users, requiring applications to grow and presenting new challenges to software providers. Some of these challenges are additional complexity in software integration, delivery, and deployment, and they start suffering from "dependency hell". One of the biggest ones is scalability, which, motivated by this growth in the digital world and users, tends to be more common and needs improved solutions.

It is from these challenges and requirements that the Microservices Architecture (MSA) emerged. In software engineering, architecture is what allows systems to evolve and provide a certain level of service throughout their life cycle. It is concerned with providing a bridge between system functionality and requirements for quality attributes that the system has to meet. Over the past decades, software architecture has been thoroughly studied. As a result, software engineers have come up with different ways to compose systems that provide broad functionality, satisfy a wide range of requirements, are fault-tolerant, highly available, scalable, and replicable systems that support this new digital world and the highest expectations of customers globally.

The MSA is an architecture originated from the Service-Oriented Computing (SOC) and it can be defined as a set of small, autonomous, and loosely coupled services, each of which is independent and should implement only a single business capability or domain (Richardson, 2014).

Over the years, another area that also had a huge advance was the Web. It evolved considerably, becoming increasingly interconnected due to components, specifically Application Programming Interfaces (API). They allow us to use features and data from one site in a completely different one (SendGrid, 2013). The data was another aspect that evolved a lot in the past few years, being nowadays considered the new gold. More than ever, we live in a world full of information where having the right data at the right time is the key to achieve market advantage. With this increase in data complexity, it is increasingly necessary to transmit data in real-time. Besides, with the amount of data flowing on the internet every minute it becomes difficult to keep an application up to date with the best and latest data. This is something that the APIs do not allow because they only provide an answer when the programmer makes a request, that is, to obtain data in real-time we need to be constantly making requests to an API.

There are some approaches, like the short and long polling, that try to accomplish it more efficiently (Biehl, 2017). In the short polling, the client asks at regular intervals for a new status from the server, however, if the server does not have new data available, it just keeps sending an empty response. Some events may only happen once in a while, so we have to figure out how to make the calls or we might miss it. In the case of the long polling, the client sends an initial polling request, but the established Hypertext Transfer Protocol (HTTP) communication channel is kept open and the server does not send any response back until there is data available. The problem with the latter approach is that HTTP calls cannot be blocked for an arbitrary period and, eventually, they will timeout and the client still needs to send a request to be ready for receiving the next event.

Apart from that, they both have disadvantages in common, such as they still overload the server even when there are no changes in the data, open a lot of HTTP connections, and it does not scale well. Besides, these approaches still have some latency and we are unable to find out, in real-time, when an action was performed, for example, a user was created or when an error was launched in a solution in our monitoring service. Thus, this model does not meet all the needs of today's solutions and the evolving expectations of API consumers.

To solve all those limitations, we have the webhooks. They are the most widely used technique for realizing events, and a more proactive and flexible approach, for both the provider and the consumer, when compared to the APIs. They are like a Really Simple Syndication (RSS) feed for events that, when an action happens, an event is triggered in real-time and a notification is sent to all the subscribers of that event, without requiring user interaction. An application subscribes to an event by providing a callback Uniform Resource Locator (URL) to which the application generating the events will send a notification. Hence, webhooks do not come to replace APIs, but to complete them with new concepts, like events, subscriptions, triggers, and notifications. Moreover, webhooks can also benefit from implementations already made for APIs, namely the OAuth internet protocol, allowing platforms built on the APIs to tie into the activity of their users (Leach, 2017). This makes the service provided even more complete and efficient since these concepts are not natively supported by the REST architecture, allowing it to respond to more use cases.

That said, webhooks are a loosely coupled architecture that allows creating new innovative integrations and powerful workflows between different services that know little or nothing about each other (Stamat, 2019). This definition is perfectly aligned with the MSA's one and could even be an alternative definition of it. Most MSAs implement APIs, however, only a few of them use the power of webhooks as a lightweight mechanism to respond to the events triggered on the microservices. These, when applied to the MSA, reduce interactions that are sometimes unnecessary between the different microservices that make up the architecture and also allow interacting with microservices outside the architecture.

2. LITERATURE REVIEW

2.1. Microservices Architecture

There are a lot of good definitions made by different authors about the MSA. Chris Richardson stated that an MSA is a software architecture that structures an application as a set of services that are easily manageable and testable, loosely coupled, independently deployed and organized by business capabilities or business domains (Richardson, 2014).

One of the most complete and well accepted is Martin Fowler's one. In his view, the MSA style is an approach to developing a single application as a suite of small services, each running in its own process (independent) and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable, scalable, and tested by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies (it is a completely separate service too) (Fowler & Lewis, 2014).

It is important to note that the word micro in microservices does not concern the size or complexity of the service, but rather the scope of the domain to which it relates. It is expected that a microservice only concerns a very specific functionality and performs it excellently, following, thus, the developmental ideology Single Responsibility Principle (Martin, 2003). Additionally, it is probably more important to think about how many services we are capable of supporting operationally than it is to think about how small they are because it is better to have slightly bigger ones and fewer of them if we do not have fully automated deployment into production (continuous deployment and delivery).

Figure 1 is an example of a simple food delivery application that implements the MSA. As we can observe, the application is composed of six services that interact with each other, and their relationships. They are loosely coupled and each one of them has its database. There is a front-end microservice that consumes and orchestrates the way the services are consumed, depending on the request.

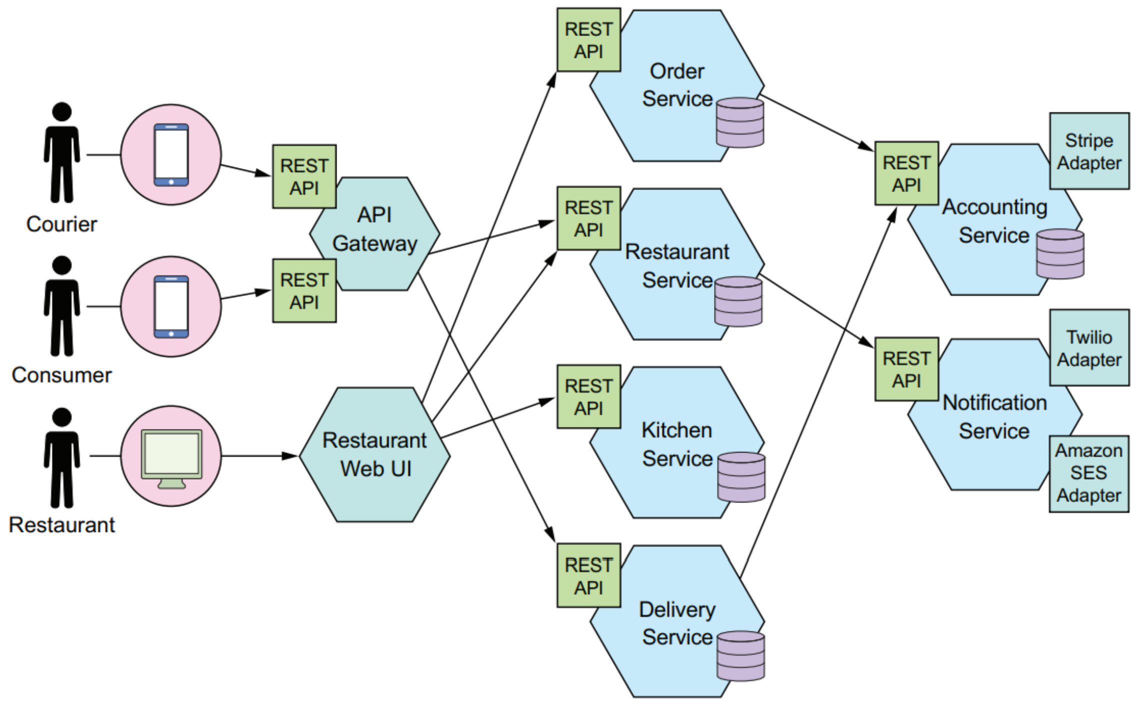

Figure 1 - Example of an MSA (Richardson, 2018)

2.1.1. Principles and patterns

Principles and patterns create a basis in an architecture that we can follow to correctly implement it.

Principles help us frame the various decisions we have to make when building our systems and the development of microservices is based on a set of them. They can be adopted entirely like that or tweaked to make sense depending on the organization's needs. However, the whole should be greater than the sum of the parts, and if we decide to drop one of them, we should, firstly, understand the impact it will have. Thus, the more aligned and refined the adoption of these principles, the more hypothetically successful the process of moving to an MSA-based application is.

It is very important to take these principles into account in the implementation process, as, according to many architecture experts, in the long run ignoring some of these can have major implications for managing the resulting solution. It may ultimately become more difficult to manage the microservice solution compared to its monolithic predecessor. Newman (2015) enumerated these principles and they are business domains-based modeling, automation culture, hide internal implementation details, decentralization of decision-making power and accountability, independent deployment (DEP), isolate fault, and highly observability.

Another important aspect of the architecture implementation is the correct choice of patterns. They address issues encountered when applying the MSA. Their choice is what defines how the principles above described will be followed. Richardson (2014) said that the MSA pattern language consists of numerous groups of patterns, such as core, decomposition, data management, deployment, cross-cutting concerns, communication which also includes discovery and style, security, testing, observability, and UI. This choice is not always easy, as some patterns have major implications and the choice of one may lead to the exclusion of another. The implementation of a pattern can also imply acceptance of the eventual inherent consistency of, for example, asynchronous communication. It is easily concluded that the wrong choice of these standards can have negative consequences on the resulting application/system.

2.1.2. Advantages and disadvantages

The MSA copes with a lot of issues from other architectures. Moreover, the correct implementation of the MSA brings a lot of improvements and advantages to the organization, such as:

- They implement a limited number of functionalities, which makes their codebase small. This limits the scope of a bug, makes it easier for a developer to understand it and also makes the IDE and application start faster, making developers more productive and speeding up the deployment;
- They are independent, so a developer can directly test and investigate their functionalities in isolation concerning the rest of the system;
- It is possible to plan gradual transitions to new versions of a microservice. The new version can be deployed "next" to the old one, and the services that depend on the latter can be gradually modified to interact with the former. This fosters continuous integration and greatly eases software maintenance;
- Changing a module of an MSA does not require a complete reboot of the whole system. The reboot regards only the microservice that needed to do it;

- Enables the continuous deployment and delivery of large, complex applications. This will improve the maintainability, testability, and deployability of a software;
- Microservices naturally lend themselves to containerization, which has a high degree of freedom in the configuration of the deployment environment that best suits their needs (both in terms of costs and quality of service);
- Scaling an MSA does not imply duplication of all its components, and developers can conveniently deploy/dispose instances of services with respect to their load (Gabbrielli et al., 2016);
- Improved fault isolation. In case of a problem and timeout on a service, the others will continue to handle requests. Contrarily, one misbehaving component of a monolithic architecture can bring down the entire system. Moreover, at the infrastructure level, with containerization the virtual environment of the service is delimited, so the resources that it may erroneously allocate are limited to the level of his virtual environment (Richardson, 2018); and
- The freedom to choose a lot of the tooling and resources on a case-by-case basis gives the flexibility to make informed decisions based on the right tooling for the case in question. The only constraint is on the technology used to make the network of interoperating microservices communicate.

The MSA is not a silver bullet that will solve all the organization's problems. So, it also presents some disadvantages, like:

- Developers must deal with the additional complexity of creating a distributed system;
- Quoting Fowler and Lewis we are shifting the accidental complexity from inside our application in glue code in our components and modules within our application out into the infrastructure (Fowler & Lewis, 2014). However, new technologies are being constantly released to help us fight this problem;
- The communication protocols and communication channels to be used, as well as asynchronous communication, also represent a source of additional complexity;
- Testing is more complex as it has to be performed at the microservice and consumer level. When launching a new version of a service, it should ensure, that none of the consumers get their needs unmet;
- It leads to increased memory consumption, due to the fact that it allocates memory for each service, which, in sum, occupies more space; and

- In addition to all the inherent technological complexity, there is also a process of organizational change, in which service-holding teams should be able to manage their entire life cycle independently. They should be multidisciplinary, which implies working together and bypasses the typical department concept (Richardson, 2018).

2.2. Webhooks

A server application changes its state due to interactions with the users or interactions with the environment, allowing clients to read or change its application state via API calls. The next step is to allow clients to observe changes in the application state and react to state changes via events. Therefore, there is a relation between the action of the API and the events that can be observed.

There is not an extensive literature review about webhooks because they have a simple, yet, powerful concept. It is an HTTP POST that occurs when something happens, sending an event notification to the HTTP callback that was listening to that event. They are the most widely used approach for realizing events and are a loosely coupled architecture that allows creating new innovative integrations and powerful workflows between different services that know little or nothing about each other (Stamat, 2019).

The most basic patterns for realizing events are the interrupt pattern and polling pattern. In the interrupt pattern, the client is notified by an external event source when something interesting happens and the external event source is responsible for the event execution. In approaches with the polling pattern, the client needs to figure out when something interesting happens, so the complete responsibility for event execution is with the client (Gao, 1996). Therefore, webhooks implement the interrupt pattern because it is the external event source that notifies the consumer when something that they subscribed to happened. Moreover, as seen previously, APIs can implement the polling pattern but still have some limitations.

Even though webhooks solve some limitations that the APIs present, the goal is not to replace them but to complete them. Providing events that are well-coordinated with the available APIs is a best practice that enriches the provided service. To apply this coordination, we can (Biehl, 2017):

- Recognize that there is a relation between the actions that can be performed via API and the state transitions that can be observed via events so, for each method, endpoint, and action an API provider exposes it should strive to provide a matching event. In general, one can say the better the coverage, the more opportunities for integration exists, and the more attractive the API becomes as a product; and
- The APIs for managing webhooks need to blend in naturally with the overall API portfolio. The APIs for managing webhooks should follow REST constraint as far as

possible, and they should be protected with the same security mechanisms as other APIs in the portfolio.

This allows API providers to create a competitive advantage, allowing them to differentiate in the market and providing optimal possibilities for integration.

2.2.1. Events and subscriptions

The two main elements of webhooks are the events and subscriptions. The events are the actions produced by the application and the subscriptions are all the events that the consumers subscribed to and, with this interaction, the client indicates its interest in receiving these events. For creating new webhooks events and subscriptions, we should treat the subscription like any other resource in an HTTP API.

Regarding the events, a good webhook service should provide as much information as possible about the event that is being notified, as well as additional information for the client to act upon that event more easily. In the case of an MSA, it should also identify which microservice is producing it, because different microservices can produce similar events. Moreover, a type attribute should be included to help consumers manipulate the event, even if they are not being sent to a single endpoint (Iacobelli, 2016).

Regarding subscriptions, the consumer has to set up an appropriate event receiver endpoint that can process events according to the specifications of the API provider When webhooks are built we should think about the consumer that will receive the data. Giving them the chance to subscribe to different events under one single URL is not the best approach because it limits them and can create problems.

2.2.2. Security

Some questions regarding security must be addressed. The first one is when an attacker tries to impersonate the sender and fake events to the receiver. If the consumer's application exposes sensitive data, it can verify that requests are generated by the true sender and not a third-party pretending to be him. There are numerous ways to solves this problem. If we want to put the work on the consumer side, we could opt to request from a whitelisted IP address, but an easier method is to set up a secret token and validate the information.

Another security concern is that a faked event can be injected, or the data of an existing event can be manipulated. One way to mitigate this risk is to make the sender sign the event payload with a shared secret. This guarantees the legitimacy of both the event payload and the sender. Normally, this kind of validation is passed through the HTTP header. Moreover, and with the use of timestamps, this can also solve the risk of an attacker recording the traffic and playing it back later.

The last risk is when an attacker uses a fake receiver to collect events. This can be solved by requiring the consumer an HTTPS endpoint with an SSL certificate, allowing for the sender to verify its identity. Moreover, sometimes self-signed certificates are used for receiver endpoints, however, they are not signed by a trusted authority and are thus now trustworthy (Biehl, 2017).

2.2.3. Advantages and disadvantages

Webhooks cope with numerous limitations that the APIs present and also has some advantages on its own, such as:

- It removes pressure from both the consumer and provider because the consumer does not need to make constant periodic calls to the API to know when there is a new change available, which overload both parties;
- They provide a simple and lightweight, yet powerful, way of implementing in real-time streaming of events with low latency;
- They allow to create more powerful integrations and workflows between different two identities that know little or nothing about each other;
- Webhooks are ubiquitous across every programming language and framework which means that everyone can receive a webhook without having to use dependencies (Leach, 2017);
- When embedded with the already implemented APIs, they create a better service and experience for both the consumer and provider; and
- They can use and take advantage of some components already implemented for the APIs, such as the OAuth protocol, documentation, etc., getting more out of the already implemented functionalities.

Due to its reverse flow, when compared to the APIs, webhooks present some obvious disadvantages, like:

- For bureaucratic reasons, it can be hard to create an endpoint to receive a webhook, especially in big companies where this needs to be negotiated between infrastructure and security teams. Moreover, webhooks might be wholly incompatible with the organization’s security model because they can be receiving data from a third-party application (Leach, 2017);
- We rely on the consumer to develop a fluid and error-free endpoint for the whole action to be performed as intended. However, there can be transmission failures, variations in latency, and quirks in the provider’s implementation meaning that even if webhooks are sent to an endpoint ordered, there are no guarantees that they will be received in that same order;

- Version upgrades are already a problem in APIs because when providers upgrade them, they can be incompatible with the consumer integration. However, in this case, the consumer can explicitly request a new version and verify if their integration works before upgrading, otherwise, they can stick to the older version while they upgrade their application to integrate with the new one. In the case of webhooks this is not that simple because the provider has to decide in advance what version to send to the consumer and this may lead to problems;
- It puts pressure on the provider's infrastructure, and it can increase if it has millions of outgoing webhooks and some of the consumers do not setup endpoint correctly, leading to retries and latency and, consequently, to a degraded system for everyone; and
- It is inefficient in terms of communication because it represents one HTTP request per event, which compared to APIs is still better but can be improved with some tricks and techniques.

3. ARCHITECTURE PROPOSAL

Based on the literature review and aforementioned good practices for webhooks, plus some from other technological areas, a webhooks architecture was created to solve the APIs limitations and to help to implement and manage webhooks in an MSA. It serves as an intermediary that registers and manages all available events and subscriptions. It manages the database and supports the task of sending notifications for all subscribers when events are triggered and, in the event of a failure, implements a system to send the notification again until the subscriber successfully receives it.

There are two different approaches to implement this solution in an MSA. The first is to create an independent microservice that centrally controls all webhooks operations and is part of the architecture. Each microservice that wants to use webhooks will communicate with it to execute the desired operation. The second approach is through the implementation of these same functionalities, but in a library directly in each microservice, thus, every operation that they want to execute is within its scope and they do not need to communicate with other microservices.

There are some differences between both implementations. One of them concerns the databases. In the first case, the database is shared between all the microservices that are interacting with the webhooks microservice, whereas for the second case, each microservice only works with its database, however, we can still have a shared webhooks database in this case. Depending on the use case it might be preferable to choose one or another. The first approach is better if we want to centralize the webhooks, it will make it easier to manage them and it does not overload the microservices itself because it isolates an operation in an individual one. The second approach is preferable if we have light microservices that do not require a lot of processing power. Since this architecture creates a standard in webhooks, the second approach gives the same results as the first one but without the ability to centrally manage all of them.

Figure 2 presents the proposal for a general and versatile architecture that can be applied in both approaches and that implements a standard to both produce and consume the webhooks. It was developed taking into account the best practices and concepts for webhooks described in the literature review and uses technologies, design patterns, and best practices from other areas, namely REST and object-oriented programming. Moreover, by applying a standard between provider and consumer we can solve some of the disadvantages presented previously.

A service can interact with the webhooks, independently if they are implemented as a microservice or a library. Inside, the endpoints and database necessary to fully manage the events and subscriptions, and interact with them, are generated on the startup of the microservice. To fully encapsulate the interactions with the database, and to not create dependencies on the implementation based on the database technology we are using, the mediator pattern and CQRS (Command Query Responsibility Segregation) were implemented using the mediatR library. The event bus is also subscribed on the startup, here we use an in-memory one, and it is used to create a pipeline between an action that is called and the subscribers of that action. When an event is triggered with the event bus it is published to a queued background service that will get all the subscriptions for that event from the database and will send the notification with the payload to the subscribers. All these notifications are individually saved to a webhooks log table in the database and a timed background service will pick up the ones that were not successfully sent and will retry to send them to the subscribers until they receive them.

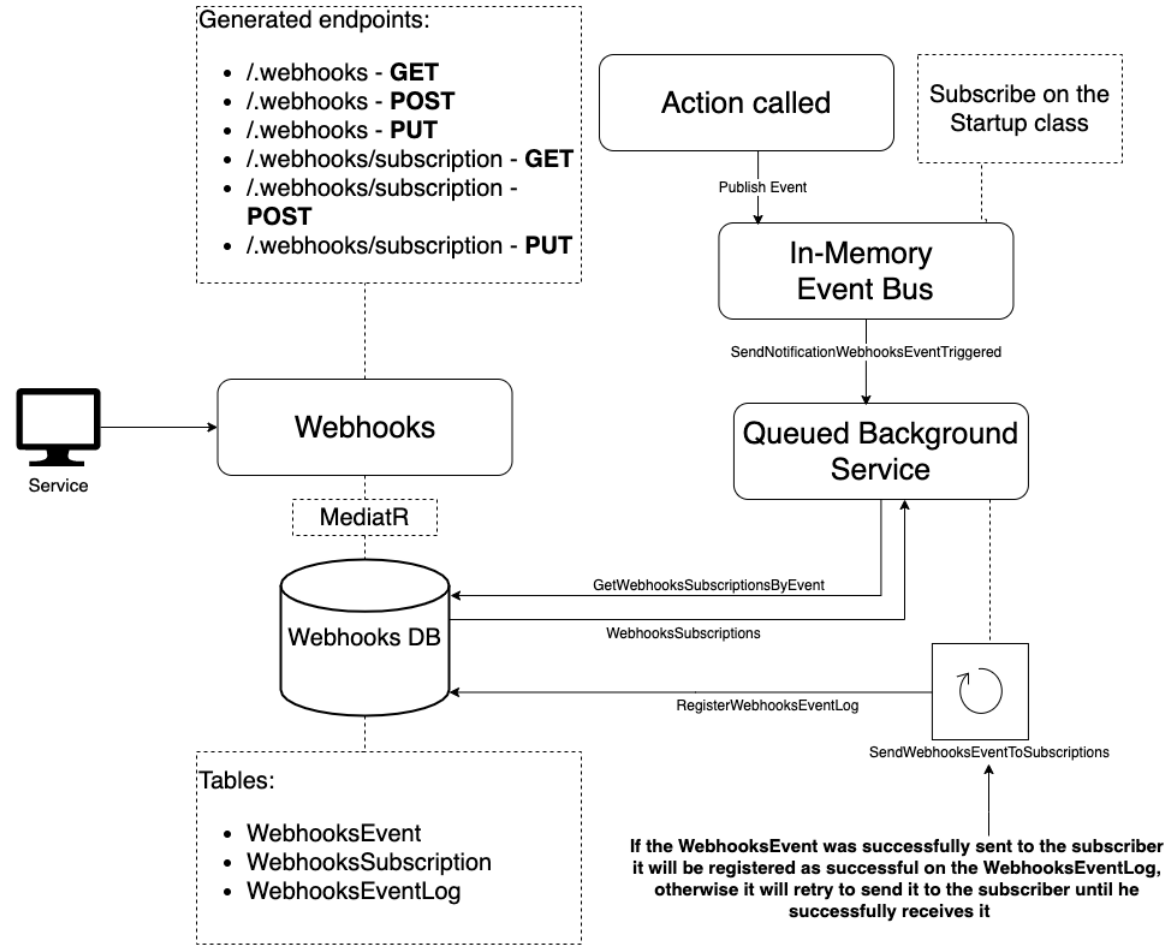
Generated endpoints:
/.webhooks - GET
/.webhooks - POST
/.webhooks - PUT
/.webhooks/subscription - GET
/.webhooks/subscription - POST
/.webhooks/subscription - PUT
Action called
Subscribe on the Startup class
Publish Event
In-Memory Event Bus
SendNotificationWebhooksEventTriggered
Service
Webhooks
Queued Background Service
MediatR
Webhooks DB
GetWebhooksSubscriptionsByEvent
WebhooksSubscriptions
RegisterWebhooksEventLog
SendWebhooksEventToSubscriptions
Tables:
WebhooksEvent
WebhooksSubscription
WebhooksEventLog
If the WebhooksEvent was successfully sent to the subscriber it will be registered as successful on the WebhooksEventLog, otherwise it will retry to send it to the subscriber until he successfully receives it

Figure 2 - Webhooks proposed architecture

To explore the architecture more in-depth, Figure 3 shows the entity-relationship diagram for the database tables. One of the best practices mentioned previously was to use a type attribute for the events to help consumers manipulate them, however, we did not include it here because we aimed to a more standard solution that does not require extensive setup. Instead, a general class to manage all the notifications was developed.

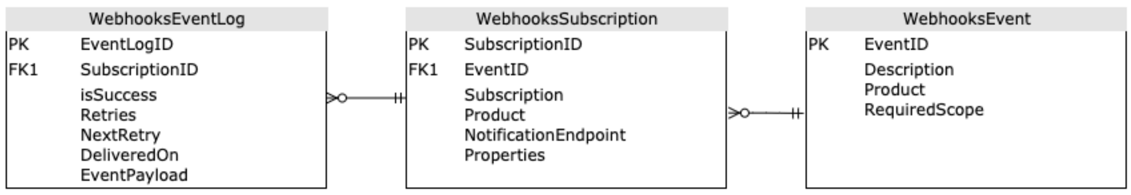

Figure 3 - Entities-Relationship Diagram for the proposed architecture

Concerning security, an attribute “RequiredScope” was added to the event. This way we can manage the webhooks permissions, allowing only the users that are authorized to access that scope to subscribe to that event, and consequently, receive it. For that, we can use tools like Identity Server to help with the authorization process, scopes management, secret generation, and API security in general.

Once the architecture is well implemented, depending on the kind of use case that the webhooks are going to be applied to, it could be a good option to create a UI to help handle and manage events and subscriptions.

3.1. Future scalability improvements

Over time it is normal for the solution to grow a lot, either if we opt for the first or second approach. In order not to overload the microservice and create bottlenecks it is necessary to scale it. Figure 4 presents a solution to scale the microservice through data partition, using the product (microservice) ID. This way we manage to scale the webhooks microservice efficiently and we take more advantage of this model because we can also play with the databases to make this flow even more flexible.

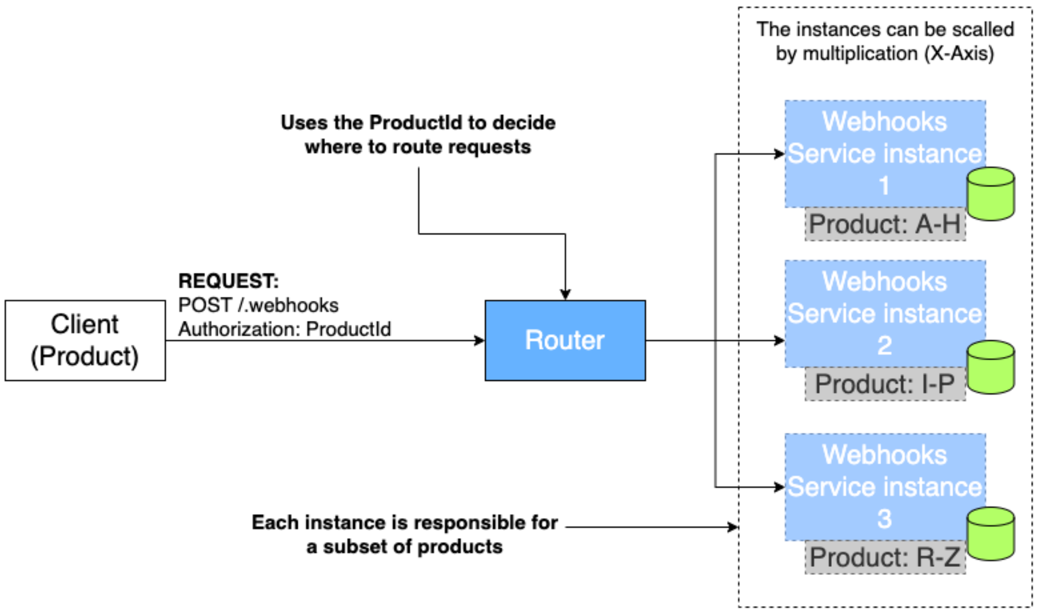
The instances can be scalled by multiplication (X-Axis)
Uses the ProductId to decide where to route requests
REQUEST:
POST /.webhooks
Authorization: ProductId
Client (Product)
Router
Webhooks Service instance 1
Product: A-H
Webhooks Service instance 2
Product: I-P
Webhooks Service instance 3
Product: R-Z
Each instance is responsible for a subset of products

Figure 4 - Webhooks architecture with data partitioning

3.2. Real case application

This architecture was already tested in a company that develops business management software and had a necessity due to their recent migration from a monolithic architecture to an MSA. They had no way of making their microservices act whenever new interesting actions occurred, either inside or outside the architecture. Moreover, since the company is not going to work with event-driven communication technology, the simpler and more efficient the solution the better, because those technologies require a lot of process power and it can overload the architecture.

Especially in this business area, there are a lot of good use cases for the webhooks because there are numerous interesting events that, once triggered, can create a lot of chained functions to act upon that. Plus, being able to act upon those events in real-time is an enormous market advantage and benefits both the company and its clients.

The webhooks were implemented internally in the company for their use and to test if they solved this necessity. To ease their adoption, the company incorporated them using the library in their SDK that automatically generates the code for all the microservices created. This approach guarantees that even if different developers create different microservices the webhooks are always going to be implemented the same way.

The company in which it was applied uses the Microsoft stack (C#, .NET Core, ASP.NET Core). Middleware is software that is assembled into an app pipeline to handle requests and responses (Anderson & Smith, 2020). The order in which the middleware is built matters because the next middleware will receive the state from the previous one and depends on it to work properly. Figure 5 represents the location of the webhooks in the ASP.NET Core middleware pipeline. It was placed after the "Session" middleware because, in this case, it was a company requirement, however, it would work if it was placed right after the "Authentication" middleware. Independently of the technology stack, if it uses the MVC (Model View Controller) pattern and has a middleware pipeline it can be implemented the same way.

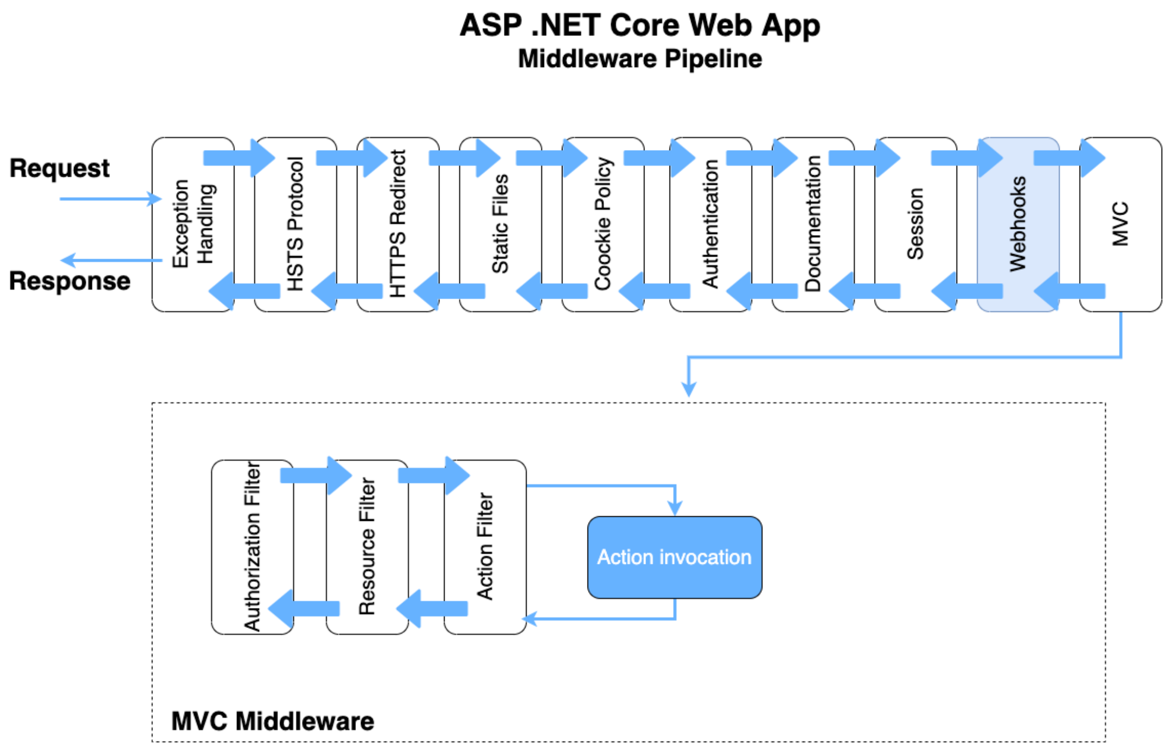

Figure 5 - Standard ASP .NET Core Web App Middleware Pipeline (Adapted from Anderson, R., & Smith, S. (2020))

4. Conclusion

Webhooks and microservices have well-aligned concepts and goals and work well together. They thrive for a common result enabling the creation of powerful workflows and new integrations between them and even agents from outside the architecture.

APIs have some limitations that we can solve by integrating webhooks with them. These limitations come from the fact that APIs are not able to retrieve data in real-time when something happens and to get that data the programmer has to use techniques, such as the short or long polling, which do not fully solve the problem and still have their limitations. Webhooks come to solve them because they work like a "reverse-API" where the application is the one that sends requests to the client when something happens, creating a zero-latency event dispatcher. They are the most widely used technique for realizing events because they are lightweight, easy to set up, simple to use, and do not need third party dependencies to work. When we put APIs and webhooks together they embed perfectly with each other allowing API providers to offer a better and richer service to the customer.

When we apply webhooks to the MSA, they are especially helpful because we can use them to make microservices send events to each other, creating more powerful integrations and workflows without having to use an event-driven communication technology that is usually process-intensive and complex and will overload the architecture.

To help to implement and standardize webhooks in an MSA an architecture was idealized. The proposed architecture was developed, tested, and validated through a real case application. It solves the aforementioned limitations for APIs and helps to integrate webhooks in MSAs to take advantage of the event-driven communication more easily. It was also tested in a real case application where a company had some necessities, has seen previously, and they were able to solve them. This resulted in the company being able to act whenever an event is triggered and act upon that instantly and with little resource usage. Not only did this architecture solved the necessities, but it also opened new opportunities to create different strategic approaches in the future, therefore, this webhooks architecture is still implemented in the company's MSA.

In the future, there is still room for improvement, namely on the architecture proposed for future scalability improvements. Even though it is a good evolution as it is, it still can be enhanced and benefit from the implementation of more scalability techniques and data management. Moreover, for the current architecture, we can apply some tricks to send more than one event per HTTP request. If an application has to send millions of events to the subscribers, it has to use a lot of resources by opening one channel per request, while if it used one channel to send multiple ones it would be more efficient.

References

Anderson, R., & Smith, S. (2020, April 6). ASP.NET Core Middleware. Retrieved from https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?WT.mc_id=-blog-shchowd&view=aspnetcore-3.1

Biehl, M. (2017). Webhooks–Events for RESTful APIs (Vol. 4). API-University Press.

Fowler, M., & Lewis, J. (2014, March 25). Microservices. Retrieved from https://martinfowler.com/articles/microservices.html

Gabbrielli M., Giallorenzo S., Guidi C., Mauro J., Montesi F. (2016) Self-Reconfiguring Microservices. In: Ábrahám E., Bonsangue M., Johnsen E. (eds) Theory and Practice of Formal Methods. Lecture Notes in Computer Science, vol 9660. Springer, Cham

Gao, G. R., Hum, H. H. J., Theobald, K. B., Tian, X. M., & Maquelin, O. (1996, May). Polling watchdog: Combining polling and interrupts for efficient message handling. In 23rd Annual International Symposium on Computer Architecture (ISCA'96) (pp. 179-179). IEEE.

Iacobelli, G. (2016, November 23). Webhooks do's and dont's: what we learned after integrating 100 APIs. Retrieved from https://restful.io/webhooks-dos-and-dont-s-what-we-learned-after-integrating-100-apis-d567405a3671

Leach, B. (2017, September 28). Should You Build a Webhooks API? Retrieved from https://brandur.org/webhooks

Martin, R. C. (2003). Agile software development: principles, patterns, and practices. Prentice Hall PTR.

Newman, S. (2015). Building Microservices: [designing fine-grained systems]. Beijing: O'Reilly.

Richardson, C. (2014). Microservices Pattern: Microservice Architecture pattern. Retrieved from https://microservices.io/patterns/microservices.html

Richardson, C. (2018). Microservices patterns: With examples in Java.

SendGrid. (2013, June 13). Why Every API Needs Webhooks. Retrieved from https://sendgrid.com/blog/why-every-api-needs-webhooks/

Stamat, D. (2019, April 25). 7 Reasons Webhooks Are Magic. Retrieved from https://blog.iron.io/7-reasons-webhooks-are-magic/