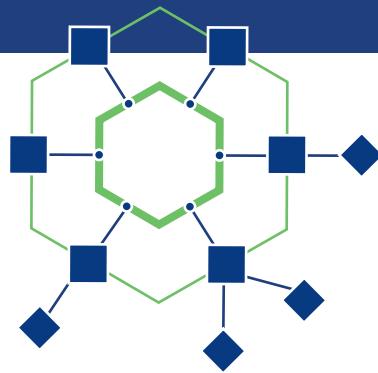




divante

Microservices Architecture for eCommerce

Piotr Karwatka
Mariusz Gil
Mike Grabowski
Aleksander Graf
Paweł Jedrzejewski
Michał Kurzeja
Antoni Orfin
Bartosz Picho



| Foreword

Name a technology conference or meetup and I'll tell you about the constant speeches referencing microservices. This modern engineering technique has grown from good old SOA (Service Oriented Architecture) with features like REST (vs. old SOAP) support, NoSQL databases and the Event driven/reactive approach sprinkled in.

Why have they become so important? Roughly speaking, because of what scale systems achieve nowadays and the number of changes that are deployed on a daily basis.

Of course microservices aren't a panacea. I've tried to make this book as informational and candid as I can. Although we promote the microservices architecture across the following chapters, please also take a look at Appendix 1 authored by Spryker's Co-Founder Alexander Graf with a very candid and pragmatic view on this topic.

This book is a rather "technical one" - starting with some Business rationale for microservices and then stepping into the engineers' shoes and trying to show you the tools and techniques required to build and scale modern eCommerce systems.



Divide and conquer

The original Zalando site was built on Magento using PHP, and at one time was the biggest Magento site in the world. The German eCommerce giant that employs over 10,000 people and ships more than 1,500 fashion brands to customers in 15 European countries generated \$3.43 billion in revenue last year. With over 700 people on its engineering team, they moved to microservices in 18 months.

The key advantages of the microservice approach are:

- **Faster Time to Market** - because of the decentralized development process and opportunities to innovate given to each separate development team.
- **Less is more** - the microservices approach leverages the Single Responsibility Principle which means that a single microservice performs exactly one business function. Therefore developers can create more efficient, clear and testable code.
- **Domain Expertise** - business features are granularly split into separate micro-applications. You'll have separate services for promotions, checkout and products catalog. Each development team typically includes business analysts and developers. It builds engagement and speeds up development.
- **Accountability** - Booking.com's approach to development is to promote the teams whose features are published for production (before the features are usually proven to increase conversion). By working on



the basis of microservices you'll have separate teams accountable for particular KPIs, providing SLA's for their parts, etc. A side effect of this approach is usually the rise of employee effectiveness and engagement.

- **Easier outsourcing** - because services are separable and usually contracts between them have to be well documented, it's rather easy to use ready-made products or outsource particular services to other companies.

| Change is too slow

It's something I usually hear when starting a new consulting engagement. After a few years in the market, enterprises tend to keep the status quo, and try to keep everything running smoothly, but nowadays it's not sufficient to become a market leader. It's crucial to experiment, change, test and select the best solutions. But it's extremely hard to work like that with a team of a few dozen engineers and extremely sophisticated business rules coded to the metal by thousands of lines of code. The microservices approach became so popular because it breaks this into smaller, self-sufficient and granular areas of responsibility that are easy to test and deploy.

| In eCommerce: your software is your company

“

Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations.

— M. CONWAY



Among all the technical challenges, microservices usually require organizational changes inside the company. Breaking the technical monolith quite often goes hand in hand with dividing enterprise departments into agile, rapid teams to achieve faster results. In the end, the final outcome is that processes that took a few months can now be executed in weeks and everybody feels engaged. It's something you cannot underestimate.

Omnichannel

To fulfill your customer's expectations about omnichannel, you have to integrate each and every piece of information about products, shipments, stocks and orders, and keep it up to datefresh. There is no single system to deal with POS applications, ERP, WMS and eCommerce responsibilities. Of course, I've seen a few that pretend to be a One-stop solution but I've never seen anything like that in production. The key is to integrate systems that are optimal for their niches and already integrated within your existing processes. Microservices are great for such an evolutionary approach. We'll describe a case study - where by exposing the APIs from PIM, CRM, ERP and creating a dedicated UI facade, we leveraged on this approach to provide a sophisticated B2B solution.

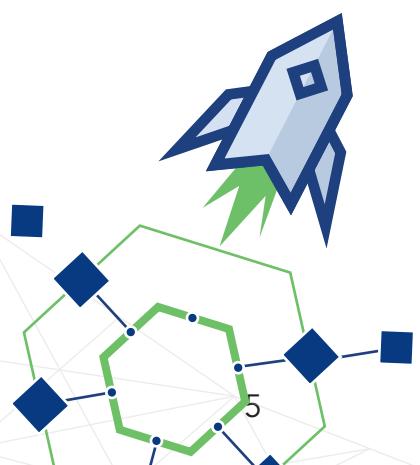
This eBook will try to help you decide if it is time for applying this approach and how to start by referencing to few popular techniques and tools worth following.

Let's get started!

Piotr Karwatka, CTO at Divante



[Go to Table of Contents](#)



About the authors



Piotr Karwatka

CTO at [Divante](#). My biggest project? Building the company from 1 -> 150+ (still growing), taking care of software productions, technology development and operations/processes. 10+ years of professional Software Engineering and Project Management experience. I've also tried my hand at writing, with the book "E-Commerce technology for managers". My career started as a software developer and co-creator of about 30 commercial desktop and web applications.

Michał Kurzeja

CTO and Co-Founder of [Accesto](#) with over 8 years of experience in leading technical projects. Certified Symfony 3 developer. Passionate about new technologies; mentors engineers and teams in developing high-quality software. Co-orgaizer of Wrocław Symfony Group meetups.

Mariusz Gil

Software Architect and Consultant, focused on high value and high complexity, scalable web applications with 17+ years of experience in the IT industry. Helps teams and organizations adopt good development and programming practices. International conference speaker and developer events organizer.

Bartosz Picho

eCommerce Solution Architect, responsible for Magento 2 technology at Divante. Specialized in application development end 2 end: from business requirements to system architectures, meeting high performance and scalability expectations. Passionate technologist, experienced in Magento 1 and 2, both Community and Enterprise editions.



Antoni Orfin

Solutions Architect specialized in designing highly-scalable web applications and introducing best practices into the software development process. Speaker at several IT conferences. Currently responsible for systems architecture and driving DevOps methodology at Droplr.com.

Mike Grabowski

Software Developer and open source enthusiast. Core contributor to many popular libraries, including React Native, React Navigation and Haul. Currently CTO at Callstack.io. Travels the world teaching developers how to use React and shares his experience at various React-related events.

Paweł Jędrzejewski

Founder and Lead Developer of [Sylus](https://Sylus.io), the first Open Source eCommerce framework. Currently busy building the business & ecosystem around the project while also speaking at international tech conferences about eCommerce & APIs.

Alexander Graf

Co-Founder and CEO of Spryker Systems. Alexander Graf (*1980) is one of Germany's leading eCommerce experts and a digital entrepreneur of more than a decade's standing. His widely-read blog Kassenzone ("The Check-Out Area") has kicked off many a debate among commerce professionals. Alexander wrote Appendix 1 to this book.



Acknowledgement

I believe in open source. This book was intended to be as open as possible. I would like to thank all the enthusiasts engaged in this project - giving me honest feedback, helping with editorials etc.

Mateusz Gromulski, Will Jarvis, Ian Cassidy, Jacek Lampart, Agata Młodawska, Tomasz Anioł, Tomasz Karwatka, Cezary Olejarczyk

Thank you guys!

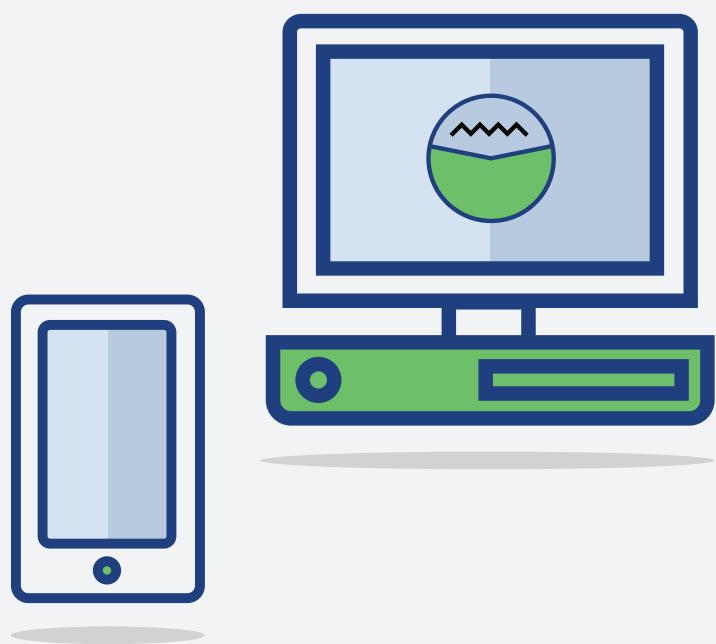
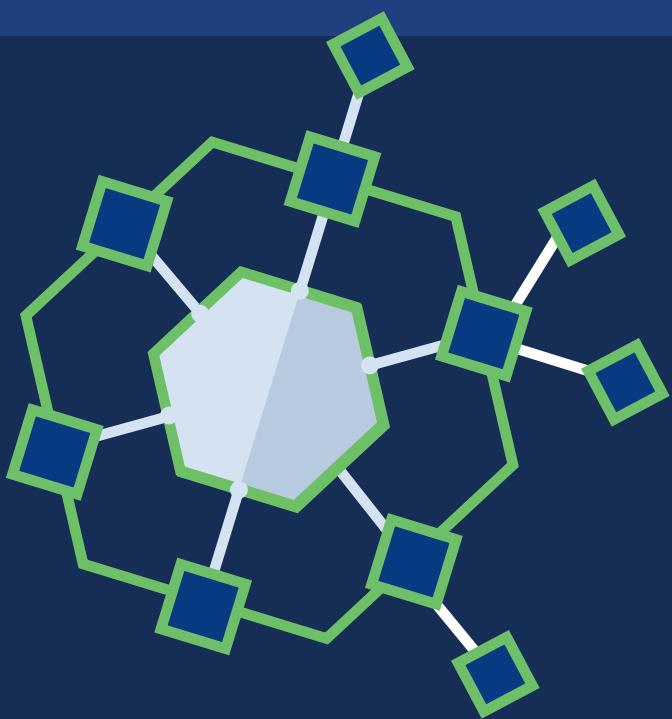


Table of contents

Foreword	2
Divide and conquer	3
Change is too slow	4
In e-Commerce: your software is your company	4
Omnichannel	5
About the authors	6
Table of contents	10
Microservices	11
The criticism	14
Evolutionary approach	16
Best practices	21
Create a Separate Database for Each Service	22
Rely on contracts between services	24
Deploy in Containers	24
Treat Servers as Volatile	25
Related techniques and patterns	27
Design patterns	30
Integration techniques	39
Deployment of microservices	50
Serverless - Function as a Service	60
Continuous Deployment	69
Related technologies	72
Microservices based e-commerce platforms	73
Technologies that empower microservices architecture	77
Distributed logging and monitoring	91
Case Studies: Re-architecting the monolith	98
B2B	99
Mobile Commerce	110
Blogs and resources	112

Microservices



Microservices

Microservice architecture structures the application as a set of loosely coupled, collaborating services. Each service implements a set of related functions. For example, an application might consist of services such as an order management service, an inventory management service, etc.

Services communicate using protocols such as HTTP/REST or (a less popular approach) using an asynchronous approach like AMQP. Services can be developed as separate applications and deployed independently. Data consistency is maintained using an event-driven architecture because each service should have its own database in order to be decoupled from other services.

The most common forces dictating the Microservice approach¹:

- Multiple teams of developers working on a single application.
- System must be easy to understand and maintain/modify, no matter the number of changes deployed.
- Urgency for new team members to be productive.
- Need for continuous deployment (although possible to achieve with monolith design, microservices include some features of DevOps approach by design).
- Scalability requirements that require running your application across server clusters.
- Desire to adopt emerging technologies (new programming languages, etc.) without major risks.

¹ According to: <http://microservices.io/patterns/microservices.html>



The assumptions of the orthogonal architecture followed by microservices architects implies the following benefits:

- Each microservice could be deployed separately and without shutting down the whole system.
- Each microservice can be developed using different technologies while allowing them to publish HTTP end-points (Golang based services can interoperate with PHP, Java...).
- By defining strict protocols (API), services are easy to test and extend into the future.
- Microservices can be easily hosted in the cloud, Docker environments, or any other server platform, and can be very easily scaled as each service can live on its own server(s), VPS(es) etc.
- The services are easy to replace.
- Services are organized around capabilities, e.g., UI, front-end, recommendation, logistics, billing, etc.

The scalability and deployment processes of microservice-based systems can be much easier to automate compared to monolithic architectures. The DevOps approach to infrastructure along with Cloud services is commonly in use. The examples of Spotify and Netflix² inspire IT engineers to implement continuous delivery and monitoring.

² <https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/>



Dockerization of IT environments, monitoring tools and DevOps tools (Ansible, Chef, Puppet and others) can take your development team to the next level of effectiveness.

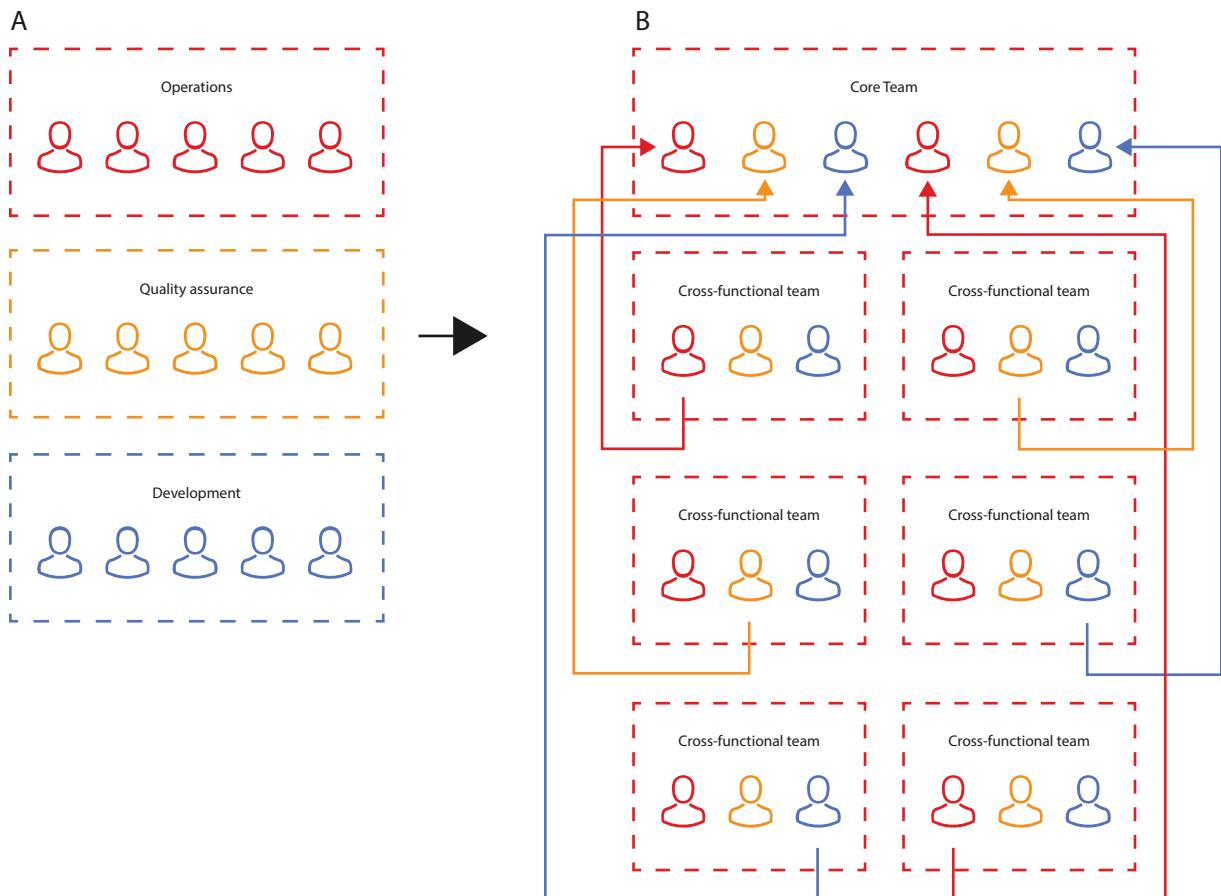


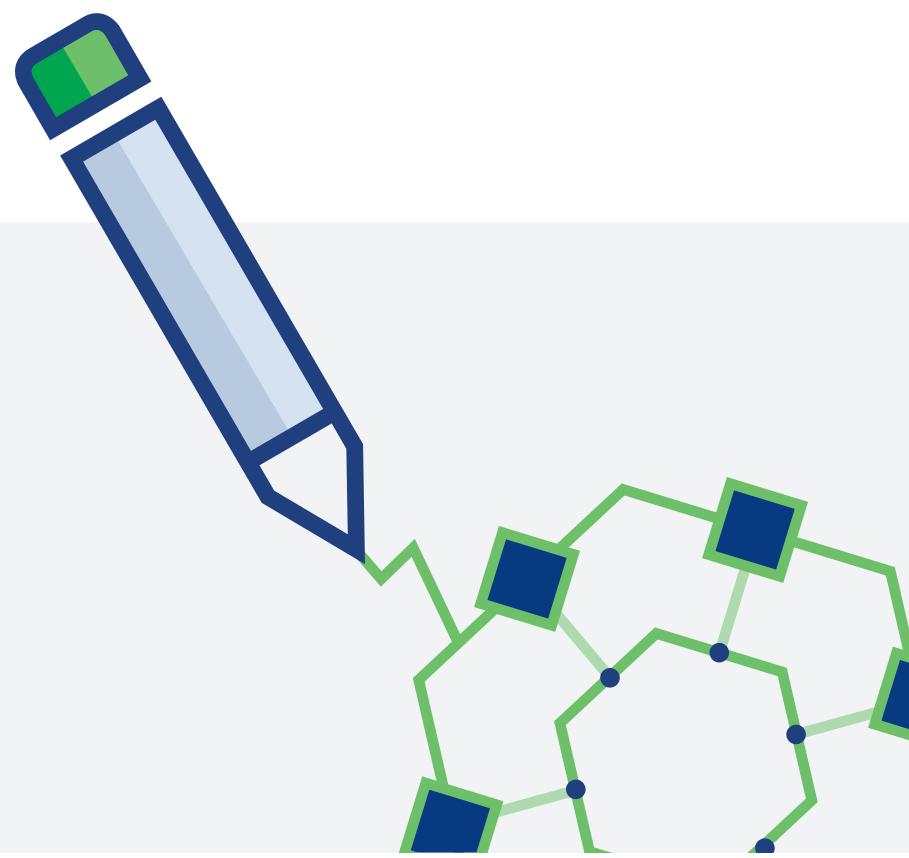
Fig. 1: A microservice approach encourages enterprises to become more agile, with cross-functional teams responsible for each service. Implementing such a company structure, as in Spotify or Netflix, can allow you to adopt and test new ideas quickly, and build strong ownership feelings across the teams.

The criticism

The microservice approach is subject to criticism for a number of issues:

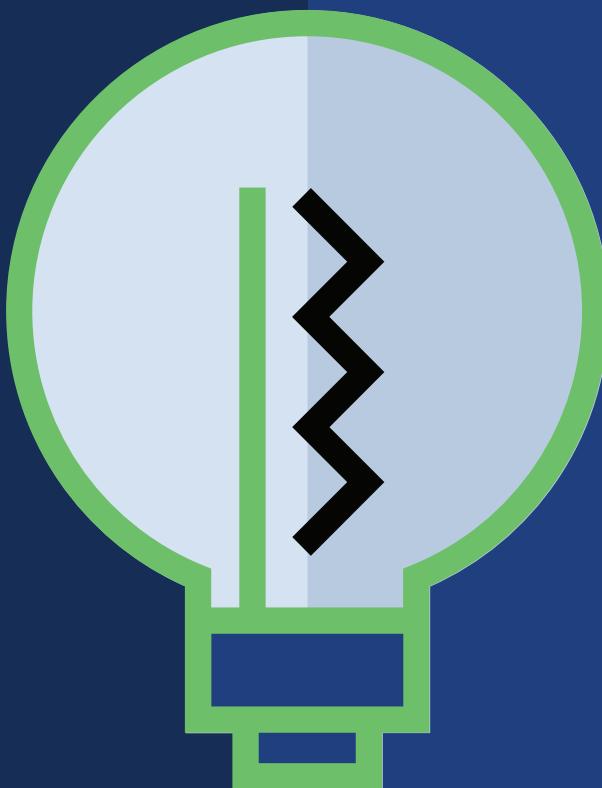


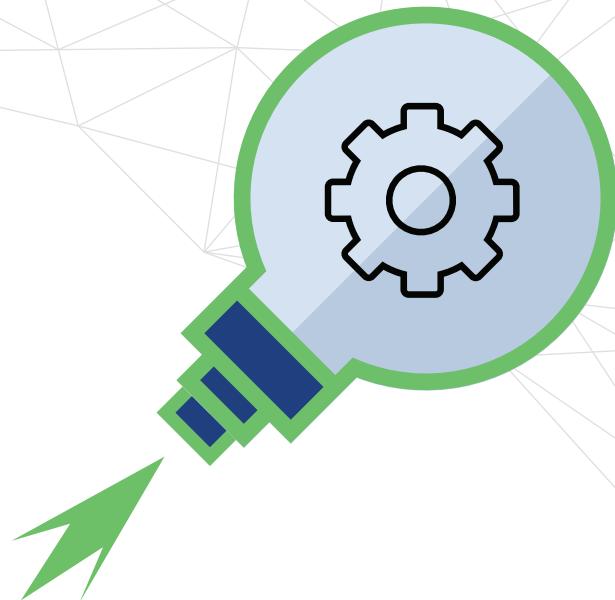
- The architecture introduces additional complexity and new problems to deal with, such as **network latency, message formats, load balancing, fault tolerance and monitoring**. Ignoring one of these belongs to the "fallacies of distributed computing".
- Automation is possible but in the simplest cases, tests and deployments may be more complicated than with the monolithic approach.
- **Moving responsibilities between services is difficult.** It may involve communication between different teams, rewriting the functionality in another language or fitting it into a different infrastructure. On the other hand, it's easy to test contracts between services after such changes.
- Starting with the microservices approach from the beginning can lead to too many services, whereas the alternative of internal modularization may lead to a simpler design.





Evolutionary approach





| Evolutionary approach

Martin Fowler, one of the pioneers³ of microservices used to say:

“

Almost all the successful microservice stories have started with a monolith that got too big and was broken up.

Almost all the cases where I've heard of a system that was built as a microservice system from scratch, has ended up in serious trouble.

³ <https://martinfowler.com/articles/microservices.html>

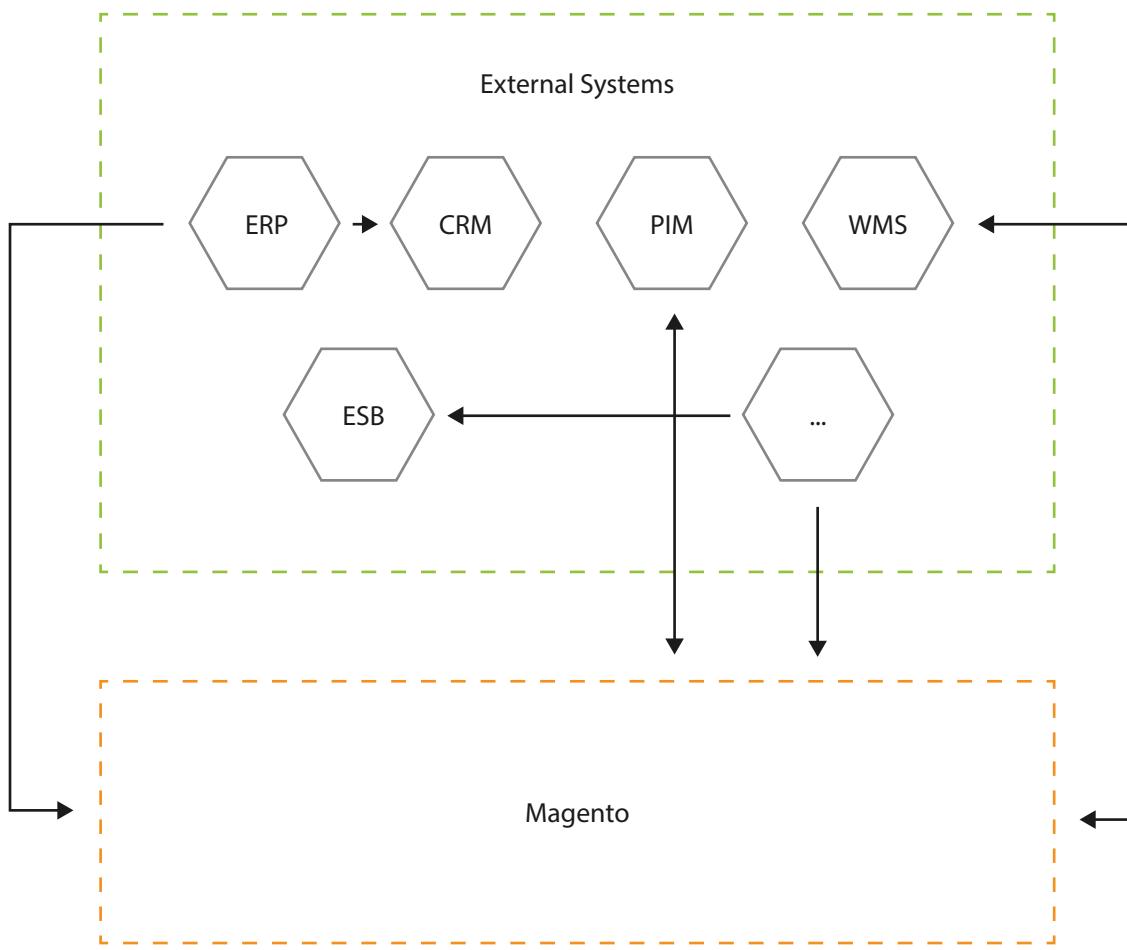


Fig. 2: Initial, monolithic architecture began after 4 years of development of a large-scale, 100M EUR/yr B2B platform.

When you begin a new application, how sure are you that it will be useful to your users? Starting with microservices from day one may significantly complicate the system. It can be much harder to pivot if something didn't go as planned (from the business standpoint). During this first phase you need to prioritize the speed of development to basically figure out what works.



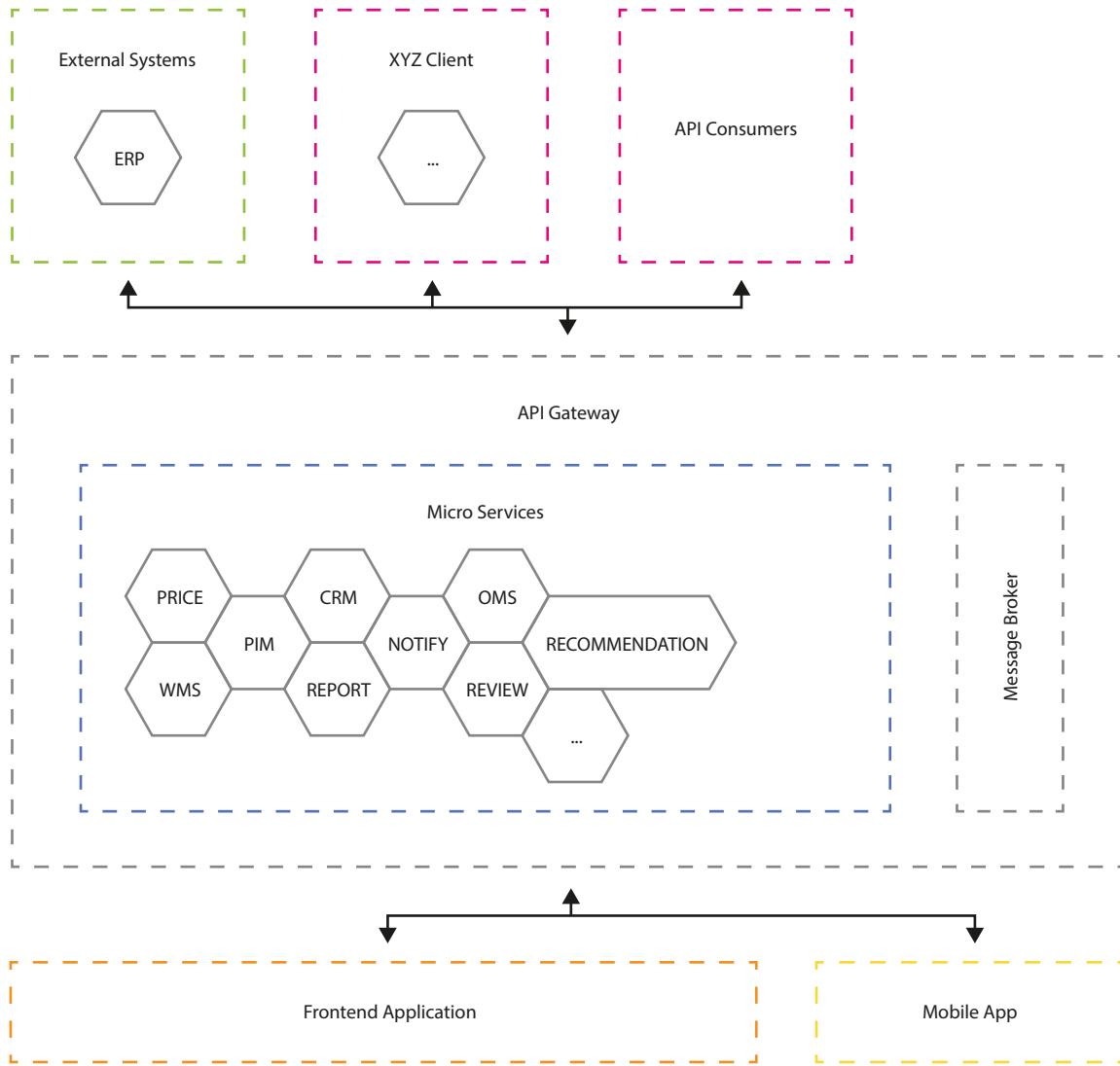


Fig. 3: The very same system but after architecture re-engineering; now the system core is built upon 10 microservices.

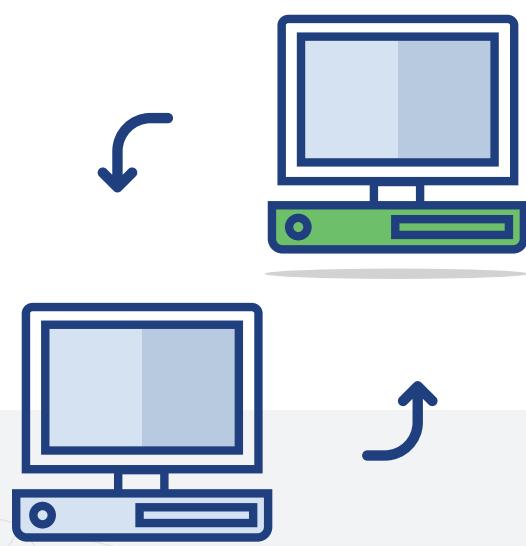
Many successful eCommerce businesses (if not all of them!) started from monolithic, at some point, all-in-one platforms before transitioning into a service oriented architecture.

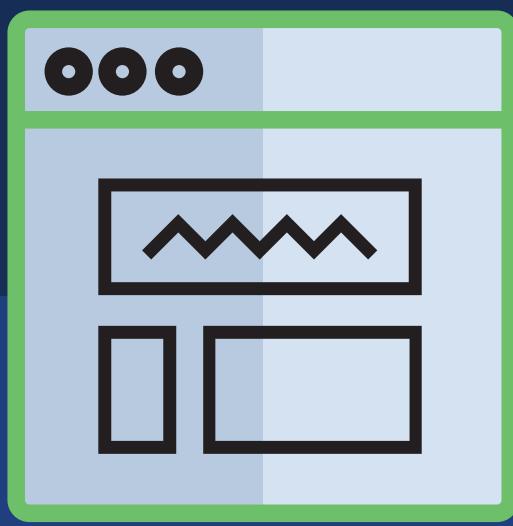
Re-engineering the architecture requires a team effort of 6-12 months (18 months in Zalando's case) - and therefore it should have a solid business foundation.



The most common reasons we've seen to initialize a transformation are the following:

- With four to five years of development, the scope of the system is so broad that implementing changes in one of the modules affects other areas and despite having unit-tests, making deep changes to the system logic is quite risky.
- Technical debt in one system area is accrued to a level at which it's extremely hard to resolve without major changes. Performance challenges exist in the product catalog, pricing/promo rules or central user database areas.
- There is a need to coordinate separate teams or vendors in a way which leads to minimal interference between them.
- The system is hard to test and deploy.
- There is a need to implement continuous deployments.





Best practices

Best practices

This eBook is intended to show you the most popular design patterns and practices related to microservices. I strongly recommend you to track the father of the micro services approach - Sam Newman. You should check out websites like: <http://microservices.io>, <https://dzone.com/> and <https://github.com/mfornos/awesome-microservices> (under the “microservices” keyword). They provide a condensed dose of knowledge about core microservice patterns, decomposition methods, deployment patterns, communication styles, data management and much more...

Create a Separate Database for Each Service

Sharing the same data structures between services can be difficult - particularly in environments where separate teams manage each microservice. Conflicts and surprising changes are not what you’re aiming for with a distributed approach.

Breaking apart the data can make information management more complicated the individual storage systems can easily de-sync or become inconsistent. You need to add a tool that performs master data management. While operating in the background, it must eventually find and fix inconsistencies. One of the patterns for such synchronization is **Event Sourcing**. This pattern can help you with such situations by providing you with a reliable history log of all data changes that can be rolled back and forth. **Eventual Consistency** and **CAP theorem** are fundamentals that must be considered during the design phase.



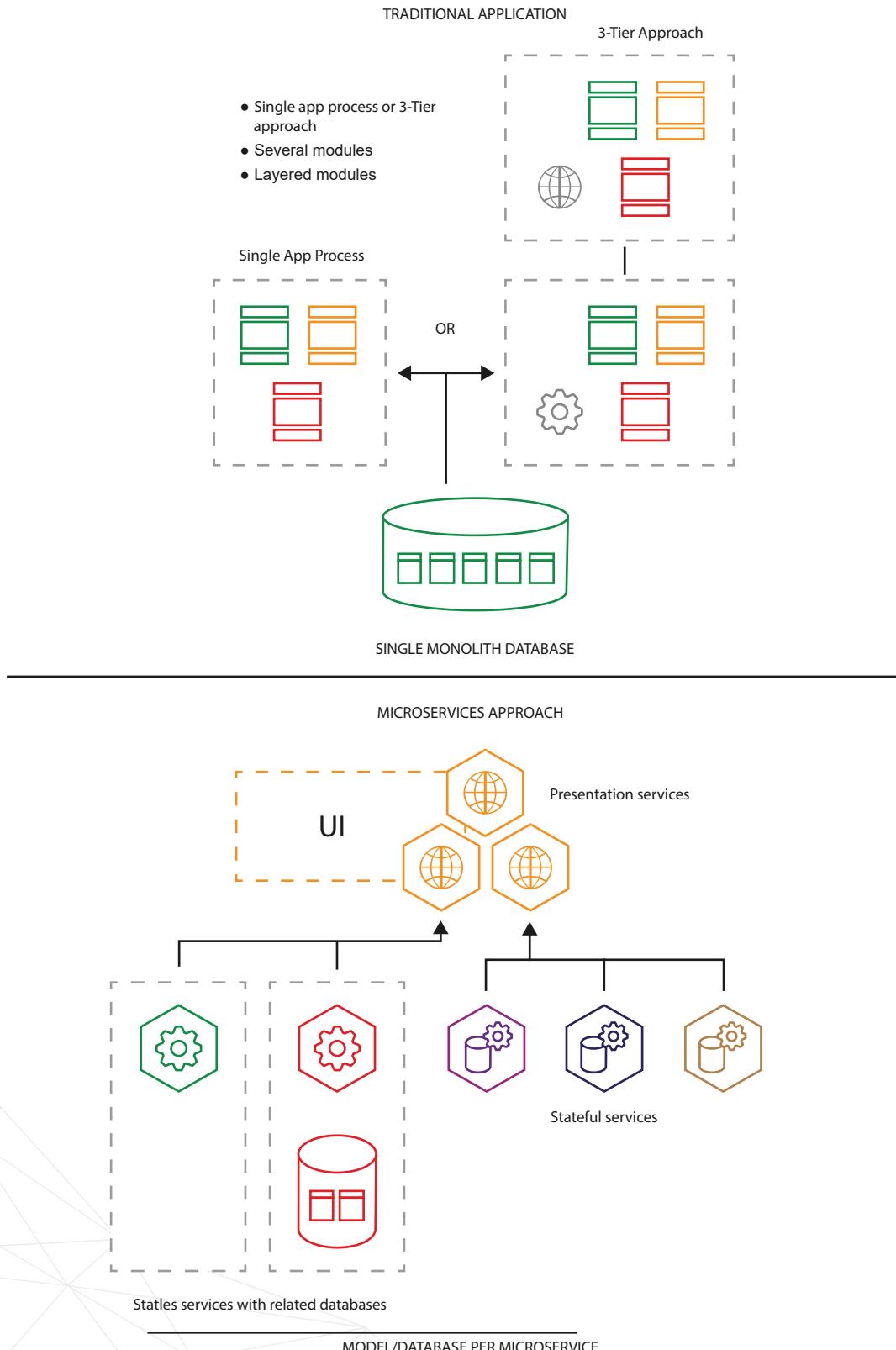


Fig. 4: Each microservice should have a separate database and be as self-sufficient as it can. From a design point of view - it's the simplest way to avoid conflicts. Remember - different teams are working on different parts of the application. Having a common database is like having a single point of failure with all conflicting changes deployed simultaneously between services.



| Rely on Contracts Between Services

Keep all code at a similar level of maturity and stability. When you have to modify the behaviour of a currently deployed (and stable) microservice, it's usually better to put the new logic into a new, separate service. It's sometimes called "immutable architecture".

Another point here is that you should maintain similar, specific requirements for all microservices like data formats, enumerating return values and describing error handling.

Microservices should comply with SRP (Single Responsibility Principle) and LSP (Liskov Substitution Principle).

| Deploy in Containers

Deploying microservices in containers is important because it means you need just one tool to deploy everything. As long as the microservice is in a container, the tool knows how to deploy it. It doesn't matter what the container is. That said, Docker seems to have become the de facto standard for containers very quickly.



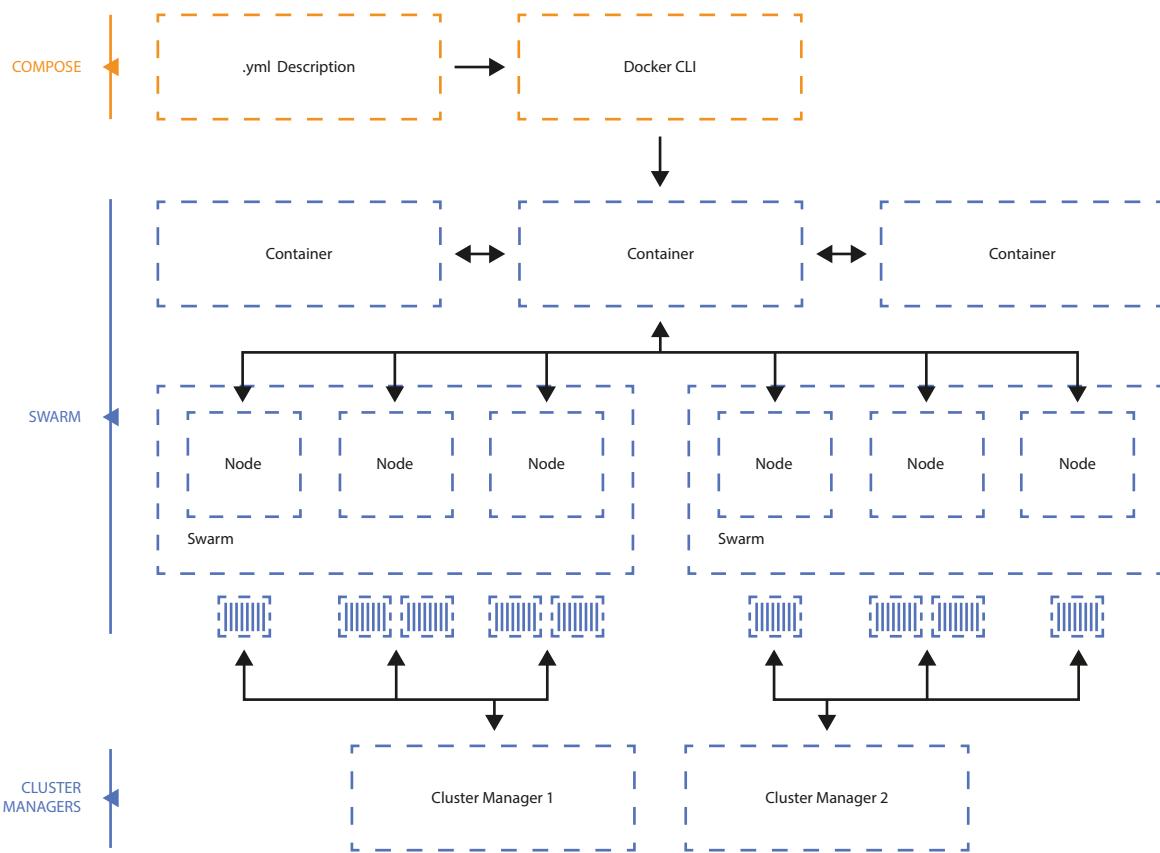


Fig. 5: Source - Docker Blog. Docker Swarm manages the whole server cluster - automatically deploying new machines with additional instances for scalability and high availability. Of course it can be deployed on popular cloud environments like Amazon.

Treat Servers as Volatile

Treat servers, particularly those that run customer-facing code, as interchangeable members of a group. It's the only way to successfully use the cloud's "auto scaling" feature.

They all perform the same function, so you don't need to be concerned with them individually. The role configuration across servers must be aligned and the deployment process should be fully automated.



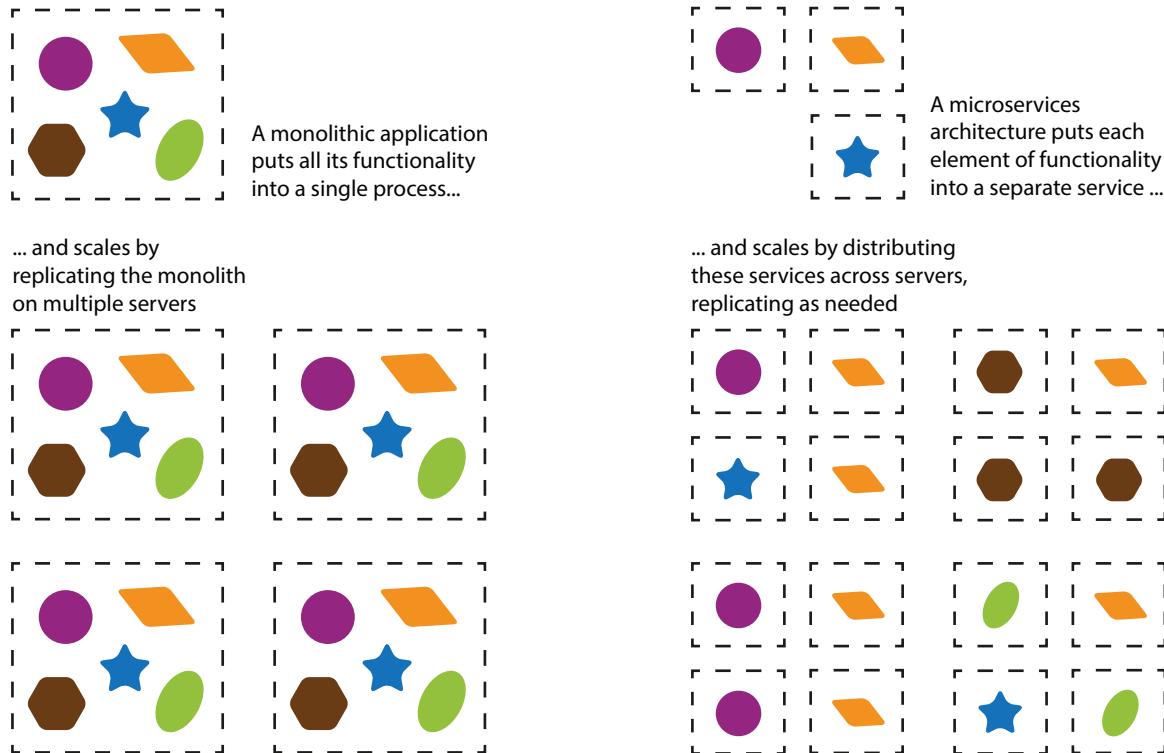
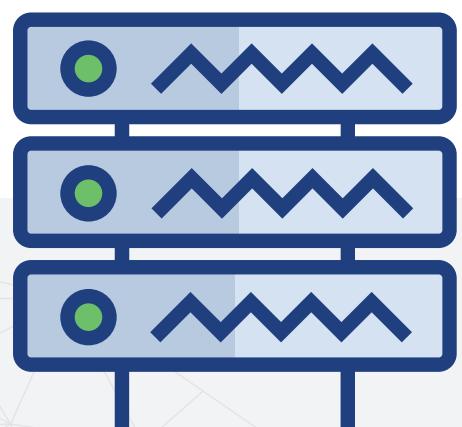
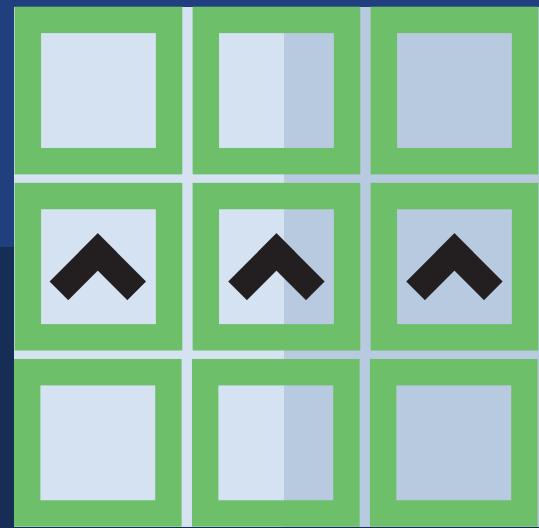


Fig. 6: Original idea - Martin Fowler
<https://martinfowler.com/articles/microservices.html>. Scaling microservices can be efficient because you can add resources directly where needed. You don't have to deal with storage replication, sticky sessions and all that kind of stuff because services are stateless and loosely-coupled by design.





Related techniques and patterns

| Related Techniques and Patterns

This eBook is intended to give you a quick-start, practical overview of the microservices approach. I believe, once interested in the topic, you can find additional sources to dig into. In this chapter I would like to mention just a few programming techniques and design patterns which have become popular with microservices gaining the spotlight. We want to cover the full scope of building microservices and tools that can be particularly useful to that goal.

| CAP theorem

Also called “Brewer theorem” after Eric Brewer, states that, for distributed systems it’s not possible to provide more than two of the following three guarantees:

- **Consistency** - every read receives the most recent data or error.
- **Availability** - every request receives a (non-error) response BUT without a guarantee of most-recent data.
- **Partition tolerance** - interpreted as a system able to work despite the number of dropped messages between cluster nodes.

In other words - when it comes to communication issues (partition of the cluster), you must choose between **consistency** or **availability**. This is strongly connected with techniques of high availability like caching and data redundancy (eg. database replication).



When the system is running normally - both availability and consistency can be provided. In case of failure, you get two choices:

- Raise an error (and break the availability promise) because it's not guaranteed that all data replicas are updated.
- Provide the user with cached data (due to the very same reason as above).

Traditional database systems (compliant with ACID⁶) prefer consistency over availability.

| Eventual consistency

When the system is running normally - both availability and consistency can be provided. In case of failure, you get two choices:

It's not a programming technique but rather something you have to think about when designing distributed systems. This consistency model is connected directly to the CAP theorem and informally guarantees that if **no new updates are made to a given data item, eventually all access to that item will return the last updated value.**

Eventually consistent services are often classified as providing BASE (**B**asically **A**vailable, **S**oft state, **E**ventual consistency) semantics, in contrast to traditional ACID guarantees.

⁶ <https://en.wikipedia.org/wiki/ACID>

To achieve eventual consistency, the distributed system must resolve data conflicts between multiple copies of replicated data. This usually consists of two parts:

- Exchanging updates between servers in a cluster
- Choosing the final state.

The widespread model for choosing the final state is “last writer wins” - achieved by including an update timestamp along with an updated copy of data.

| Design patterns

Having knowledge of the core theories that underpin the issues which we may encounter when developing and designing a distributed architecture, we can now go into higher-level concepts and patterns. Design patterns are techniques that allow us to compose code of our microservices in a more structured way and facilitate further maintenance and development of our platform.

| CQRS

CQRS means Command-Query Responsibility Segregation. The core idea behind CQRS is the extension of the CQS concept by Bertrand Meyer, where objects have two types of methods. Command methods perform actions in systems and always return nothing, query methods return values and they have no effect on the system.



In CQRS, write requests (aka commands) and read requests (aka queries) are separated into different models. The write model will accept commands and perform actions on the data, the read model will accept queries and return data to the application UI. The read model should be updated if, and only if, the write model was changed. Moreover, single changes in the write model may cause updates in more than one read model. What is very interesting is that there is a possibility to split data storage layers, set up a dedicated data store for writes and reads, and modify and scale them independently.

For example, all write requests in the eCommerce application, like adding a new order or product reviews, can be stored in a typical SQL database but some read requests, like finding similar products, can be delegated by the read model to a graph engine.

General flow in CQRS application:

- Application creates a command as a result of user action.
- Command is processed, write model saves changes in data store.
- Read model is updated based on changes in write model.

Pros:

- Better scalability and performance.
- Simple queries and commands.
- Possibility to use different data storage and theirs functionalities.
- Works well in complex domains.



Cons:

- Increased complexity of the entire system.
- Eventually consistent, read model may be out of sync with write model for a while.
- Possible data and code duplication.

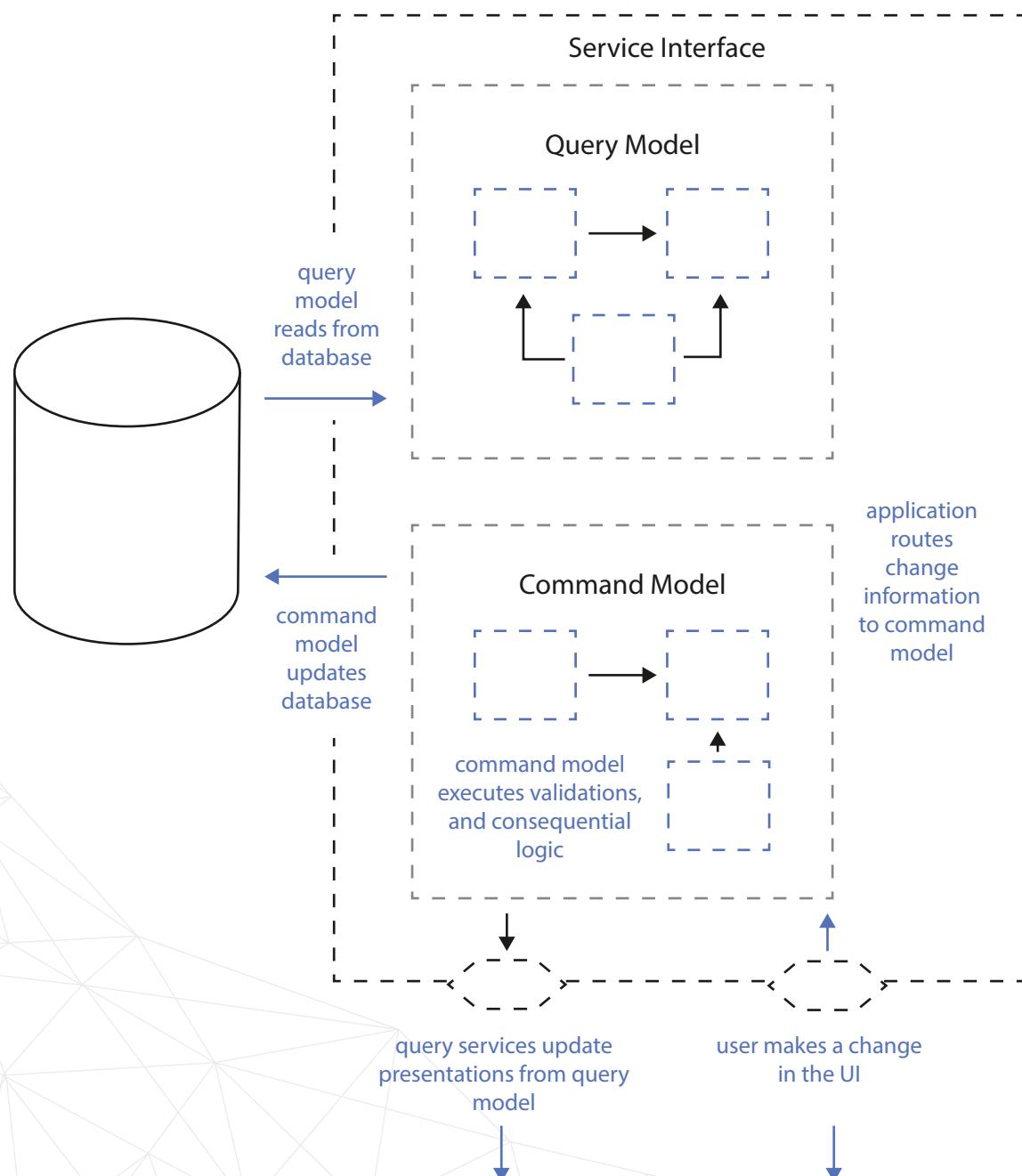




Fig. 9: CQRS architecture (<https://martinfowler.com/bliki/images/cqrs/cqrs.png>).

I Event Sourcing

Data stores are often designed to directly keep the actual state of the system without storing the history of all the submitted changes. In some situations this can cause problems. For example, if there is a need to prepare a new read model for some specific point of time (like your current address on an invoice from 3 months ago - which may have changed in the meantime - and you haven't stored the time-stamped data snapshots, it will be a big deal to reprint or modify the correct document).

Event Sourcing stores all changes as a time-ordered sequence of events; each event is an object that represents a domain action from the past. All events published by the application object persist inside a dedicated, append-only data store called Event Store. This is not just an audit-log for the whole system because the main role of Event Store is to reconstruct application objects based on the history of the related events.



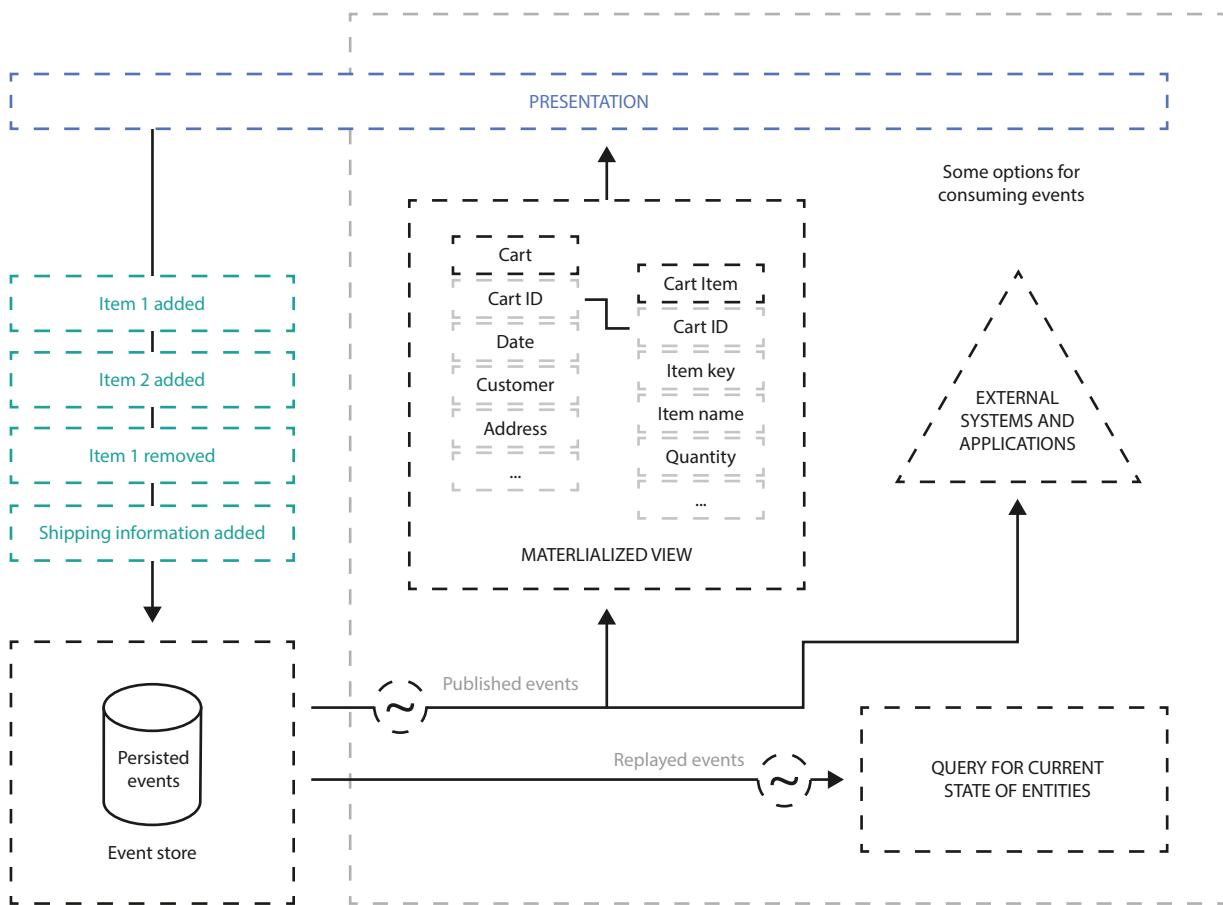


Fig. 10: Event Sourcing overview
https://docs.microsoft.com/en-us/azure/architecture/patterns/_images/event-sourcing-overview.png.

Consider the following sequence of domain events, regarding each Order lifecycle:

- OrderCreated
- OrderApproved
- OrderPaid
- OrderPrepared
- OrderShipped
- OrderDelivered



During the recreation phase, all events are fetched from the EventStore and applied to a newly constructed entity. Each applied event changes the internal state of the entity.

The benefits of this approach are obvious. Each event represents an action, which is even better if DDD is used in the project. There is a trace of every single change in domain entities.

But there are also some potential drawbacks here... How can we get the current states of tens of objects? How fast will object recreation be if the events list contains thousands of items?

Fortunately, the Event Sourcing technique has prepared solutions to these problems. Based on the events, the application can update one or more from materialized views, so there is no need to fetch all objects from the event history to get their current states.

If the event history of the entity is long, the application may also create some snapshots. By “snapshot”, I mean the state of the entity after every n-th event. The recreation phase will be much faster because there is no need to fetch all the changes from the Event Store, just the latest snapshot and further events.



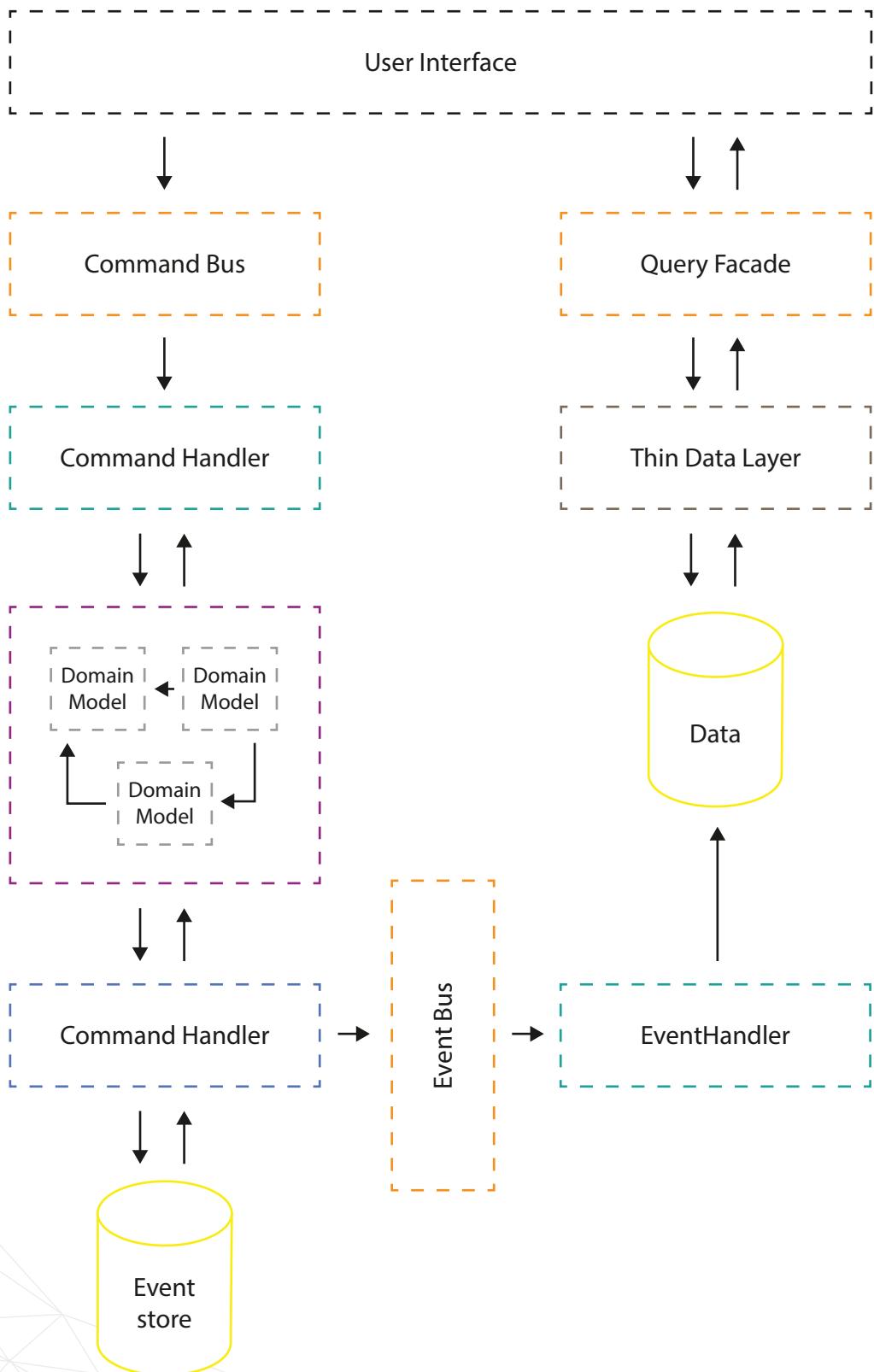


Fig. 11: Event Sourcing with CQRS
<https://pablocastilla.files.wordpress.com/2014/09/cqrs.png?w=640>.



Event Sourcing works very well with CQRS and Event Storming, a technique for domain event identification by Alberto Brandolini. Events found with domain experts will be published by entities inside the write model. They will be transferred to a synchronous or asynchronous event bus and processed by event handlers. In this scenario, event handlers will be responsible for updating one or more read models.

Pros:

- Perfect for modeling complex domains.
- Possibility to replay all stored events and build new read models.
- Reliable audit-log for free.

Cons:

- Queries implemented with CQRS.
- Eventually consistent model.

I Event driven data management

Microservices should be coupled as loosely as possible, It should be possible to develop, test, deploy and scale them independently. Sometimes an application should even be able to work without particular services (to comply with HA - high availability)... To achieve these requirements, each microservice should have a separate data store. Sounds easy - but what about the data itself? How to spread the information changes between services? What about consistency within the data?



One of the best solutions is simply using events. If anything important happened inside a microservice, a specific event is published to the message broker. Other microservices may connect to the message broker, receive, and consume a dedicated copy of that message. Consumers may also decide which part of the data should be duplicated to their local store.

Safe publishing of events from the microservice is quite complicated. Events must be published to the message broker if, and only if, data stored in a data store has changed. Other scenarios may lead to huge consistency problems. Usually it means that data and events should persist inside the same transaction to a single data store and then propagate to the rest of the system.

Switching from theory to a practical point of view, it's quite a common case to use RabbitMQ as a message broker. RabbitMQ is a very fast and efficient queue server written in Erlang with wide set of client libraries for the most popular programming languages. A popular alternative to RabbitMQ is Apache Kafka, especially for bigger setups or when event stream mining and analytics is critical.

Spreading data across multiple separated data stores and achieving consistency using events can cause some problems. For example, there is no easy way to execute a distributed transaction on different databases. Moreover, there can also be consistency issues because when events are inside the message broker, somewhere between microservices, the state of the whole system is inconsistent. The data store behind the original microservice is updated but changes aren't applied on data stores behind other microservices. This model, called Eventually Consistent,



is a Data will be synchronized in the future but you can also stop some services and you will never lose your data. They will be processed when services are restored.

In some situations, when a new microservice is introduced, there is a need to seed the database. If there is a chance to use data directly from different „sources of truth”, it’s probably the best way to setup a new service. But other microservices may also expose feeds of theirs events, for example in the form of ATOM feeds. New microservices may process them in chronological order, to compile the final state of new data stores. Of course, in this scenario each microservice should keep a history of all events, which can sometimes be a subsequent challenge.

Integration techniques

System integration is key to developing efficient microservices architecture. Services must talk to each other in a consistent way. The overall structure of a platform could be easily discoverable by hiding all of the dependencies behind facades like a common API gateway.

Moreover, all of that communication should use authentication mechanisms as microservices are commonly exposed to the outside world. They should not be designed with the intention of residing only in our firewall-protected network. We show two possible ways of making our integration secure by using token based techniques such as OAuth2 and JWT.



API Gateways

With the microservices approach, it's quite easy to make internal network communication very talkative. Nowadays, when 10G network connections are standard in data-centers, there may be nothing wrong with that. But when it comes to communication between your mobile app and backend services, you might want to compress as much information as possible into one request.

The second reason to criticise microservices might be a challenge with additional sub-service calls like authorization, filtering etc.

To overcome the mentioned obstacles, we can use the API Gateway approach. It means you can compile several microservices using one facade. It combines multiple responses from internal sub-services into a single response.

With almost no business logic included, gateways are an easy and safe choice to optimize communication between frontend and backend or between different backend systems.



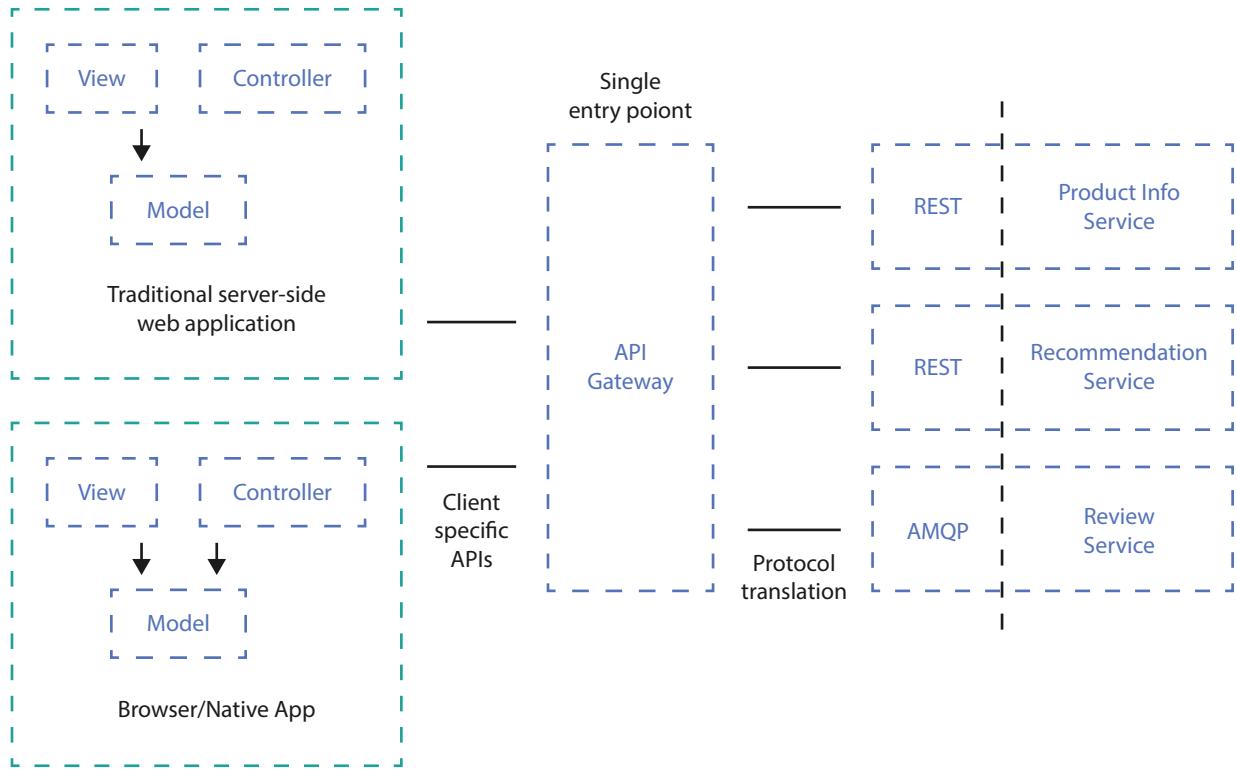


Fig. 12: Using an API gateway you can compose your sub-service calls into easy to understand and easy to use facades. Traffic optimization, caching and authorization are additional benefits of such an approach

The API Gateway - which is an implementation of classic Proxy patterns - can provide a caching mechanism as well (even using a vanilla-Varnish cache layer without additional development effort). With this feature alone, using cloud approaches (like Amazon solutions), can scale API and services very easily.

Additionally, you can provide common authorization layers for all services behind the gateway. For example - that's how Amazon API Gateway Service⁷ + Amazon Cognito⁸ work.

⁷ <https://aws.amazon.com/api-gateway/>

⁸ <http://docs.aws.amazon.com/cognito/latest/developerguide/authentication-flow.html>

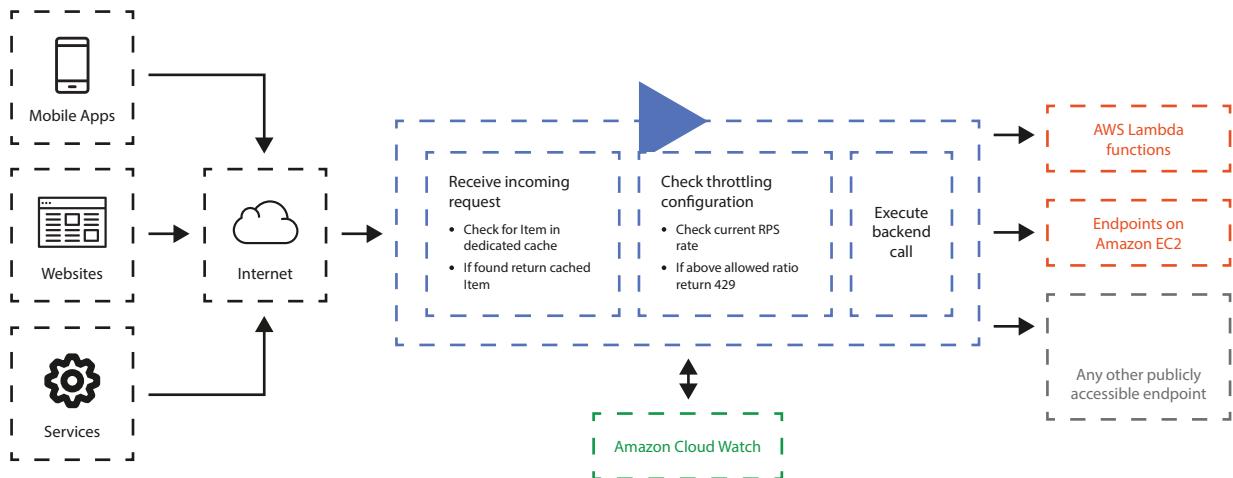


Fig. 13: Amazon API Gateway request workflow

<https://aws.amazon.com/api-gateway/details/>). Amazon gateway supports caching and authorization features in spite of your web-service internals.

Swagger⁹ can help you, once a Gateway has been built, with direct integration and support to Amazon services.

Backend for Frontends

A typical example of an API Gateway is the backend for frontends (BFF) pattern. It is about facades and compiling several microservices into optimized / device or channel-oriented API services. Its microservice design pattern was proposed by Sam Newman of Thought Works (author of "Building Microservices"): to create single purpose edge APIs for frontends and other parties.

Creating such a facade-API brings at least two benefits to your application:

- If you manage to have a few micro services behind your facade, you can avoid network latency - which is especially important on mobile devices.

⁹ <http://docs.aws.amazon.com/cognito/latest/developerguide/authentication-flow.html>

Using a facade, you can hide all network traffic between services executing the sub-calls in internal networks from the end-client.

- Then you can optimize your calls to be more compliant with a specific domain model. You can model the API structures by merging and distributing subsequent service calls instead of pushing this logic to the API client's code.

The diagram below shows a migration from General Purpose API to a dedicated backends for frontends approach which integrates the sub-services into logic.

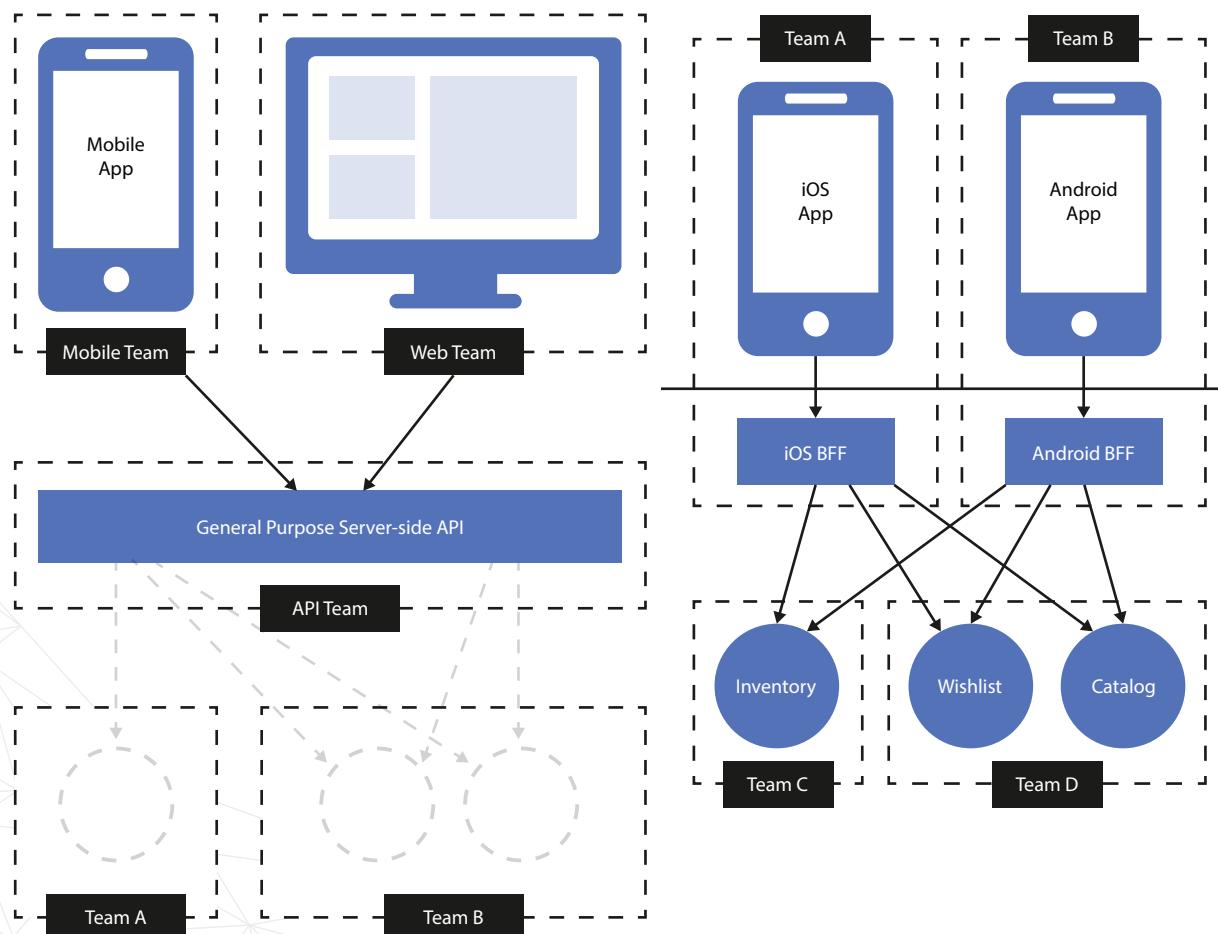


Fig. 14: Backend for frontends architecture is about minimizing the number of backend calls and optimizing the interfaces to a supported device.



There are many approaches to separate backend for frontends and roughly speaking it always depends on the differences in data required by a specific frontend, or usage-patterns behind specific API clients. One can imagine a separate API for frontend, mobile apps - as well as separate interfaces for iOS and Android if there are any differences between these applications regarding how service calls are made or their respective data formats.

One of the concerns of having a single BFF per user interface is that you can end up with lots of code duplication between the BFFs themselves.

Pete Hodgson (ex. Thought Works) suggests that BFFs work best when organized around teams. The team structure should drive how many BFFs you have. This is a pragmatic approach to not over-engineer your system but rather have one mobile API if you have one mobile team etc.

It's then a common pattern to separate shared algorithms, models and code to separate the shared service or library used by frontend-related facades. Creating such duplications can be avoided.

Let me quote a conclusion on BFF presented by Sam Newman himself:

Backends For Frontends solve a pressing concern for mobile development when using microservices. In addition, they provide a compelling alternative to the general-purpose API backend, and many teams make use of them for purposes other than just mobile development. The simple act of limiting the number of consumers they support makes them much easier to work with and change, and helps teams developing customer-facing applications retain more autonomy¹⁰.

¹⁰ <http://samnewman.io/patterns/architectural/bff/>

I Token based authorization (oauth2, JWT)

Authorization is a key feature of any enterprise grade application. If you remember the beginnings of web 2.0 and Web API's back then, a typical authorization scenario was based on an API key or HTTP authorization. With ease of use came some strings attached. Basically these "static" (API key) and not strongly encrypted (basic auth.) methods were not secure enough.

Here, delegated authorization methods come into action. By delegated, we mean that authorization can be given by an external system / identity provider. One of the first methods of providing such authentication was the OpenID standard¹¹ developed around 2005. It could provide a One Login and Single Sign On for any user. Unfortunately, it wasn't widely accepted by identification providers like Google, Facebook or e-mail providers.

The OAuth standard works pretty similarly to OpenID. The authorization provider allows Application Developers to register their own applications with the required data-scope to be obtained in the name of the user. The user authorizes specific applications to use with their account.

Facebook or Google Account login screens are a well known part of oauth authorization.

¹¹ <http://openid.net/>

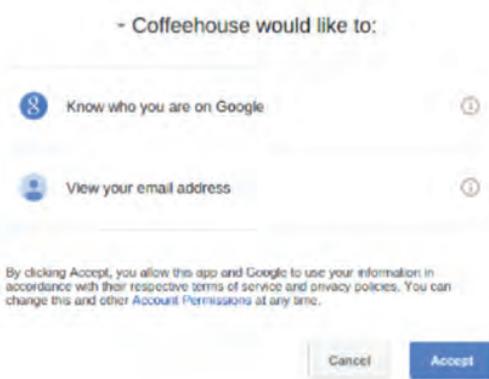


Fig. 15: Authorization screen for Google Accounts to authorize external application to use Google APIs in the name of the user.

After accepting the application request the authority party returns a temporary Access Token which should be used with API calls to verify the user identity. The Internal Authorization server checks tokens with its own database of issued tokens - paired with user identities, ACLs, etc.

Authorization tokens are issued for a specific amount of time and should be invalidated afterwards. Token authorization is 100% stateless; you don't have to use sessions (like with good, old session based authorization)¹². OAuth 2.0 requires SSL communication and avoids additional request-response signatures required by the previous version (requests were signed using HMAC algorithms); also, the workflow was simplified with 2.0 removing one additional HTTP request.

¹² <http://stackoverflow.com/questions/7561631/oauth-2-0-benefits-and-use-cases-why>

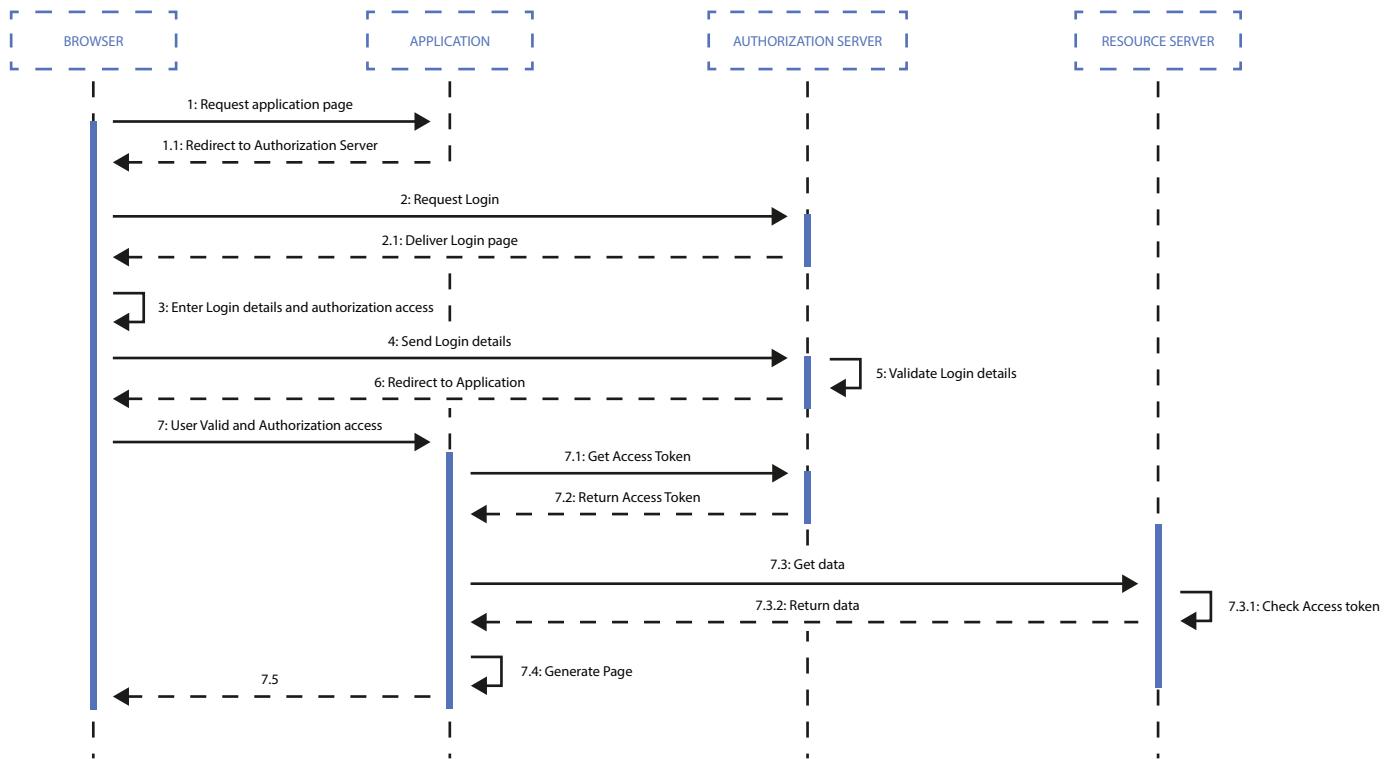


Fig. 16: Authorization flow for oauth2.

OAuth tokens don't push you to display the authentication dialog each time a user requires access to their data. Following this path would make it impossible to check e-mail in the background or do any batch processing operations. So how to deal with such background-operations? You should use "offline" tokens¹³ - which are given for longer time periods and can also be used to remember client credentials without requiring login/password each time the user hits your application.

There is usually no need to rewrite your own OAuth code as many open source libraries are available for most OAuth providers and frameworks. Just take a look on Github!

¹³ <https://auth0.com/docs/tokens/refresh-token>



There are SaaS solutions for identity and authorization, such as Amazon Cognito¹⁴ or Auth0¹⁵ that can be easily used to outsource the authorization of your API's.

I JSON Web Tokens (JWT)

Yet another approach to token based authorization is JWT¹⁶ (JSON Web Tokens). They can be used for stateless claim exchange between parties. As OAuth tokens require validation by the authenticating party between all requests - JSON Web Tokens are designed to self-contain all information required and can be used without touching the database or any other data source.

JWT are self-contained which means that tokens contain all the information. They are encoded and signed up using HMAC.

This allows you to fully rely on data APIs that are stateless and even make requests to downstream services. It doesn't matter which domains are serving your APIs, so Cross-Origin Resource Sharing (CORS) won't be an issue as it doesn't use cookies¹⁷.

¹⁴ <https://aws.amazon.com/cognito/>

¹⁵ <https://auth0.com/how-it-works>

¹⁶ <https://jwt.io/>

¹⁷ <https://jwt.io/introduction/>

Validation of HMAC tokens¹⁸ requires the knowledge of the secret key used to generate the token. Typically the receiving service (your API) will need to contact the authentication server as that server is where the secret is being kept¹⁹.

Please take a look at the example.

Example token:



```
eyJhbGciOiJIUzI1NilsInR5cCl6IkpxVCJ9.eyJzdWliOiIxM-
jM0NTY3ODkwIwibmFtZSI6Ikpvag4gRG9lIwiYWWRtaW4iOn
RydWV9.TJVA95OrM7E2cBab30RMHrHDcEfijoYZgeFONFh
7HgQ
```

Contains following informations: Please take a look at the example.

Header (algorithm and token type)	{ "alg": "HS256", "typ": "JWT" }
Payload (data)	{ "sub": "1234567890", "name": "John Doe", "admin": true }
Signature	HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload),) secret base64 encoded

¹⁸ https://en.wikipedia.org/wiki/Hash-based_message_authentication_code

¹⁹ <https://jwt.io/introduction/>



JWT tokens are usually passed by the HTTP Bearer header, then stored client side using localStorage or any other resource. Tokens can be invalidated at that time (exp claim included into token).

Once returned from authorization, service tokens can be passed to all API calls and validated server side. Because of the HMAC based signing process, tokens are safe.

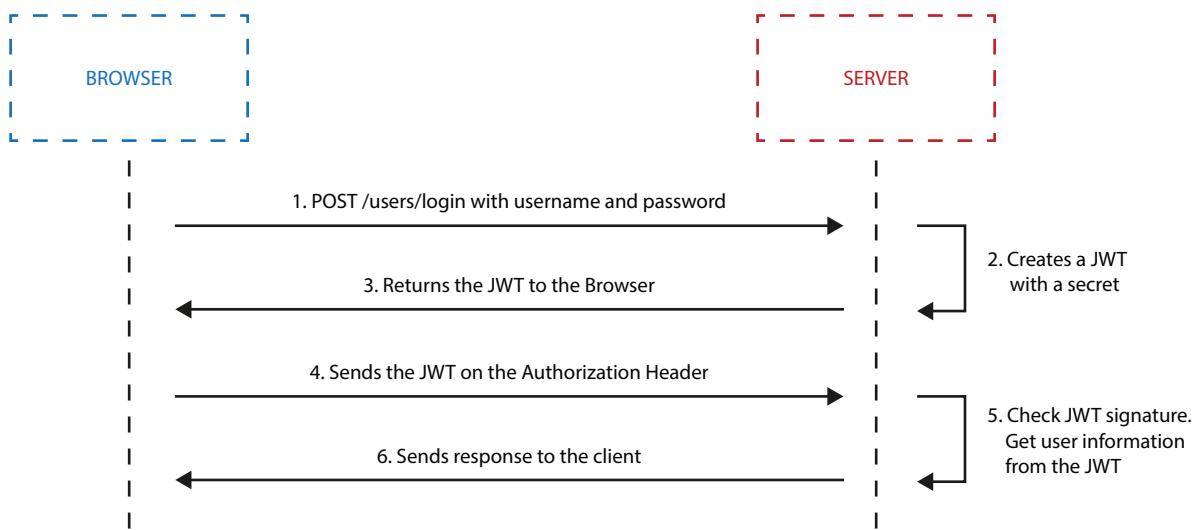


Fig. 17: JWT based authorization is pretty straight forward and it's safe. Tokens can be trusted by authorized parties because of the HMAC signature; therefore information contained by them can be used without checking ACL's and any further permissions.

Deployment of microservices

If done wrong, microservices may come with an overhead of operational tasks needed for the deployments and maintenance. When dividing a monolithic platform into smaller pieces, each of them should be easy to deploy in an automatic way.

¹⁸ https://en.wikipedia.org/wiki/Hash-based_message_authentication_code

¹⁹ <https://jwt.io/introduction/>



Nowadays, we see two main concepts that facilitates such a process - containerization and serverless architecture.

| Docker and containerization

If you are not familiar with containerization, then here are the most common benefits that make it worth digging deeper into this concept:

- Docker allows you to build an application once and then execute it in all your environments no matter what the differences between them.
- Docker helps you to solve dependency and incompatibility issues.
- Docker is like a virtual machine without the overhead.
- Docker environments can be fully automated.
- Docker is easy to deploy.
- Docker allows for separation of duties.
- Docker allows you to scale easily.
- Docker has a huge community.

Let's start with a quote from the Docker page:

Docker containers wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, system libraries – anything you can install on a server. This guarantees that it will always run the same, regardless of the environment it is running in.

This might sound familiar: virtualization allows you to achieve pretty much the same goals but in contrast to virtualization, Docker runs all processes directly on the host operating system. This helps to avoid the overhead of a virtual machine (both performance and maintenance).



Docker achieves this using the isolation features of the Linux kernel such as Cgroups and kernel namespaces. Each container has its own process space, filesystem and memory. You can run all kinds of Linux distributions inside a container. What makes Docker really useful is the community and all projects that complement the main functionality. There are multiple tools to automate common tasks, orchestrate and scale containerized systems. Docker is also heavily supported by many companies, just to name a couple: Amazon, Google, Microsoft. Currently, Docker also allows us to run Windows inside containers (only on Windows hosts).

| Docker basics

Before we dig into using Docker for the Microservices architecture let's browse the top-level details of how it works.

Image - holds the file system and parameters needed to run an application. It does not have any state and it does not change. You can understand an image as a template used to run containers.

Container - this is a running instance of an image. You can run multiple instances of the same image. It has a state and can change.

Image layer - each image is built out of layers. Images are usually built by running commands or adding/modifying files (using a Dockerfile). Each step that is run in order to build an Image is an image layer. Docker saves each layer, so when you run a build next time, it is able to reuse the layers that did not change. Layers are shared between all images so if two images start with similar steps, the layers are shared between them. You can see this illustrated below.





Fig. 18: You can use <https://imagelayers.io/> to analyze Docker image layers and compare them to each other. For example: ruby, python, node images share five layers - this means that if you download all three images the first 5 layers will be downloaded only once.

As you can see, all compared images share common layers. So if you download one of them, the shared layers will not be downloaded and stored again when downloading a different image. In fact, changes in a running container are also seen as an additional, uncommitted layer.

Registry - a place where images and image layers are kept. You can build an image on your CI server, push it to a registry and then use the image from all of your nodes without the need to build the images again.

Orchestration (docker-compose) - usually a system is built of several or more containers. This is because you should have only one concern per container. Orchestration allows you to run a multi-container application much easier and docker-compose is the most commonly used tool to achieve that. It has the ability to run multiple containers that can be connected with networks and share volumes.

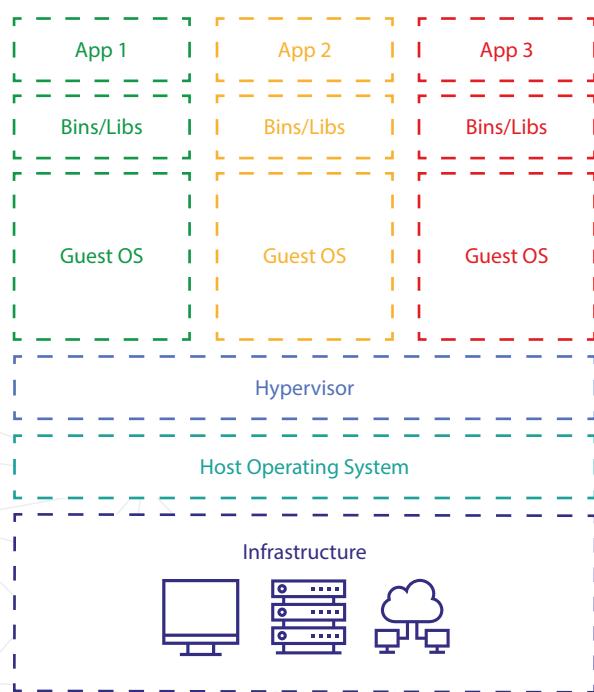


VM vs. Container

As mentioned earlier, Docker might seem similar to virtual machines but works in an entirely different way.

Virtual machines work exactly as the name suggests: by creating a virtualized machine that the guest system is using. The main part is a Hypervisor running on the host system and granting access to all kinds of resources for the guest systems. On top of the Hypervisor, there are Guest OS's running on each virtual machine. Your application is using this Guest OS.

What Docker does differently is directly using the host system (no need for Hypervisor and Guest OS), it runs the containers using several features of the Linux kernel that allow them to securely separate the processes inside them. Thanks to this, a process inside the container cannot influence processes outside of it. This approach makes Docker more lightweight both in terms of CPU/Memory usage, and disk space usage.



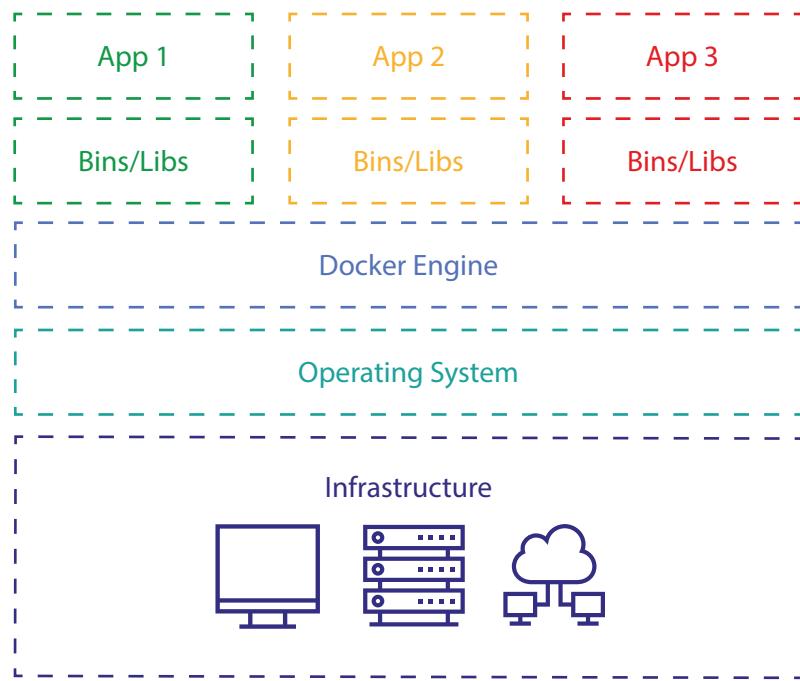


Fig. 19: Similar features, different architecture - Virtualization vs. Dockerization. Docker, leverages containerization - lightweight abstraction layer between application and the operating system / hardware. It separates the user processes but without running the whole operating system/kernel inside the container.

| From dev to production

Ok, so we have the technical introduction covered. Now let's see how Docker helps to build, run and maintain a Microservice oriented application.

| Development

Development is usually the first phase where Docker brings some extra value, and it is even more helpful with Microservice oriented applications. As mentioned earlier, Docker comes with tools that allow us to orchestrate a multi-container setup in a very easy way. Let's take a look at the benefits Docker brings during development.



Easy setup - low cost of introducing new developers

You only need to create a Docker configuration once and then each new developer on the team can start the project by executing a single command. No need to configure the environment, just download the project and run docker-compose up. That's all!

This might seem too good to be true but I have a good, real-life example of such a situation. I was responsible for a project where a new front-end developer was hired. The project was written in a very old PHP version (5.3) and had to be run on CentOS. The developer was using Windows and he previously worked on Java projects exclusively. I had a quick call with him and we went through a couple of simple steps: downloading and installing Docker, cloning the git repository and running docker-compose. After no more than 30 minutes he had a perfectly running environment and was ready to write his first lines of code!

No dependencies version mismatch issue

This issue often arises if a developer is involved in multiple projects, but it escalates in Micro-service oriented applications. Each service can be written by a different team and using different technologies. In some cases (it usually happens quite often) there might be a version mismatch within the same technology used in different services. A simple example: one service is using an older elastic version, and another a newer one. This can be dealt with accomplished by configuring two separate versions - but it is much easier to run them side-by-side in dedicated containers. A very simple example of such a configuration for docker-compose would look like this:





```
service_x_elastic:  
  image: elasticsearch:5.2.2  
service_y_elastic:  
  image: elasticsearch:2.4.4
```

Possibility to test if the application scales

Testing if the application scales is pretty easy with Docker. Of course, you won't be able to make some serious load testing on your local machine, but you can test if the application works correctly when a service is scaled horizontally. Horizontal scalability usually fails if the Microservice is not stateless and the state is not shared between instances. Scaling can be very easily achieved using docker-compose:

```
docker-compose scale service_x=4
```

After running this command there will be four containers running the same service_x. You can (and you should) also add a separate container with a load balancer like HAProxy in front of them. That's it. You are ready to test!

No more "works on my configuration" issues

Docker is a solution that allows one configuration to be run everywhere. You can have the same - or almost the same - version running on all developer machines, CI, staging, and production. This radically reduces the amount of "works on my configuration" situations. At least it reduces the ones caused by different setups.



Continuous Integration

Now that you have a working development setup, configuring a CI is really easy. You just need to setup your CI to run the same docker-compose up command and then run your tests, etc. No need to write any special configuration; just bring the containers up and run your tests. I've worked with different CI servers like Gitlab CI, Circle CI, Jenkins and the setup was always quick and easy. If some tests fail, it is easy to debug too. Just run the tests locally.

Pre-production

When you have your development setup up and running, it is also quite easy to push your application to a staging server. In most projects I know, this process was pretty straight-forward and required only a few changes. The main difference is in the so called volumes - files/directories that are shared between your local disk and the disk inside a container. When developing an application, you usually setup containers to share all project files with Docker so you do not need to rebuild the image after each change. On pre-production and production servers, project files should live inside the container/image and should not be mounted on your local disk.

The other common change applies to ports. When using Docker for development, you usually bind your local ports to ports inside the container, i.e. your local 8080 port to port 80 inside the container. This makes it impossible to test scalability of such containers and makes the URI look bad (no one likes ports inside the URI).



So when running on any production or pre-production servers you usually put a load balancer in front of the containers.

There are many tools that make running pre-production servers much easier. You should definitely check out projects like Docker Swarm, Kubernetes and Rancher. I really like Rancher as it is easy to setup and really easy to use. We use Rancher as our main staging management tool and all co-workers really enjoy working with it. Just to give you a small insight into how powerful such tools are: all our team members are able to update or create a new staging environment without any issues - and within a few minutes!

| Production

The production configuration should be exactly the same as pre-production. The only small difference might be the tool you use to manage the containers. There are a multitude of popular tools used to run production containers but my two favorites are Amazon EC2 Container Service and Google Cloud with Kubernetes on top. Both tools allow you to scale easily on new hosts.

One important thing you should keep in mind when going with Docker on production - monitoring and logging. Both should be centralized and easy to use.

| Cons

Docker has some downsides too. The first one you might notice is that it



takes some time to learn how to use Docker. The basics are pretty easy to learn, but it takes time to master some more complicated settings and concepts. The main disadvantage for me is that it runs very slowly on MacOS and Windows. Docker is built around many different concepts from the Linux kernel so it is not able to run directly on MacOS or Windows. It uses a Virtual Machine that runs Linux with Docker.

| Summary

Docker and the Microservice architecture approach work very well together and both concepts gain popularity each year. Over the past 4 years, we have been able to observe how Docker has gotten better and more mature with each release. At the same time, the whole ecosystem has grown and new tools have been published giving us more possibilities that we could not have thought of. By using Docker, we are able to easily run our Microservice oriented applications on our developer machines and then run the same setup on pre- and production servers. Right now we can configure a setup within minutes and then release our application to a server also within minutes. I'm really curious about what new possibilities we will get in the coming months.

| Serverless - Function as a Service

Serverless is not exclusively bound to microservice oriented applications but it is definitely good to know this concept, as it might be helpful in many cases.



Let me start with a couple of quotes that might be helpful for you to understand what serverless is about:

“

Serverless is a new cloud computing trend that changes the way you think about writing and maintaining applications.

— AUTH0.COM

“

Deploy your applications as independent functions, that respond to events, charge you only when they run, and scale automatically.

— SERVERLESS.COM

“

Serverless architectures refer to (...) custom code that's run in ephemeral containers.

— MARTINFOWLER.COM

As you can see, each of the quotes looks at serverless from a totally different perspective. This does not mean that some of the quotes are better, I think that all describe serverless in a very good way.

Serverless is considered to be a very bad name for what we are talking about. This is for two reasons:

- **Serverless as a concept** has a broader meaning than what it usually refers to; Serverless architecture can be used to describe both **Backend as a Service** and **Function as a Service**. Usually, and also in this article, we are interested in the latter: FaaS.



- **Serverless is a lie.** The truth is that servers are still there, Ops are also there. So why is this called „serverless“ - it's called so because you, as a business or as a developer, do not need to think about servers or ops. They are hidden behind an abstraction that makes them invisible to you. Both servers and ops are managed by a vendor like Amazon, Google, Microsoft, etc.

In the context of microservice architecture, FaaS is the concept that is interesting for us.

| Serverless providers

Currently, there are 4 major Clouds that allow us to use serverless architecture:

- **AWS Lambda** - named as the first adopter of FaaS, easily integrates with the rest of Amazon Web Services such as SNS or S3.
- **Google Cloud Functions** - still in beta, allows us to run our functions in Google Cloud. The drawback is, it currently only supports Node.js and JavaScript.
- **Azure Functions** - supports the widest range of languages (JavaScript, C#, F#, Python, PHP, Bash, Batch, and PowerShell) and easily allows us to integrate with Github for storing our code.
- **IBM Bluemix OpenWhisk** - it uses the open-source Apache OpenWhisk project running on top of the IBM Bluemix infrastructure.



The most notable feature is that you can use your Docker images to run as functions. A meaningful use-case of IBM OpenWhisk is a DarkVision Application²⁰, which shows how that technology can be used with techniques like Visual Recognition, Speech to Text and Natural Language Understanding.

Although it seems that we have a choice, we must keep in mind that commonly, such services are tightly coupled with other services of the particular Cloud, such as databases, message brokers or data storages. Mostly, the wiser choice is just to use the serverless functionality of the Cloud that we already use to run the rest of our microservices.

In the next sections, we'll use AWS Lambda for all of the examples, but the core concepts remain the same across all of the serverless providers.

■ FaaS

In an FaaS approach, developers are writing code - and code only. They do not need to care about the infrastructure, deployment, scalability, etc. The code they write represents a simple and small function of the application.

²⁰ <https://github.com/IBM-Bluemix/openwhisk-darkvisionapp>



It is run in response to a trigger and can use external services:

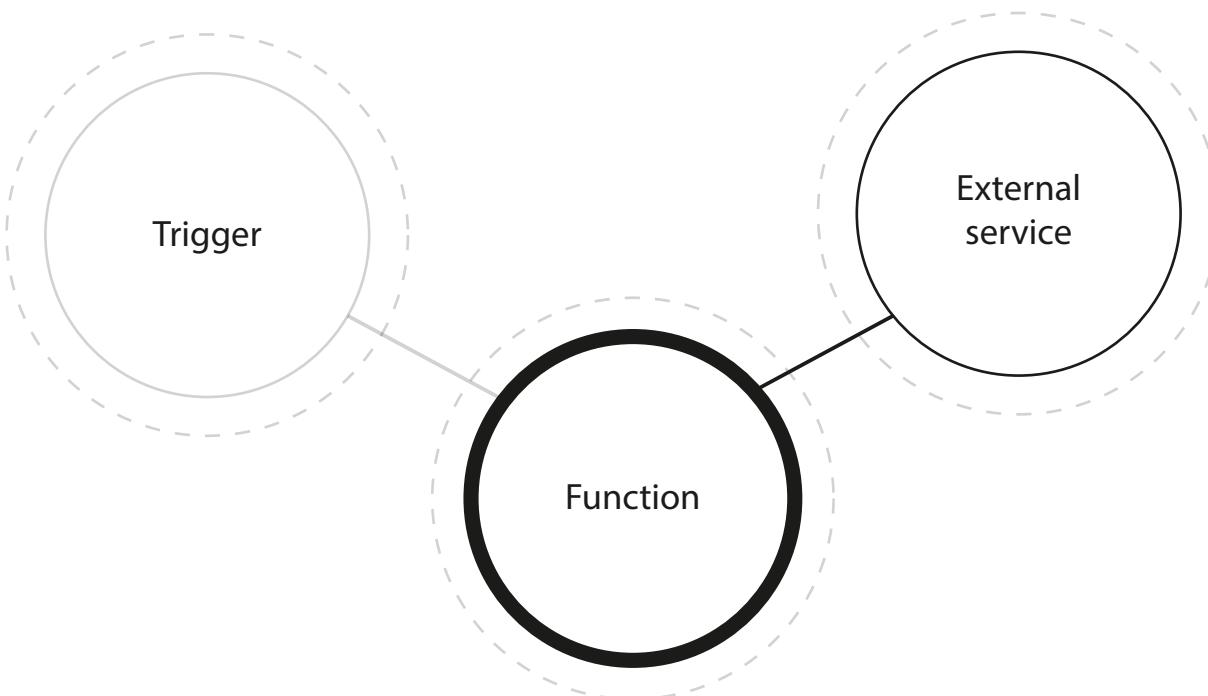


Fig. 20: Basic function as a service architecture consists of only two elements: the function to be run and a trigger to listen for. Usually the function is also connected to third-party services like a database.

A trigger can be almost anything. Based on AWS Lambda, the most popular FaaS service, the trigger might be:

- API call (any HTTP request).
- S3 bucket upload.
- New event in queue.
- Scheduled jobs.
- Other AWS Lambda functions.
- and many others, you can check it:

<http://docs.aws.amazon.com/lambda/latest/dg/invoking-lambda-function.html>.



Each function should comply with the following rules:

- It should not access the disk - AWS allows using a temporary /tmp directory that allows storing 512MB of data.
- It should be stateless and share-nothing. You can imagine it as a server powered up and configured to only handle one request (and then destroyed).
- Concise - your function should not take too long to run (usually seconds, but up to 300 seconds for AWS Lambda).

Once you have such a function, you just upload it to your service provider and provide some basic configuration. From that moment, on each action configured as a trigger, your function will be executed. The service provider tracks how long it takes for your function to execute, and multiplies the time by the amount of RAM configured (that's a limit you can change). You pay for GB-seconds of execution. This means that if your function is not executed, you do not pay anything and if your function is executed thousands of times during one day, you pay only for the GB-seconds your function took to run. There are no charges for scaling or idle time.

The cost of one GB-second on AWS Lambda is currently \$0.00001667 - this means that if your application requires 1024MB of RAM, and runs overall for 1,000,000 seconds (one million seconds), that is 277 hours (over 11 days), you will be charged \$16.77; There is also an additional price of \$0.20 per 1 million requests. It gets even better if you check out



the free tier that Amazon offers. Each month you get 3,200,000 seconds to run a function with a 128MB memory limit for free. That's over 890h - over 37 days!

I think the calculations above clearly show that you can gain a huge benefit by moving some parts (or all parts) of your application to a FaaS provider. You get the scalability and ops for free, as you do not need to take care of it.

Internally, functions are run in small, ephemeral and stateless containers that are spawn if your application needs to scale up.

You can find an interesting cost comparison to EC2 instances here: <https://www.trek10.com/blog/lambda-cost/>.

Architecture

I won't describe the architecture details of a serverless application in this article as it should be quite straightforward when writing a microservice application. The obvious and required step is to move as much presentation and logic to the customer as possible. Usually, your application front-end should be a mobile app or a single-page app.

On the back-end, you can start with a very simple architecture where the function is triggered by an API call and then connects to a DynamoDB instance (or any other on premise data source like MongoDB, MySQL) to fetch/modify some data. Then, you can apply direct read access to some data in your DynamoDB and allow clients to fetch the data directly,



but handle all data-modifying requests using your function. You can also introduce Event Sourcing very easily by having one function that records an event and other functions that take the event in order to refresh your read model.

You can also use FaaS to implement batch processing: split the stream of data into smaller chunks and then send them to another function that will run multiple instances of itself simultaneously. This allows you to process the data much faster. FaaS is often used to do real-time log processing.

It's easy!

Just a quick „hello world” example to show you how easily you can start writing serverless applications:



```
exports.handler = (event, context, callback) => {  
    callback(null, 'Hello World');  
};
```

Summary

Benefits

FaaS is easy to learn and implement, and it allows you to reduce the time to market. It also allows you to reduce costs, and to scale easily. Each function you write fits easily into a sprint, so it is easy to write serverless applications in agile teams.



Drawbacks

There might be a small vendor lock-in if you do not take this into consideration and do not introduce proper architecture. You should be aware of the communication overhead that is added by splitting the app into such small services. The most common issues mentioned are multitenancy (the same issue as with running containers on Amazon) and cold start - when scaling up, it takes some time to handle the first request by a new container. It might also be a bit harder to test such an application.

Good use-cases

Here are some use-cases that are interesting in my opinion:

- Mostly static pages, including eCommerce; You can host static content on a CDN server or add cache in front of your functions.
- Data stream analysis.
- Processing uploads - image/video thumbnails, etc.
- Actions users pay for. For example, adding watermarks to an ebook.
- Other cases when your application is not fully using the server capacity or you need to add scalability without investing much time and money.



Continuous Deployment

Just imagine that each of your microservices needs to be first built and then deployed manually, not even mentioning running unit tests or any kind of code-style tools. Having tens of those would be extremely time-consuming and would often be a major bottleneck in the whole development process.

Here comes the idea of Continuous Deployment - the thing that puts the workflow of your whole IT department together. In Continuous Deployment we can automate all things related with building Docker containers, running unit and functional tests and even testing performance of newly built services. At the end, if everything passes - nothing prevents us from automatically deploying working solutions into production.

The most commonly used software that handles the whole process is Jenkins, Travis CI, Bamboo or CircleCI. We'll show you how to do it using Jenkins.

Designing deployment pipeline

Going from the big picture, a common pipeline could look like this:

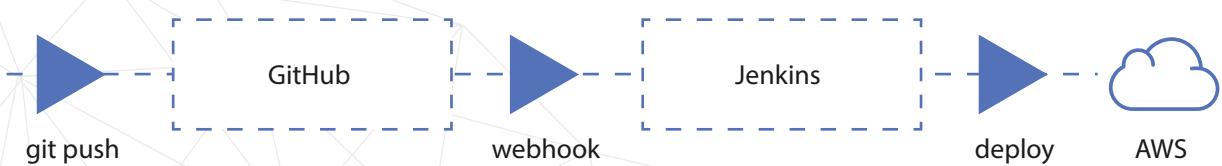


Fig. 21: Overview of our final Continuous Deployment pipeline.



Most of the hard work is done by that nice looking guy, called Jenkins. When someone pushes something to our Git repository (e.g. Github), the webhook triggers a job inside our Jenkins instance. It can consist of the following steps:

1. Build Docker image.
2. Run unit-tests inside the container.
3. Push image to our images repository (e.g. Docker Hub, Amazon ECR).
4. Deploy using Ansible or task schedulers like Amazon ECS.
 - a. Run functional tests (Selenium).
 - b. Run performance tests (JMeter).

After all this, we can set up a Slack notification that will inform us of success or failure of the whole process. The important thing is, that we should keep our Jenkins instance clean, so running all of the unit tests should be done inside a Docker container.

Coding our pipeline

Once we have the idea of our build process, we can code it using the Jenkinsfile. It's a file that describes our whole deployment pipeline. It consists of stages and build steps. Mostly, at the end of the pipeline we include post actions that should be fired when the build was successful or failed.

We should keep this file in our application's code repository - that way developers can also work with it, without asking DevOps for changes in the deployment procedure.

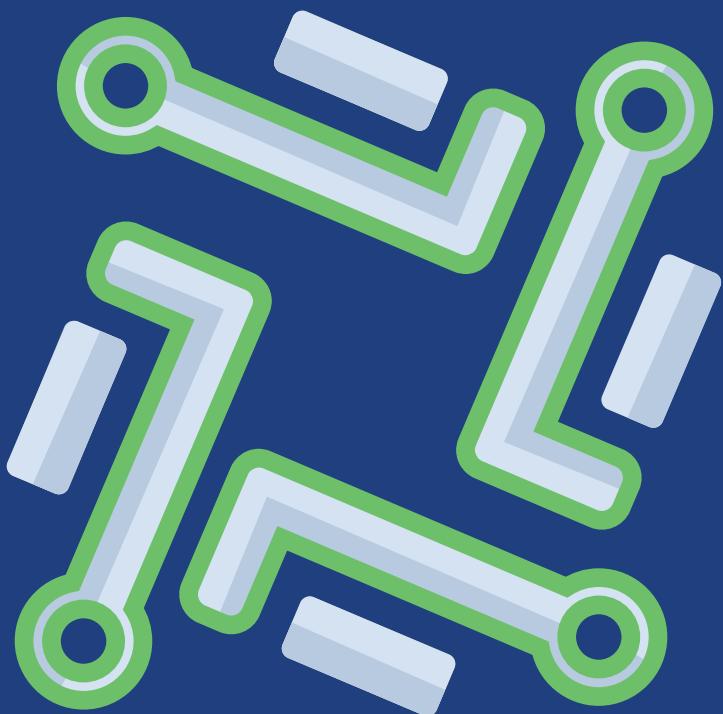


Here is a sample Jenkinsfile built on the basis of the previously mentioned steps. As we can see, the final step is to run another Jenkins job named deploy. Jobs can be tied together to be more reusable - that way we can deploy our application without having to run all of the previous steps.

```
#!/groovy
pipeline {
    agent any
    stages {
        stage('Build Docker') {
            steps {
                sh "docker build ..."
            }
        }
        stage('Push Docker Image') {
            steps {
                sh 'docker push ...'
            }
        }
        stage('Deploy') {
            steps {
                build job: 'deploy'
            }
        }
    }

    post {
        success {
            slackSend color: 'good', message: "Build Success"
        }
        failure {
            slackSend color: 'danger', message: "Build Failed"
        }
    }
}
```





Related technologies

Microservices based eCommerce platforms

There are major open-source platforms that were built using the Microservices approach by design. This section tries to list those that we think could be used as a reference for designing your architecture - or even better - could be used as a part of it.

Sylius

Sylius is the first Open Source eCommerce platform constructed from standalone components. What does it mean in practice? Every aspect of the shopping process is handled by individual PHP libraries. While the project itself provides a complete shop solution with a REST API, these decoupled components can be used separately to build Microservice applications.

Let's say we need to have two services for handling a Product Catalog and Promotions, respectively. The solution would be to take the two components and use them to develop two standalone applications. Before Sylius, you would have needed to write everything from scratch or strip the functionality from an existing eCommerce software.

On top of that, Sylius is based on the highly scalable Symfony framework, which integrates with a wide range of caching solutions, from Redis, Memcache to Varnish. It also provides tools for RAPID API development with JSON/XML support, which allows you to prototype your microservice in a much shorter timeframe and lower the costs of development.



Spryker

Spryker is a “Made in Germany” eCommerce platform created with a SOA approach with separated Backend (ZED) and Frontend (YVES) applications. The platform is designed with high throughput and scalability in mind. It’s not the classic microservices approach - you can learn more about Spryker’s founder’s view on that in Appendix 1 to this book.

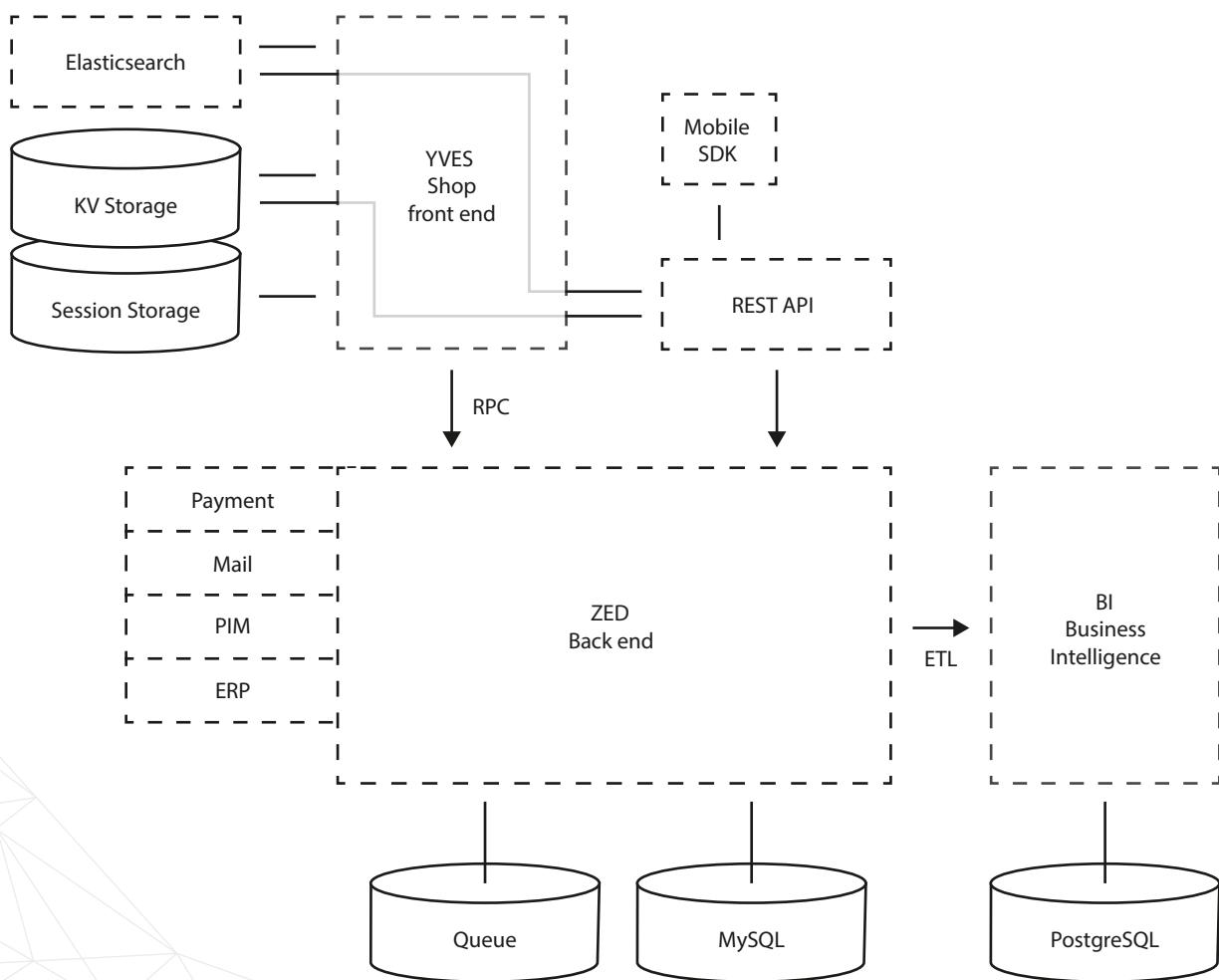


Fig. 14: Backend for frontends architecture is about minimizing the number of backend calls and optimizing the interfaces to a supported device.

The Spryker source code is available on Github:

<https://github.com/spryker>. The platform comes with an interesting licensing model - per developer seat (not related to revenues, servers etc...).

Open Loyalty

A loyalty/rewards program that can be easily integrated with eCommerce and/or POS. It's interesting because of the CDB module (Central Data Base) which is responsible for gathering a 360deg. view of each customer.

Open Loyalty leverages the CQRS and Event Sourcing design patterns. You can use it as a headless CRM leveraging a REST API (with JWT based authorization).

We've seen many cases of Open Loyalty being used as CRM and marketing automation.

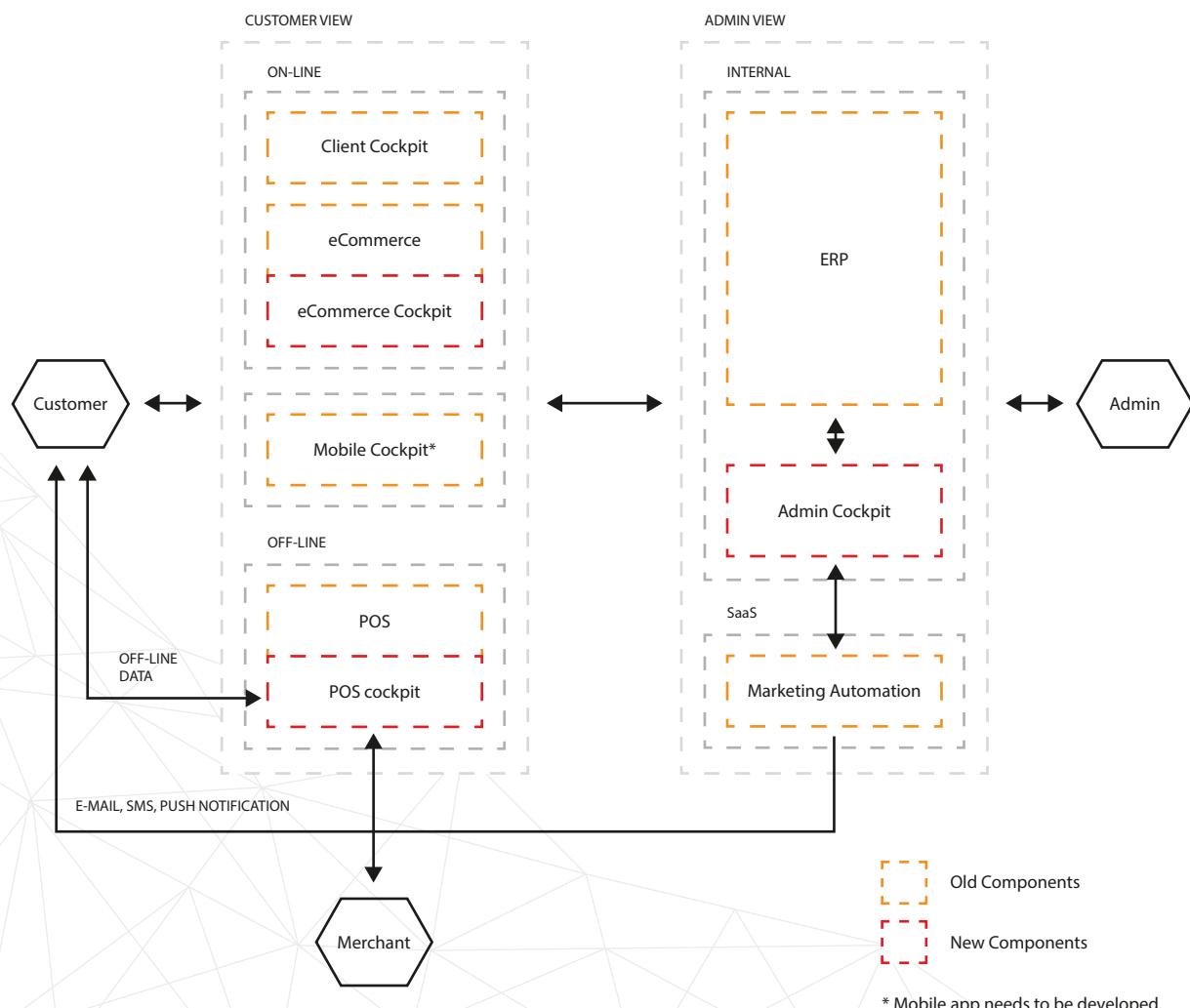


Fig. 23: Open Loyalty architecture - each application works as separate service.

The platform is open source and you can find the code on Github (<https://github.com/DivanteLtd/open-loyalty>).

More information: <http://openloyalty.io>.

I Technologies that empower the microservices architecture

Pimcore is an Enterprise Content platform for:

- CMS - content management.
- PIM - master data management for products.DAM - digital assets management for attachments, videos and pictures.
- eCommerce Framework - for building checkout features and managing orders.

The Pimcore REST API²² can be used to make Pimcore an eCommerce backend for Mobile applications or to extend existing eCommerce platform catalog capabilities, etc.

It's a open source technology developed in Austria with a really active community and version 5.0 (based on Symfony Framework) on the horizon.

More on Pimcore: <http://pimcore.org>.

²² https://www.pimcore.org/docs/latest/Web_Services/index.html



I Technologies that empower the microservices architecture

The microservices architecture introduces new concepts that sometimes also require new or different tools compared to the monolithic approach. Also, keeping in mind, that this approach may lead to more complexity of our platform, we should automate as many things as we can from the beginning.

We'll show you some of the most widely used tools and technologies that could empower your development by making things easier, more automated and are very suitable when diving into Microservices.

I Ansible

DevOps is an agile way to maintain software. It emphasizes communication between IT and SD²³.

Ansible is a tool for automation of DevOps routines. Ansible uses an agentless architecture which means that no additional software is needed to be installed on target machines; communication is done by issuing plain SSH commands. It automates applications deployment, configuration management, workflow orchestration and even cloud provisioning – all in one tool. Shipping with nearly 200 modules in the core distribution, Ansible provides a vast library of building blocks for managing all kinds of IT tasks.

²³ <https://pl.wikipedia.org/wiki/DevOps>



Ansible composes each server (or group of them, named inventory) from reusable roles. We can define ours, such as Nginx, PHP or Magento, and then reuse them for different machines. Roles are next tied together in "Playbooks" that describe the full deployment process.

There're plenty of well-written, already made Playbooks that you could adapt and reuse for configuring your infrastructure. As an example, when installing Magento you could use:

<https://github.com/aslaen/AnsiblePlaybooks/tree/master/ansible-magento-lemp>.

To configure our first servers with the Nginx web server and PHP, we should first create two roles that will be next used in a final Playbook.



1. Nginx:

```
# in ./roles/nginx/tasks/main.yml
- name: Ensures that nginx is installed
  apt: name=nginx state=present

- name: Creates nginx configuration from Jinja template file
  template:
    src: "/etc/nginx/nginx.conf.j2"
    dest: "/etc/nginx/nginx.conf"
```





2. PHP:

```
# in ./roles/php/tasks/main.yml
- name: Ensures that dotdeb APT repository is added
  apt_repository: repo="deb http://packages.dotdeb.org
jessie all" state=present

- name: Ensures that dotdeb key is present
  apt_key: url=https://www.dotdeb.org/dotdeb.gpg
state=present

- name: Ensures that APT cache is updated
  apt: update_cache=yes

- name: Ensures that listed packages are installed
  apt: pkg="{{ item }}"
  with_items:
    - php7.0-cli
    - php7.0-curl
    - php7.0-fpm
```

Having these roles, we can now define a playbook that will combine them to set-up our new server with nginx and php installed:



```
# in ./php-nodes.yml
- hosts: php-nodes
  roles:
    - nginx
    - php
```



The last thing we need to do is to tell Ansible the hostnames of our servers:

```
⟨ooo⟩ # in ./inventory  
[php-nodes]  
php-node1.acme.org  
php-node2.acme.org
```

Deployment is now as easy as typing a single shell command that will tell Ansible to run the `php-nodes.yml` playbook on hosts from the inventory file as root (-b):

```
⟨ooo⟩ $ ansible-playbook -i inventory php-nodes.yml -b
```

As we defined two hosts in a “`php-nodes`” group, Ansible is smart enough to run the Playbook concurrently for every server. That way we’re able to make a deployment on a bigger group of machines at once without wasting time doing it one-by-one.

| ReactJS

React is an open source user interface (UI) component library. It was developed at Facebook to facilitate creation of interactive web interfaces. It is often referred to as the V in the “MVC” architecture as it makes no assumptions about the rest of your technology stack.



With React, you compose your application out of components. It embraces what is called component-based architecture - a declarative approach to developing web interfaces where you describe your UI with a tree of components. React components are highly encapsulated, concern-specific, single-purpose blocks. For example, there could be components for address or zip code that together create a form. Such components have both a visual representation and dynamic logic.

Some components can even talk to the server on their own, e.g., a form that submits its values to the server and shows confirmation on success. Such interfaces are easier to reuse, refactor, test and maintain. They also tend to be faster than their imperative counterparts as React - being responsible for rendering your UI on screen - performs many optimisations and batches updates in one go.

It's most commonly used with Webpack - a module bundler for modern Javascript. One of its features - code-splitting - allows you to generate multiple Javascript bundles (entry points) allowing you to send clients only the part of Javascript that is required to render that particular screen.

One of the interesting movements in frontend-development nowadays is an Isomorphic approach. Which means that both frontend and backend are sharing the same code. In this particular case, frontend app can be created in React and backend code run by NodeJS.

| NodeJS

NodeJS is a popular (de facto industry standard) JavaScript engine that



can be used server-side and in CLI environments. There are plenty of JavaScript Web frameworks available, like Express (<https://expressjs.com/>) and HapiJS (<https://hapijs.com/>) - to name but two. As NodeJS is built around Google's V8 JavaScript engine (initially developed as Chrome/Chromium JS engine) it's blazingly fast. Node leverages the events-polling/non-blocking IO architecture to provide exceptional performance results and optimizes CPU utilization (for more, read about the c10k problem: <http://www.kegel.com/c10k.html>).

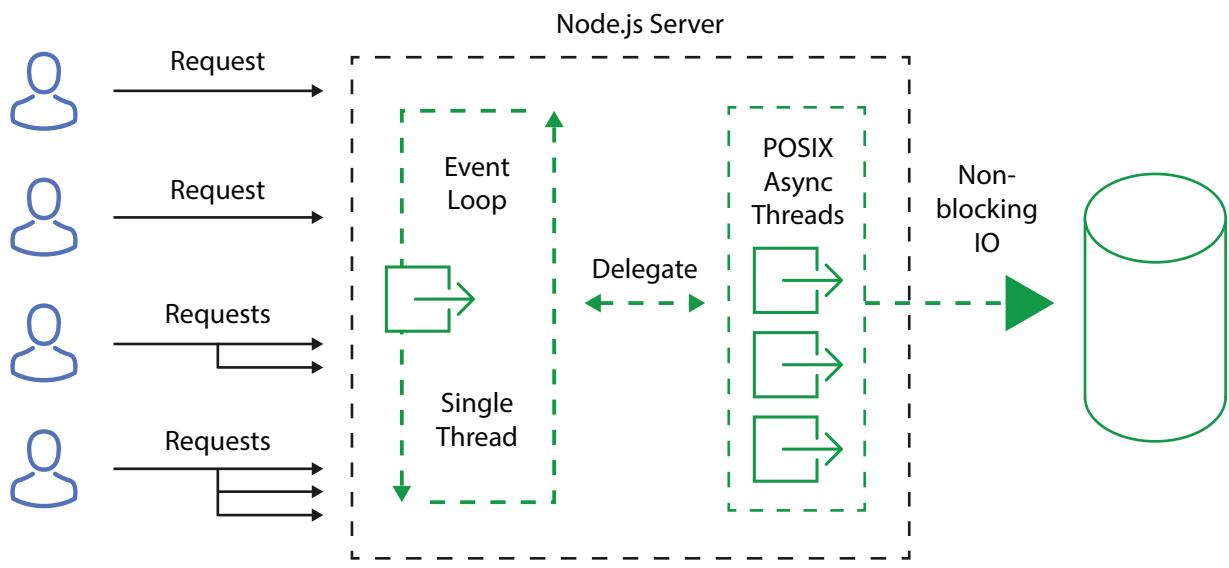


Fig. 24: Node.js request flow. Node leverages Event polling and maximizing the memory and CPU usage on running parallel operations inside single threaded environment.

interoperate with frontend JS code very easily. There is an emerging trend of building Universal apps - which more or less means that the same code base is in use on the frontend and backend. One of the most interesting and popular frameworks for building Isomorphic apps is React Js (<https://facebook.github.io/react/>).

NodeJS is used as a foundation for many CLI tools - starting from the most popular "npm" (Node Package Manager), followed by a number of tools like Gulp, Yeoman and others.



JavaScript is the rising star of programming languages. It can even be used for building desktop applications - like Visual Studio Code or Vivaldi web browser (!); these tools are coded in 100% pure JavaScript - but for the end users, nothing differs from standard desktop applications. And they're portable between operating systems by default!

On the server side, NodeJS is very often used as an API platform because of the platform speed. The event polling architecture is ideal for rapid but short-lived requests.

Using “npm” one can install almost all available libraries and tools for the JS stack - including frontend and backend packages. As most modern libraries (eg. GraphQL, Websockets) have Node bindings, and all modern cloud providers support this technology as well, it’s a good choice for backend technology backing microservices.

Famous NodeJS users

“

Node.js helps us solve this (boundary between the browser and server) by enabling both the browser and server applications to be written in JavaScript. It unifies our engineering specialties into one team which allows us to understand and react to our users' needs at any level in the technology stack.

—Jeff Harrel, Senior Director of Payments Products and Engineering at PayPal²⁴

²⁴ <https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/>



LinkedIn

“

One reason was scale. The second is, if you look at Node, the thing it's best at doing is talking to other services.

— Mobile Development Lead, Kiran Prasad²⁵

eBay

“

We had two primary requirements for the project. First, was to make the application as real time as possible-i.e., maintain live connections with the server. Second, was to orchestrate a huge number of eBay-specific services that display information on the page-i.e.

— Senthil Padmanabhan, Principal Web Engineer at eBay²⁶

Other projects that leverage NodeJS:

Uber

<https://nodejs.org/static/documents/casestudies/Nodejs-at-Uber.pdf>

Netflix

<http://thenewstack.io/netflix-uses-node-js-power-user-interface/>

Groupon

<http://www.datacenterknowledge.com/archives/2013/12/06/need-speed-groupon-migrated-node-js/>

²⁵ <http://venturebeat.com/2011/08/16/linkedin-node/>

²⁶ <http://www.ebaytechblog.com/2013/05/17/how-we-built-ebays-first-node-js-application/>



■ Swagger

This powerful tool is too commonly used only for generating nice-looking documentation for APIs. Basically, swagger is for defining the API interfaces using simple, domain-driven JSON language.

The editor is only one tool from the toolkit but other ones are:

- **Codegen** - for generating the source code scaffold for your API - available in many different languages (Node, Ruby, .NET, PHP).
- **UI** - most known swagger tool for generating useful and nice looking interactive documentation.

Everything starts with a specification file describing all the Entities and interfaces for the REST API. Please take a look at the example below:

```
<oo> {
  "get": {
    "description": "Returns pets based on ID",
    "summary": "Find pets by ID",
    "operationId": "getPetsById",
    "produces": [
      "application/json",
      "text/html"
    ],
    "responses": {
      "200": {
        "description": "pet response",
        "schema": {
          "type": "array",
          "items": [
            {
              "name": "pet",
              "in": "body",
              "schema": {
                "type": "object",
                "properties": {
                  "id": {
                    "type": "string"
                  },
                  "category": {
                    "type": "string"
                  },
                  "name": {
                    "type": "string"
                  },
                  "photoUrls": {
                    "type": "array",
                    "items": [
                      "string"
                    ]
                  },
                  "tags": {
                    "type": "array",
                    "items": [
                      {
                        "type": "string"
                      }
                    ]
                  },
                  "status": {
                    "type": "string"
                  }
                }
              }
            }
          ]
        }
      }
    }
  }
}
```



```

    "type": "array",
    "items": {
        "$ref": "#/definitions/Pet"
    }
},
},
},
],
"parameters": [
{
    "name": "id",
    "in": "path",
    "description": "ID of pet to use",
    "required": true,
    "type": "array",
    "items": {
        "type": "string"
    },
    "collectionFormat": "csv"
}
]
}

```

\$ref relates to other entities described in the file (data models, structures etc). You can use primitives as the examples and return values (bool, string...) as well as hash-sets, compound objects and lists. Swagger allows you to specify the validation rules and authorization schemes (basic auth, oauth, oauth2).



```
{  
  "oauth2": {  
    "type": "oauth2",  
    "scopes": [  
      {  
        "scope": "email",  
        "description": "Access to your email address"  
      },  
      {  
        "scope": "pets",  
        "description": "Access to your pets"  
      }  
    ],  
    "grantTypes": {  
      "implicit": {  
        "loginEndpoint": {  
          "url":  
            "http://petstore.swagger.wordnik.com/oauth/dialog"  
        },  
        "tokenName": "access_token"  
      },  
      "authorization_code": {  
        "tokenRequestEndpoint": {  
          "url":  
            "http://petstore.swagger.wordnik.com/oauth/requestToken",  
          "clientIdName": "client_id",  
          "clientSecretName": "client_secret"  
        },  
        "tokenEndpoint": {  
          "url":  
            "http://petstore.swagger.wordnik.com/oauth/token",  
          "tokenName": "access_code"  
        }  
      }  
    }  
  }  
}
```



Last but not least swagger the OpenAPI specification format has become more and more a standard and should be considered when starting new API projects. It's supported by many external tools and platforms - including Amazon API Gateway²⁷.

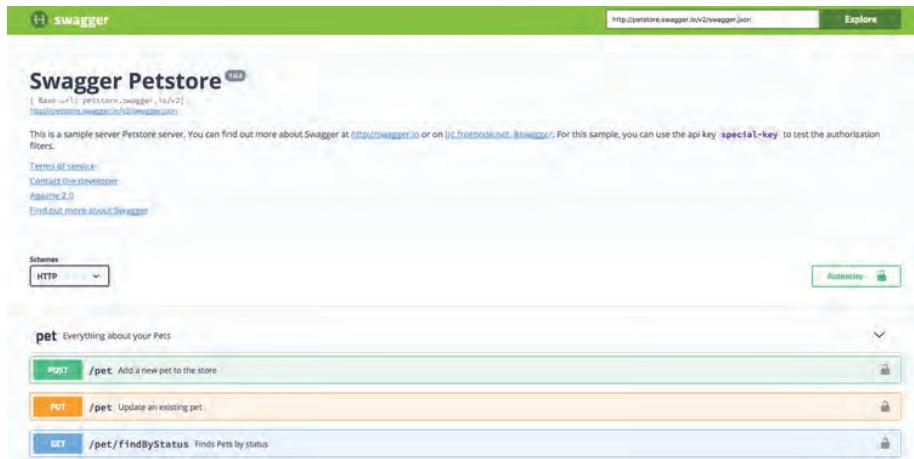


Fig. 25: Swagger UI generates a nice-looking specification for your API along with a "try-it-out" feature for executing API calls directly from the browser.

Elasticsearch

The simplest way to start with a microservices approach in eCommerce is often to delegate the search feature to an external tool like Elasticsearch/Solr or to SaaS solutions like Klevu.

Elasticsearch is a search engine available via REST API (updates, reads, searches...). It can be a micro service for catalog operations in eCommerce and it's often used to leverage the NoSQL scalability of its internal document database over standard SQL solutions.

²⁷ <https://m.signalvnoise.com/the-majestic-monolith-29166d022228#.90yg49e3j>

Elasticsearch supports full-text search with faceted filtering and support for most major languages with stemming and misspelling correction features.

There are plenty of modules to Magento and other platforms that synchronize product feeds to ES and then provide catalog browsing via ES web-services - without touching the relational database.

Elasticsearch is even used for log analysis with tools like Kibana and Logstash. With its ease of use, performance and scalability characteristics, it is actually best choice for most eCommerce and content related sites.

Elastic is well supported by cloud providers like Amazon and supports Docker.

GraphQl

Modeling a great REST API is hard - using and supporting changes in an API over time is sometimes even harder. GraphQL (<http://graphql.org>) is a query language; a proposition to a new way of thinking about APIs.

Widely used REST APIs are organized around HTTP endpoints. GraphQL APIs are different; they are built in terms of types and fields, and relations between them. It gives clients the ability to ask for what they need directly instead of many different REST requests. All the necessary data will be queried and returned with a single call.



Data definition:



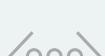
```
type Project {  
    name: String  
    tagline: String  
    contributors: [User]  
}
```

Sample query:



```
{  
  project(name: "GraphQL") {  
    tagline  
  }  
}
```

Query result:



```
{  
  "project": {  
    "tagline": "A query language for APIs"  
  }  
}
```

GraphQL was developed internally by Facebook in 2012 and open-sourced 3 years later with Relay, a JavaScript framework for building data-driven React applications. Nowadays, the GraphQL ecosystem is growing rapidly; both server and frontend libraries are available for many programming languages and developers have dedicated tools for GraphQL API design. Many other organizations, including Github, Pinterest and Shopify are adopting GraphQL because of its benefits.



| Distributed logging and monitoring

Distributed systems require new levels of application monitoring and logging. With monolithic applications you can track one log-file for events (usually) and use some Zabbix triggers to get a complete view of a server's state, application errors, etc.

| Graylog

With distributed services you have to track a whole bunch of new metrics:

- Network latency - which can ruin the communication between crucial parts.
- Application errors on the service level and violation of service-contracts.
- Performance metrics.
- Security violations.

To make it even worse, you must track all those parameters across several clusters in real time. Without such a level of monitoring, no high availability can be achieved and the distributed system is even more vulnerable to downtime than a single monolithic application.

The good news is that nowadays there are plenty of tools to measure web-app performance and availability. One of the most interesting is Graylog (<http://graylog.org>).

Used by many microservice predecessors like LinkedIn, eBay, and Twilio, Graylog centralizes logs into streams.



The screenshot shows the Graylog web interface with the 'Streams' tab selected. There are six streams listed:

- Successful user logins**: All successful user logins. 0 messages/second. Must match all of the 2 configured stream rule(s). [View stream rules](#)
- Failed user logins**: All failed user logins. 0 messages/second. Must match all of the 1 configured stream rule(s). [View stream rules](#)
- John workstation realtime debugging (ignore)**: Just me debugging locally. 0 messages/second. No configured rules. [View stream rules](#)
- Exceptions on all platforms**: All exceptions anywhere in the stack. 0 messages/second. Must match all of the 1 configured stream rule(s). [View stream rules](#)
- Firewall messages**: All logs from all firewalls. 0 messages/second. Must match all of the 4 configured stream rule(s). [View stream rules](#)
- HTTP 500's**: All HTTP requests that we answered in the 500 range. 0 messages/second. Must match all of the 1 configured stream rule(s). [View stream rules](#)

Fig. 26: In graylog you've got access to messages in real time with alerts configured for each separate message stream.

Graylog is easy to integrate, leveraging HTTP communication, syslog (with UDP support for minimum network load) or third party log collectors like fluentd²⁹. It can be integrated with e-mail, SMS, and Slack alerts.

The screenshot shows the 'Alerts configuration for stream »Exceptions on all platforms«'. It includes the following sections:

- Add new alert condition**: A form to trigger alerts when there are more than 0 messages in the last 0 minutes and wait at least 0 minutes until triggering a new alert (grace period). When sending an alert, include the last 0 messages of the stream evaluated for this alert condition. [Add alert condition](#)
- Configured alert conditions**: A table with one entry: 'Field value condition' (Alert is triggered when the field millis has a higher millis value than 250 in the last 3 minutes. Grace period: 0 minutes). [Edit condition](#) | [Delete condition](#)
- Callbacks**: A section for selecting a callback type: 'Send to external system' (Selected) or 'Send to another Graylog instance'. [Select Callback Type](#) | [Send to external system](#) | [Send to another Graylog instance](#)

Fig. 27: Alerts configuration is a basic feature for providing HA to your microservices ecosystem.

²⁹ <http://www.fluentd.org/>



Distributed systems require new levels of application monitoring and logging. With monolithic applications you can track one log-file for events (usually) and use some Zabbix triggers to get a complete view of a server's state, application errors, etc.

| New Relic

Whereas Graylog is focused around application logging, New Relic is centered around the performance and numeric metrics of your applications and servers: network response times, CPU load, HTTP response times, network graphs, as well as application stack traces with debugging information.

New Relic works as a system daemon with native libraries for many programming languages and servers (PHP, NodeJS...). Therefore, it can be used even in production where most other debugging tools come with too much significant overhead. We used to work with New Relic on production clusters - even with applications with millions of unique users per month and dozens of servers in a cluster.

We used to implement our own custom metrics to monitor response times from 3rd party services and integrations. Similarly to Graylog, New Relic can set up dashboards and alerts.



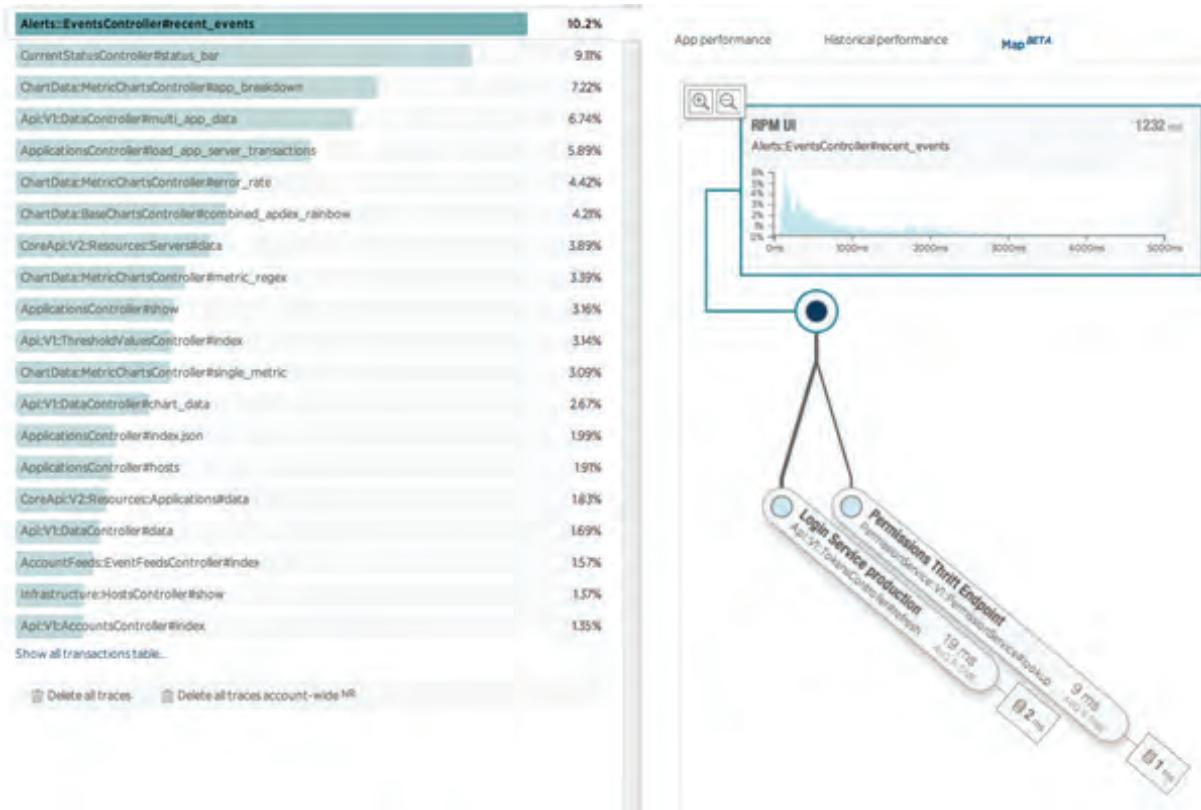


Fig. 28: The coolest feature of New Relic is stack-trace access - on production, in real time.



New Relic Insights

Data visualization tools and customizable dashboards, allow you to observe business analytics data and performance information at the same time.

By combining application, environment and business data - like transactions, pageviews and order details - into one reporting tool, you can more precisely see how your app performance affects your business.

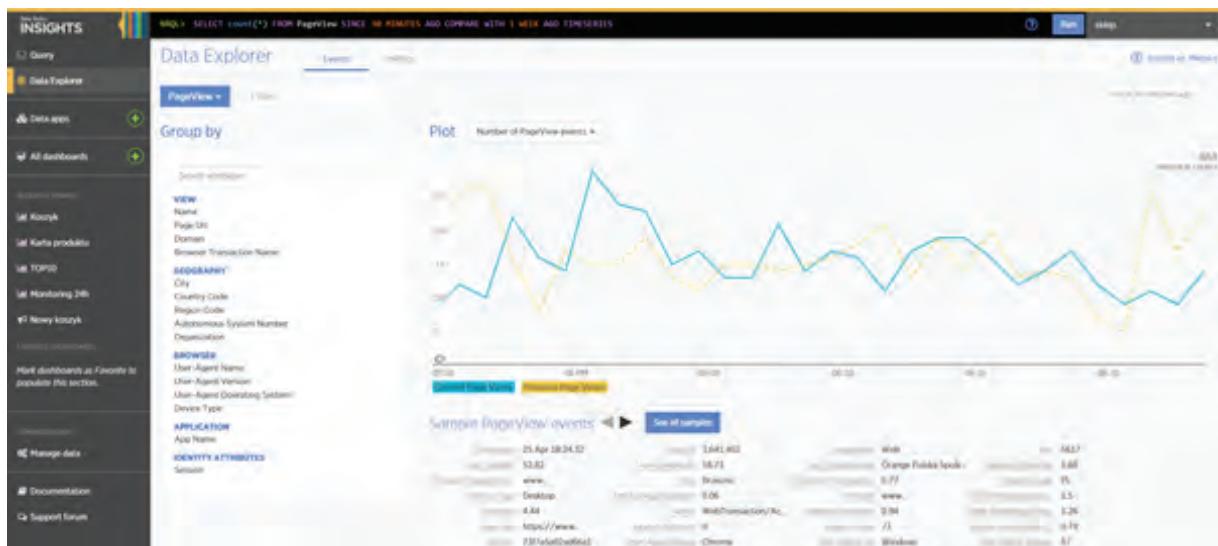


Fig. 29: New Relic Insights Data Explorer with sample plot.



New Relic Insights NRQL Language

You can also use the NRQL (New Relic Query Language) with syntax similar to SQL language to explore all collected data and create application metric reports.

For example, you can attach customer group IDs to order requests to check if particular customer groups have an unusually bad experience during the order process.

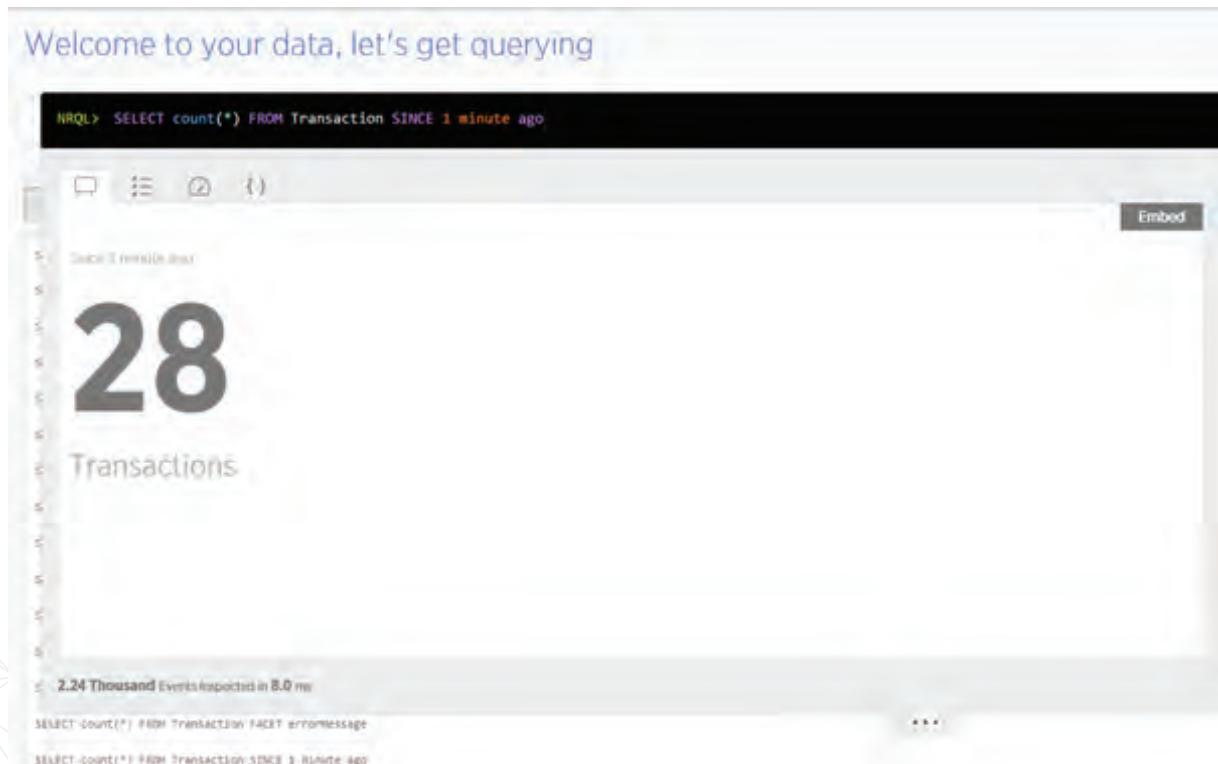


Fig. 30: New Relic usage of NRQL with sample output.



| Take care of the front-end using New Relic Browser

Another powerful feature allows you to easily detect any javascript issue on the front-end of your application. Additionally, New Relic will show you a detailed stack trace and execution profile.



Fig. 31: The New Relic Browser module displays a list of javascript issues on front-end application.



Case Studies:

Re-architecting the monolith



| Case Studies: Re-architecting the Monolith

Here I'll briefly present two case studies of the microservices evolution which I've been able to observe while working at Divante.

| B2B

One of our B2B clients came to us with the following issues to be solved:

- While on Magento 1 with SKUs catalog exceeding 1M products - performance bottlenecks relating to catalog and catalog updates became hard to work-around.
- Monolithic architecture, strongly tied to external systems (such as CRM, ERP, WMS) hindered changes and development of new features.
- CRM that became the SPoF (Single Point of Failure). Pivotal CRM was in charge of too many key responsibilities including per-customer pricing, cart management and promotions.
- Serious amount of technological debt due to legacy code.
- Scalability problems - the platform should be able to handle a new business model that requires broadening the offer and entering new markets.

The online platform was generating 100M+ EUR revenue/year at the time. The challenge was a serious one.



The architecture of this system resembled a "death star". However, its complexity was not between microservices, but between external systems.

The first instinct was to move the site 1:1 from legacy Magento 1 to a new platform. OroCommerce and Magento 2 were considered.

The work on collecting business requirements from stakeholders inside the company and putting them into the Business Requirements Document (BRD) was quickly started. We formulated nearly 1,000 business requirements. Then we mapped them into features. Finally, we scored each available platform on its ability to meet the requirements:

- **Functionality available out of the box.**
- **Functionality after customization.**
- **Functionality requiring additional/external modules.**
- **New features.**

We double-checked both platforms in terms of technical solutions, scalability, performance, possibility of modification and the possibility of further development.

During the analysis, we realized that it would be somewhat risky to collect all the requirements for such a huge platform right away. We felt that before we had finished analyzing the requirements, they would have changed a few times already. Brief research showed us that none of the systems were capable of meeting all the specific requirements, both functional and non-functional. We realized this was not the right approach and could lead us back to where we started - a monolithic application.

Before you decide to take a similar step (to go along with a ready-made platform in the center of a microservices eco-system), look at the pros & cons of this approach.

Pros & Cons of choosing an end-2-end platform:

Pros:

- Rapid development and time to market.
- It's usually a stable, well-tested product.
- A community that will help in solving many problems.
- The possibility to use a large base of ready-made, fully-featured modules (Magento Marketplace).
- Official support from the software provider.
- Official updates, security patches.

Cons:

- It's still a monolithic application that sooner or later will lead us to the starting point - problems with scalability and maintenance.
- Very high licensing costs for the Enterprise version.



- Large platforms require specialists with specific skills for a particular system who can be difficult to acquire.
- Ready-made functionalities often requires serious modifications to fully adapt them, which can lead to incompatibility with the base system - no updates or patches.
- They often provide outdated solutions, limiting the introduction of modern technologies.

| A New approach

Eventually, after conducting a feasibility study, we suggested that our client use a more optimal way of solving the problem.

The fundamental assumption was to abandon migration to a new platform and change the architecture by deconstructing the current system and deploying it as an eco-system of distributed microservices. In order to succeed, we needed an effective analysis and implementation process.



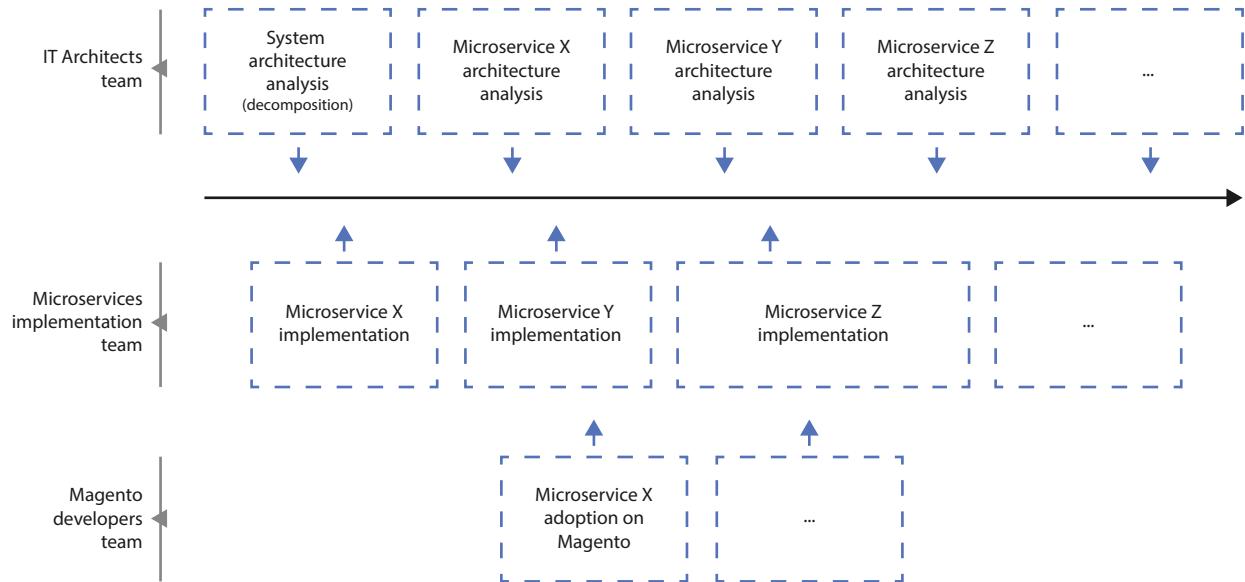


Fig. 7: Agile analysis and implementation process to achieve goals.

The first step of the "architecture analysis" process was the development of a high-level architecture of the entire system by a team of architects, focused on service responsibilities. The results of their work included:

- Key business processes supported by the system.
- Goals and requirements for scalability, security, performance, SLA and potential development directions.
- Identified risks.
- Block diagram of disclosed microservices:
 - Defined scope and responsibility of each service.
 - PaReveal patterns of integration between services, taking into account emergency situations handling, avoiding SPoF.
- Defined events and business objects.
- High-level architecture diagram of the system.

The architects worked together along with the client. The client's domain experts were engaged in session-based workshops using the event



storming technique borrowed from the popular Domain Driven Design (DDD) domain modeling approach. You can find more information on the technique on its creator's blog:

<http://ziobrando.blogspot.com/2013/11/introducing-event-storming.html>.

Based on the collected data, the team provided the implementation team with complete documentation.

After several workshops, a distributed architecture with dedicated main areas/services was created with the following key services defined:

- **PRICING** - managing individual prices and promotions for clients.
- **PIM** (product information management) - responsible for product information and attributes; with planned 1mln+ SKUs it must be implemented as a scalable, probably NoSQL based data warehouse.
- **WMS** (warehouse management system) - product stock management.
- **CRM** (customer relationship management) - in charge of syncing data with Pivotal CRM (orders, statuses, shopping carts ...).
- **REPORT** - reporting and monitoring features.
- **NOTIFY** - user notifications and alerts management.
- **REVIEW** - product reviews system.
- **RECOMMENDATIONS** - recommendations engine.
- **FRONTEND APP** - in the first version - the good, old Magento1; then it was planned to move this layer to a ReactJS + NodeJS thin client.
- **MOBILE APP**.

We started with a 20 page architecture document and then created a list of standards for coding each separate service.



We tried to leverage the HTTP protocol standards, providing documentation and technical requirements, such as specific frameworks and database servers to be used. It's very important to make use of such synthetic and consistent standards while dealing with distributed software.

We decided to start by implementing the first service that is critical for the system due to its SPoF and which would give us the best performance results: PRICING and PIM.

It was crucial to figure out how to separate the platform from Pivotal CRM for calculating end-client product prices and therefore to avoid a SPoF and maintain High Availability (initially the platform used real-time WebService calls to get the prices from the CRM when users entered the page).

PIM was selected to solve problems with growing the SKUs database by moving to an ElasticSearch NoSQL solution instead of Magento's EAV model.

We created these services as separate Symfony3 applications that were integrated with the Magento1 frontend later on.

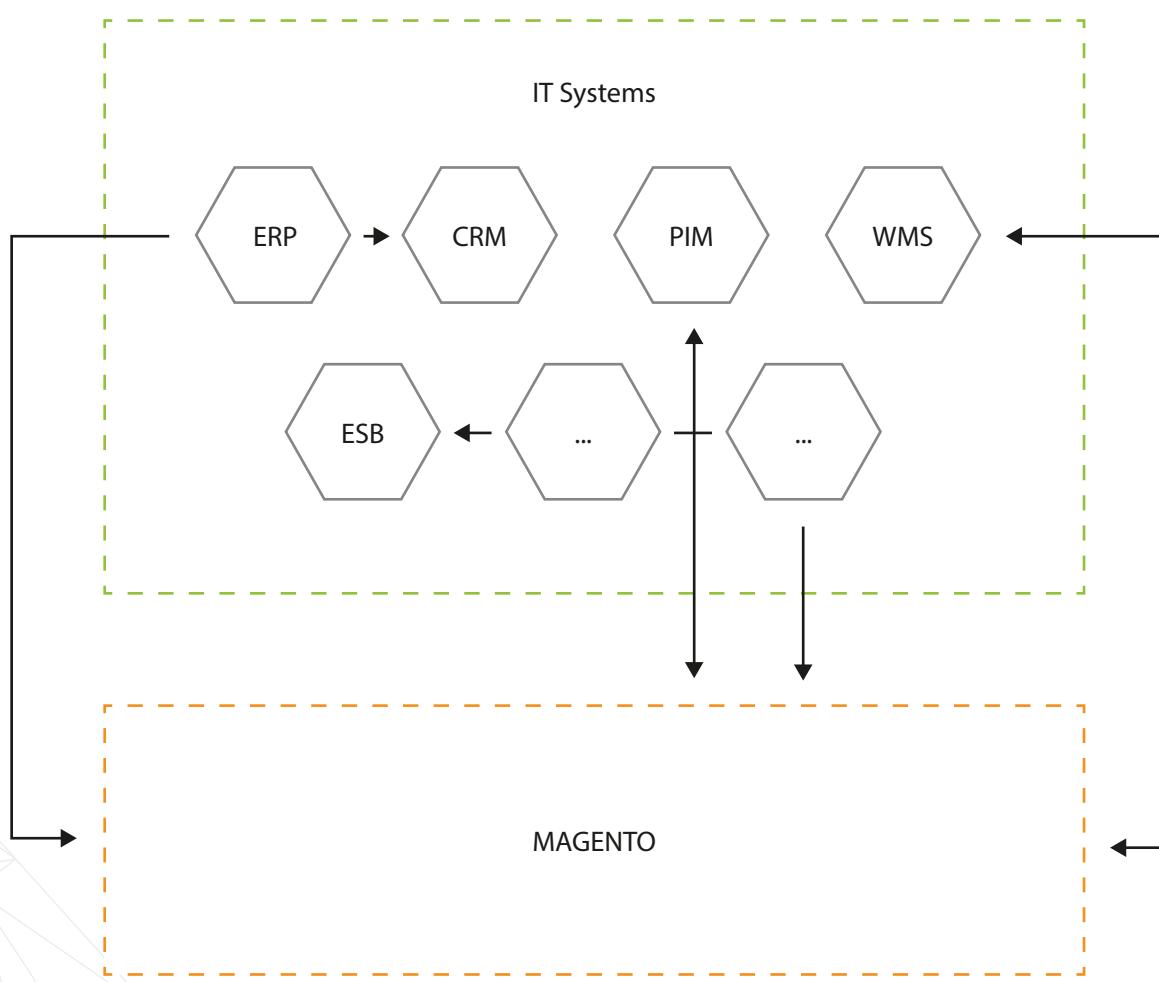
Roughly speaking - we just removed the Magento1 modules responsible for the catalog and wrote our own which called the micro-services instead of hitting the database.

Then we followed this path further, by rewriting and exchanging monolithic modules with distributed services one by one.

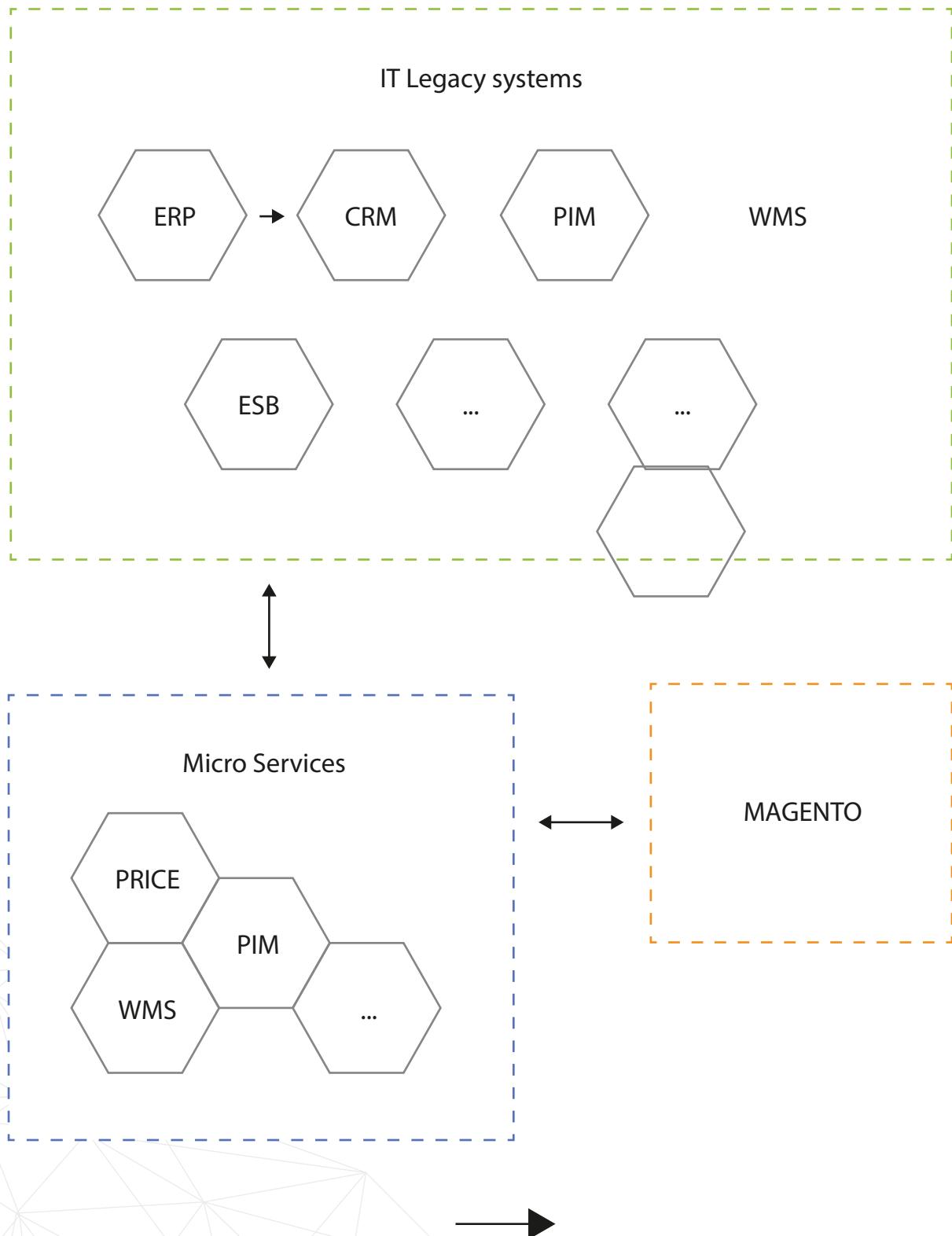


The project was finished with a roughly cut-down Magento (serving only as an application frontend) and 9 services supporting all the business logic. One day, if needed, we can simply move on from Magento, implementing a new frontend using a ReactJS/NodeJS stack or any other modern tech stack.

Beginning



Step 1



Step 2

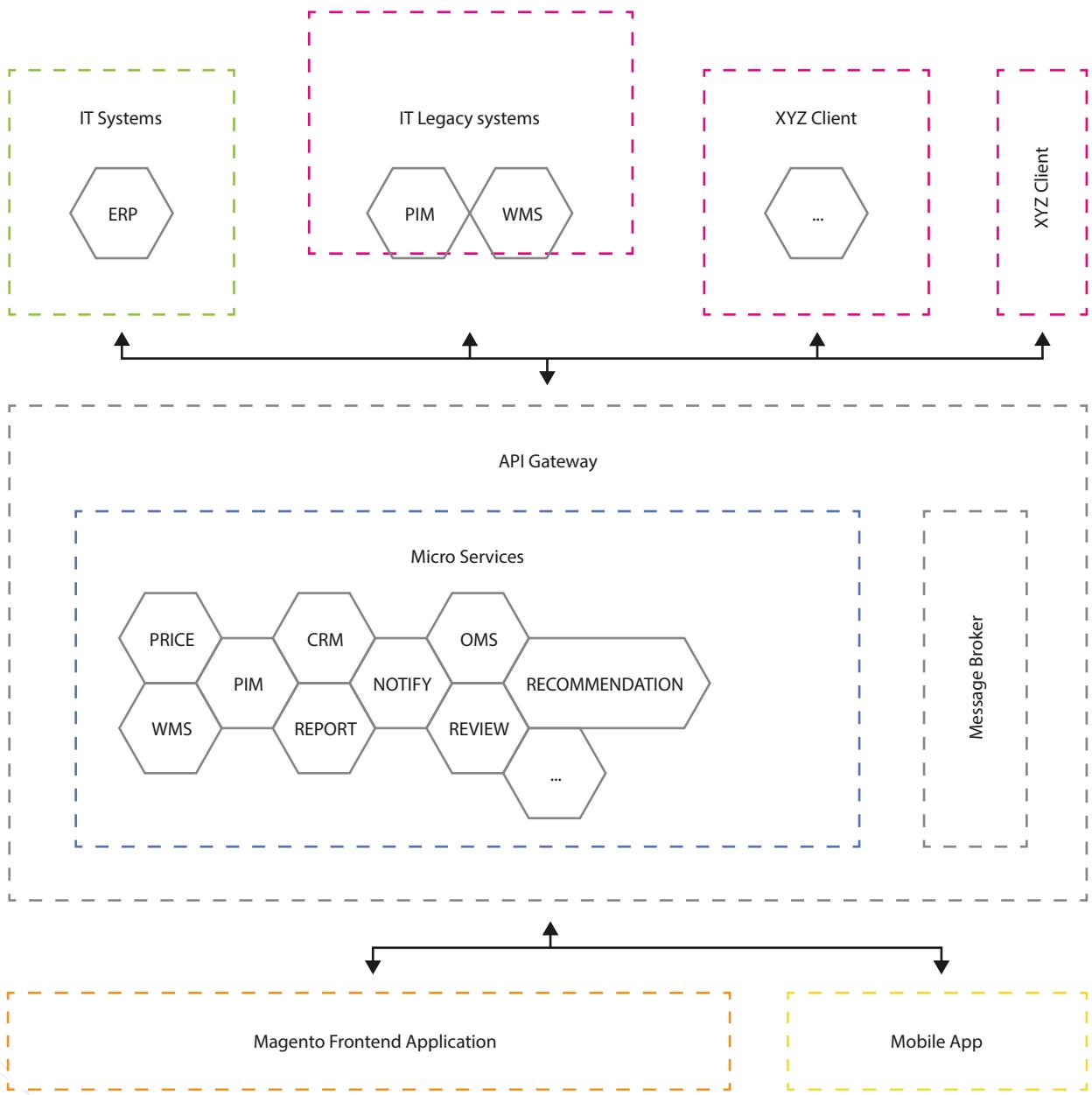


Fig. 8: Evolutionary (notrevolutionary) steps to create a new platform from a monolithic application.



Each service was designed with its own denormalized database (ElasticSearch or PerconaDB for relational data orders) and was designed with high availability in mind. Data between services is exchanged via a RabbitMQ data bus using an Event Driven Data Management approach⁴.

We haven't decided (at this point) to go with any technology other than PHP, so all services were implemented using the Symfony framework; mostly for simplicity, as well as cost optimization of the development process.

You can find more great technologies that focus on microservices later in this book and at <https://github.com/mfornos/awesome-microservices>.

To sum-up our challenge please find our notes on the pros and cons of the microservice approach below:

Pros:

- Small teams can work in parallel to create new, and maintain current, services. Many of you have probably experienced problems with working in large teams, as we did.
- The possibility of using heterogeneous technologies - ElasticSearch for products, PerconaDB for orders.
- Increased critical fault-tolerance by using bulkheads/service contracts.
- Incremental replacement of legacy code and original systems with new, effective solutions.

⁴ <https://www.nginx.com/blog/event-driven-data-management-microservices/>

- Scalability - we can scale only the services that require it.
- Programmers have a lot of fun, so it's quite easy to keep the team motivated.

Cons:

- Extensive client involvement is required during the BA phase.
- New skills and quite a lot of architectural experience is required from developers and architects to design the initial phases.
- New challenges in maintaining the monitoring of the entire infrastructure.

Mobile Commerce

One of the coolest features of the microservices architecture is that you're no longer bound to your one-and-only platform. It's crucial, particularly when the application at hand has to meet different expectations. In our case - an eCommerce platform with dental equipment - we have three different areas to be covered:

- State-of-the-art content management system with e-learning features.
- Basic eCommerce features - checkout and promotions for ordering dental equipment. CRM features, user profiles and segmentation for tracking all the users.

The platform was designed to work on mobile devices only.



At the start we considered whether or not to use one platform for the backend, or maybe to write dedicated solutions. It's hard to find software with enterprise level CMS, PIM, CRM and eCommerce features altogether.

Therefore we decided to go with the following software products:

- Pimcore - as a CMS and PIM; we created all the content (e-learning, static pages, product content) in Pimcore⁵ and expose it via API.
- Magento2 - as a checkout and for eCommerce features.
- Dedicated iOS and Android apps for the frontend.

We used the "Backend for Frontends" approach described in this eBook to provide optimized API gateways for both mobile applications and the RWD website. Key areas like product content and e-learning pages were fully manageable in Pimcore and provided the end client with HTML renderings.

Magento checkout was integrated using API REST calls for placing orders.

Nowadays, all new open source products (and of course, not just open-source) expose most of their features via API. It's cool to focus on the end client's value (frontend) and not reinvent the wheel on the backend.

We did almost no custom development work on the backends!

⁵ <http://pimcore.org> - Enterprise grade Content Management platform, PIM and DAM

Appendix 1: Microservices and unicycling by Alexander Graf

Thanks to Alexander Graf, the founder of Spryker.com for this part. Initially published as a blog post on Alexanders' blog: <https://tech.spryker.com/microservices-and-unicycling-9ed452998b77>.

After the unspeakable NoSQL hype of about two years ago had reached its peak “Why are you still working with relational databases?”, the topic of microservices was brought to the fore in discussions about back-end technologies. In addition, with React, Node & Co., the front-enders have developed quite a unique little game that, it seems, nobody else can see through. After about two years of Spryker, I have had the pleasure of being able to follow these technical discussions first-hand. During my time with the mail order giant Otto Group, there was another quite clearly defined technical boogeyman—the so-called Host System, or the AS400 machines, which were in use by all the main retailers. Not maintainable, ancient, full of spaghetti code, everything depended on it, everything would be better if we could be rid of it and so on and so forth—so I was told. On the other side were the business clowns—I’m one, too—for whom technology was just a means to an end. Back then, I thought those who worked in IT were the real hard workers, pragmatic thinkers, who only answered to the system and whose main goal was to achieve a high level of maintainability. Among business people there were, and there still are, those I thought only busied themselves with absurd strategies and who banged on about omnichannel, multi-channel, target group shops and the like. Over the last eight years of Kassenzone.de, these strategies were always my self-declared final boss. It was my ultimate aim to disprove them and demonstrate new approaches.



To my great disappointment, I have come to realize that people in IT—or ‘developers’ as they are called today—work with the same thought processes as the business clowns. There is an extremely high tendency to chase after trends and basic technical problems are not sufficiently analyzed, nor are they taken seriously enough. Microservices is a wonderful example of this. It is neither an IT strategy, nor is it an architecture pattern. At most, it describes just a type of IT and system organization. Just like omnichannel. Omnichannel doesn’t mean anything. It’s a cliché that is pretty much just filled with “blurb” and the same way of thinking is apparent on the topic of microservices. From the outside, omnichannel can be seen as the result of strong growth if a company’s range of services can, therefore, cause it to become a leader in many channels. This is exactly what happens with microservices, which may be the result of strong growth in IT, because you have to divide large applications into services so that you don’t have too many developers working on them at the same time. But this is far cry from being an IT strategy. In many conversations at the code.talks conference, this impression was (unfortunately) confirmed. Yoav Kutner (founder of Magento1) cut his teeth on the rollout of the first of the big Magento1 projects and reports with a shake of the head that developers always follow the next hype without having considered where the real problem lies. Yes, I know that that sounds all very general, but let’s have a closer look at the topic of microservices.

Martin Fowler, IT guru and champion of microservices has written dozens of articles on the subject and describes microservices as follows:



“

In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and are independently deployable 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.

This does sound quite promising and it can also help with the corresponding problems. Otto's IT team has already reached the Champions League where this is concerned and produced the obligatory article, called “On Monoliths and Microservices” on the subject. Guido from Otto also referred to this topic at the code.talks event:

“

When we began the development of our new Online Shop otto.de, we chose a distributed, vertical-style architecture at an early stage of the process. Our experience with our previous system showed us that a monolithic architecture does not satisfy the constantly emerging requirements. Growing volumes of data, increasing loads and the need to scale the organization, all of these forced us to rethink our approach.

There are also other examples which benefit excellently from this approach. Zalando is an example of a company which is open about using it in “From Jimmy to Microservice”. The approach can also crop up for quickly growing tech teams, such as that of Siroop.



What's often forgotten when people sing its praises are the costs associated with such an approach. Martin Fowler calls these costs the Microservice Premium and clearly warns against proceeding in this direction without caution:

66

The microservices approach is all about handling a complex system, but in order to do so the approach introduces its own set of complexities. When you use microservices you have to work on automated deployment, monitoring, dealing with failure, eventual consistency, and other factors that a distributed system introduces. There are well-known ways to cope with all this, but it's extra effort, and nobody I know in software development seems to have acres of free time. So my primary guideline would be don't even consider microservices unless you have a system that's too complex to manage as a monolith.

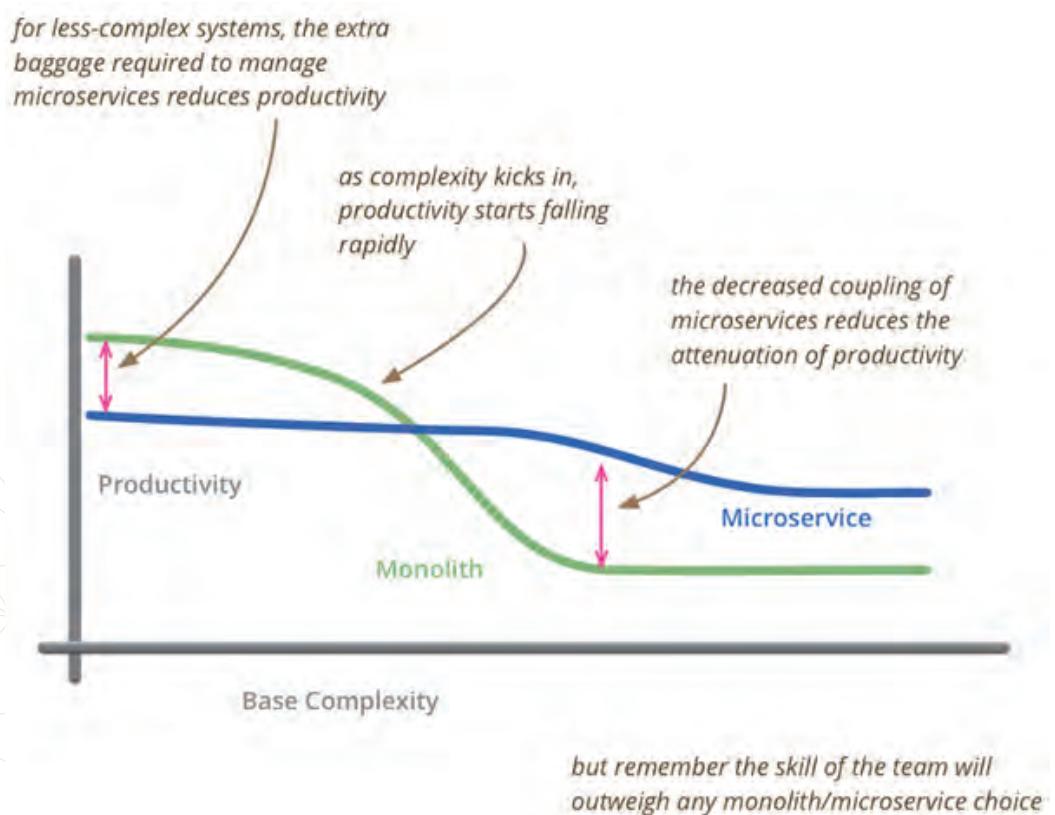


Fig. 29: Image: <http://martinfowler.com/>.



Technically speaking, this restriction has many different origins. Whether it's the latencies which must be called up for one procedure due to dozens of active services, clearly difficult debugging, the demanding hardware setup or the complex data retention (each service has its own database). The fundamentally hard to manage technology zoo notwithstanding. Here, excellent parallels to eCommerce organizations can be drawn. Who is quicker and more effective in the development and scaling of new models:

1. a company with dozens of departments and directorates, which must be in a permanent state of agreement, but which are extremely good in each of their individual disciplines;
2. or a company at which up to 100 employees sit in one room, all knowing what's going on and all talking to each other.

The second example is quicker, that goes without saying. With larger models, at which scaling is a rather uniform approach, the first example is better. Not much is different in the case of microservices. To understand this context better, Werner Vogels' (Amazon CTO) test on his Learnings with AWS³⁰ is highly recommended:

“

We needed to build systems that embrace failure as a natural occurrence even if we did not know what the failure might be. Systems need to keep running even if the "house is on fire." It is important to be able to manage pieces that are impacted without the need to take the overall system down. We've developed the fundamental skill of managing the "blast radius" of a failure occurrence such that the overall health of the system can be maintained.

³⁰<https://www.thoughtworks.com/insights/blog/monoliths-are-bad-design-and-you-know-it>



Although there are, therefore, really good guidelines for sufficient handling of the topic, so as to find out whether microservices make sense for an IT organization (not for most!), you can regularly find contributions such as that by Sam Gibson³¹ online or on conference panels:

“

In principle, it is possible to create independent modules within a single monolithic application. In practice, this is seldom implemented. Code within the monolith most often, and quickly, becomes tightly coupled. Microservices, in contrast, encourage architects and developers the opportunity to develop less coupled systems that can be changed faster and scaled more effectively.

In Kassenzone reader's language, this pretty much means: Pure Play business models are good if they are implemented in an orderly fashion but it will only really come good if you run many channels well. The winning strategy is omnichannel. Now, you could simply brush such statements off, but it is astounding just how quickly and strongly such simple thought processes spread and become the truth all by themselves. The voices which oppose them³² are quiet in comparison, but the arguments are quite conclusive.

“

In principle, it is possible to create independent modules within a single monolithic application. In practice, this is seldom implemented. Code within the monolith most often, and quickly, becomes tightly coupled. Microservices, in contrast, encourage architects and developers the opportunity to develop less coupled systems that can be changed faster and scaled more effectively.

³¹<https://www.thoughtworks.com/insights/blog/monoliths-are-bad-design-and-you-know-it>

³²<http://blog.cleancoder.com/uncle-bob/2014/10/01/CleanMicroserviceArchitecture.html>

Technically and methodically, a lot is said for the use of “good” monolithic structures for a great deal of eCommerce companies, but doing so requires a lot of effort producing good code, something which, in the short term, you don’t have to do in the microservices world. If, then, a mistake in the scaling arises, the affected CTOs would probably wish they had the AS400 system back.

The founder of Basecamp has hit the nail on the head with his own system, which he describes as “The Majestic Monolith”³³. And, where content is concerned, I’m with him:

“

Where things go astray is when people look at, say, Amazon or Google or whoever else might be commanding a fleet of services, and think, hey it works for The Most Successful, I’m sure it’ll work for me too. Bzzzzzzz!! Wrong! The patterns that make sense for organizations’ orders of magnitude larger than yours, are often the exact opposite ones that’ll make sense for you. It’s the essence of cargo culting. If I dance like these behemoths, surely I too will grow into one. I’m sorry, but that’s just not how the tango goes.

It’s bit like if companies who own an old bicycle, which they don’t know how to ride properly, want a little too much. They see the unicyclist at the circus performing dazzling tricks on his unicycle and say to themselves: My bike is too old, that’s why I can’t ride it. I’ll just start with a unicycle right away, at least that’s *forward-thinking*.

³³ <https://m.signalvnoise.com/the-majestic-monolith-29166d022228#.90yg49e3j>



Appendix 3: Blogs and resources

There are plenty of websites, blogs and books you can check to read more about microservices and related architectural patterns. The book "Building Microservices, Designing Fine-Grained Systems" by Sam Newman and O'Reilly Media

(<http://shop.oreilly.com/product/0636920033158.do>) should be at the top of the top of your list. The most important information has been collected into one place. It is all you need to know to model, implement, test and run new systems using microservices or transform the monolith into a distributed set of smaller applications. A must-have book for every software architect. O'Reilly Media has also released another interesting book, "Microservice Architecture" by Irakli Nadareishvili, Ronnie Mitra, Matt McLarty and Mike Amundsen

(<http://shop.oreilly.com/product/0636920050308.do>), which is also worth a read.

With knowledge from Sam Newman, you should be ready to discover websites like:

- <http://microservices.io>,
- <https://github.com/mfornos/awesome-microservices>,
- and <https://dzone.com/> (under „microservices” keyword), curated lists of articles.

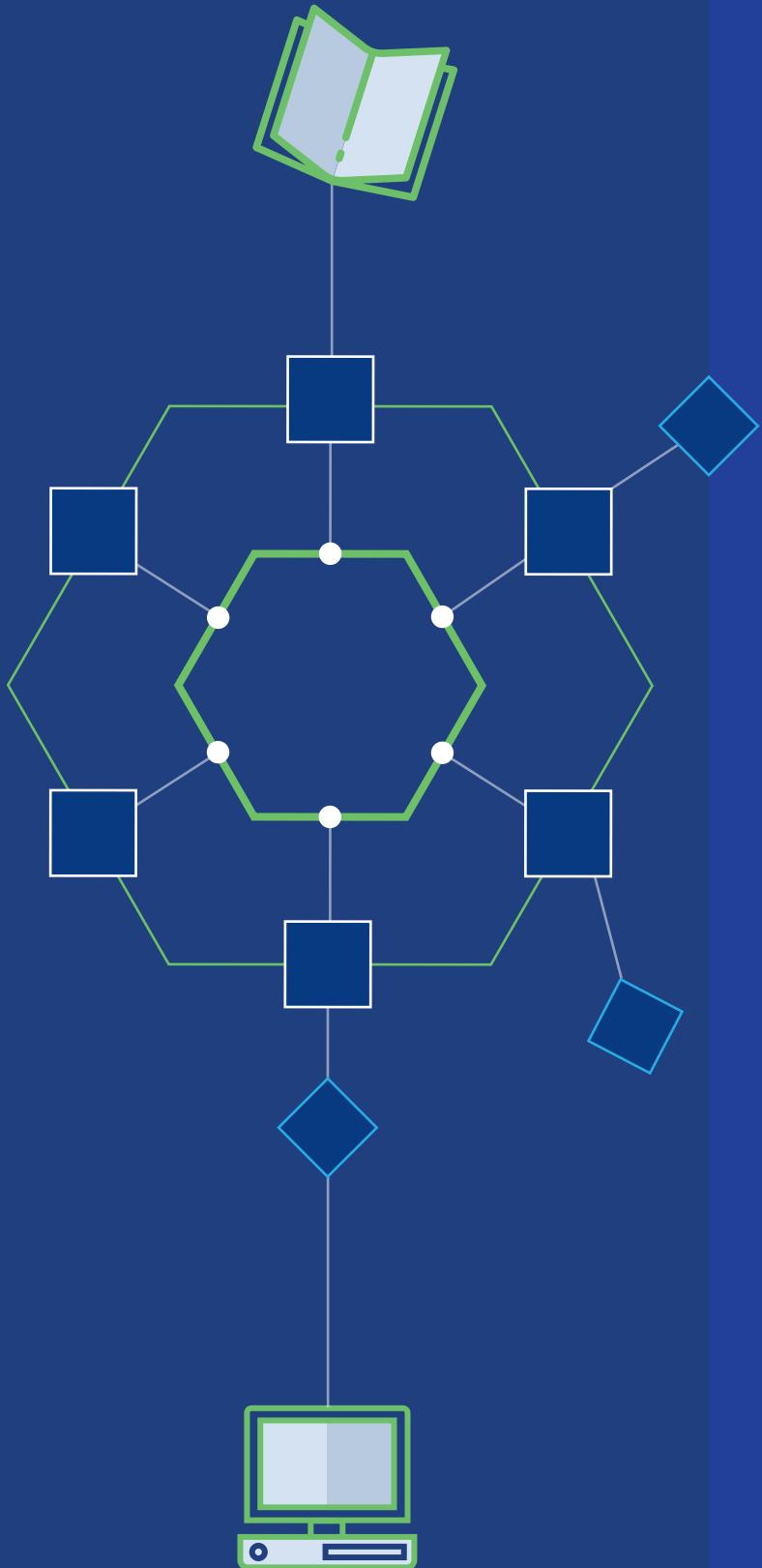
It's a condensed dose of knowledge about core microservice patterns, decomposition methods, deployment patterns, communication styles, data management and many more... There you can also find many interesting presentations and talks recorded at conferences. The last website specifically, <https://dzone.com/>, should be very interesting for IT people.



Depending on your time, you can subscribe to the newsletter “Microservices Weekly” (<http://www.microservicesweekly.com>) for a weekly set of articles about architecture and container-based virtualization or visit the Microservices section at the InfoQ website (<https://www.infoq.com/microservices/>), one of the most important websites with articles and talks related to software development.

As you can see, knowledge is all around us. Don’t forget about Martin Fowler and his “Microservice Resource Guide” (<https://martinfowler.com/microservices/>). Martin is Chief Scientist at ThoughtWorks, the publisher of “Technology Radar” (<https://www.thoughtworks.com/radar>; highly recommended as well) and author of a few bestselling books. Martin Fowler’s wiki is a Mecca for software architects and “Microservice Resource Guide” is only one of them...





Thank you!

If you want to know more about microservices, just drop me a message at pkarwatka@divante.co.

www.divante.co



Loyalty Platform in Open Source

with ready-to-use features

Try Our Solution



What can you build with OpenLoyalty?



Loyalty program for retail with
online and offline stores



Loyalty module for eCommerce



Motivational program for
sales departments



Customer care program