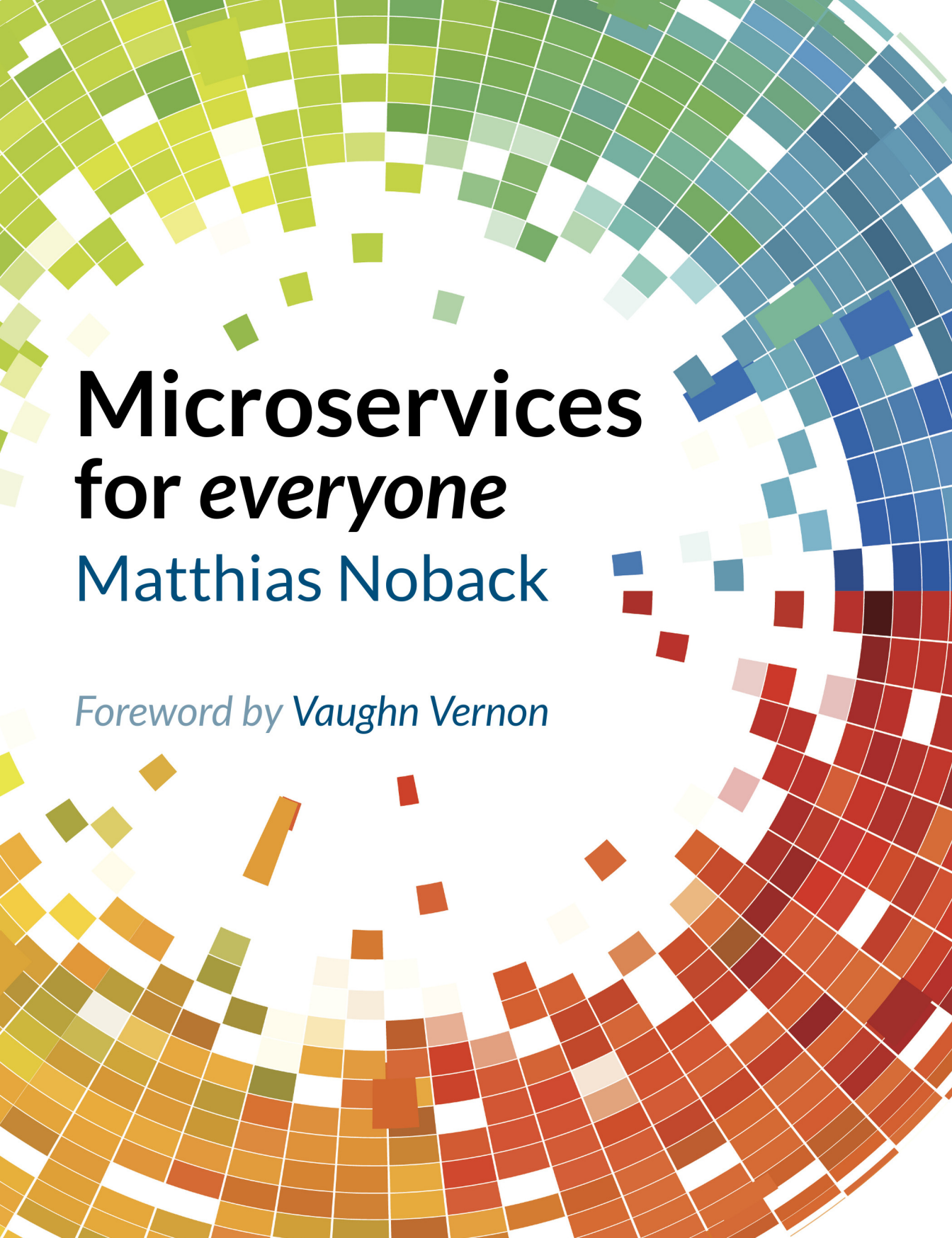

Microservices
for everyone
Matthias Noback
Foreword by Vaughn Vernon

Microservices for everyone

Matthias Noback

This book is for sale at http://leanpub.com/microservices-for-everyone

This version was published on 2017-09-25

Also By Matthias Noback

A Year With Symfony

Un Año Con Symfony

Principles of Package Design

Один год с Symfony

To Lies, Lucas & Julia

Contents

Foreword

Preface

Scepticism

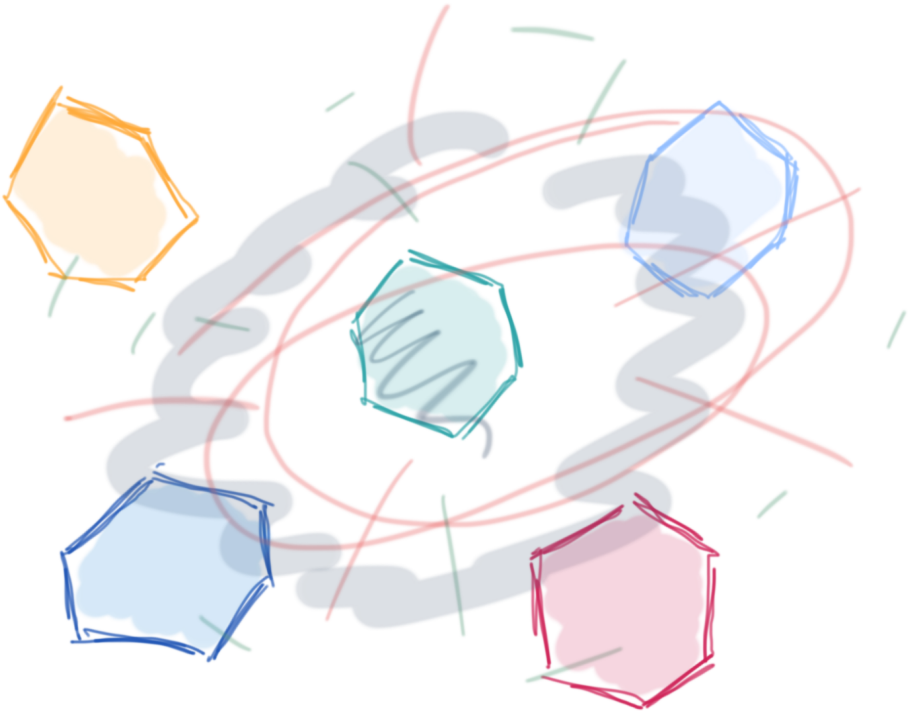
I can almost hear you think: "Bah, microservices. Nothing good could come from that!"…

> We replaced our monolith with micro services so that every outage could be more like a murder mystery.
>
> — Honest Status Page[1] (2015)

I am absolutely certain that this a solvable problem, but nonetheless, it may scare you away from considering a microservice architecture as a viable choice for your project. Especially since you receive reminders of what a bad choice that may be on a daily basis (at least if you're on Twitter):

> If one piece of your web of microservices suffers an outage and your whole system crashes and burns, then you have a distributed monolith.
>
> — Matt Jordan[2] (2016)

[1] https://twitter.com/honest_update/status/651897353889259520
[2] https://twitter.com/mattcjordan/status/811286734369681408



Which makes me wonder: how is this different from when we have a single application? If something goes wrong in a monolith we usually throw an exception, and let the application crash, right? Is it even possible to make our system as resilient as gets depicted here? When the disaster is too big, there may be nothing we can do to recover from it. Still, I'm certain that with a few simple implementation patterns we can make our microservice system much more resilient than any monolith we have encountered so far.

> If your microservices must be deployed as a complete set in a specific order, please put them back in a monolith and save yourself some pain.
>
> — Matt Stine[3] (2016)

[3] https://twitter.com/mstine/status/755470158861217792

This sounds like good advice though. One of the main design goals of microservices is that services can be created and removed on-the-fly and other services shouldn't produce any failures because of that. If this is not the case, indeed we should get back to our monolith. But not too fast! There are some good solutions available:

> Microservices *without* asynchronous communication are as good as writing monolith app.
>
> — Ajey Gore[4] (2016)

Asynchronous, event-driven communication is one way to approach the dependency problem. But it is not the one and only solution. In fact, as we will see later on, synchronous communication is still a viable solution. It needs a bit of extra work though. But as soon as you find out how to solve things in an asynchronous fashion, you'll be looking for other places where you can switch from synchronous to asynchronous communication.

I'm sure there are plenty of teams that decided to make their next project a microservices project. It then took a lot of research and a lot of work and the project still ended up in quite a bad shape. There are many reasons for this. Probably most of those reasons are the same as for any other kind of software project: the usual issues related to estimations, deadlines, budgets, etc. Or, as often happens, developers were eager to try something new, to escape from the suffocating work on the "legacy system", seeking their salvation in a microservice architecture. Or, they were able to run their services on their own machine, but had trouble getting the whole system up and running, monitored and all, in an actual production environment.

Optimism

While still surrounded by *microservice negativism* the tech community has in fact been floating on a wave of *microservices hype*. Trusting on my built-in "tech radar" and "mood calculator" though, it feels like we're almost past this hype. If I look around me, we're more in the *assess* phase: "This could be something for us, let's find out." And I agree, it's time to prove that this can work. It's my current belief that we need the following ingredients for that:

[4] https://twitter.com/ajeygore/status/723905406863433728

- We need to put a lot of focus on our *domain* and create (but also continuously refine) a suitable model for it. In order to do so, we need to apply Domain-Driven Design (DDD), take a sincere interest in the business domain and find out how we can contribute by creating software.
- We need to consciously and continuously look for ways to refine our service boundaries and how services are *integrated*.
- We need to develop some *organizational awareness*, and look for bottlenecks in the way teams are structured and how they communicate.
- We need to adopt a *DevOps* mindset, since we need to be able to set up and manage the infrastructure that runs our services.

That's a nice little list, but it might represent a lot of work for you. Not necessarily programming work, but *learning* work. And this is generally the hardest kind of work. As Alberto Brandolini[5] puts it: "Learning is the bottleneck". This quote itself is derived from Dan North's article "Deliberate Discovery"[6] (2010), who says that it's not learning but *ignorance* which is the "single greatest impediment to throughput". Looking at the list of ingredients above, you may well find that you're not quite ready to start you microservice journey, or maybe *you* are, but your team is not. You may not have much experience with DDD, you may not be concerned with organizational structures, and you may not like fixing things on a server. And above all, you may feel that you don't have enough time to learn it all.

My first comforting message to you is: *you are not alone*. Looking at online lists of resources on various topics that might interest programmers, it becomes apparent that the target audience is expected to only take an interest in programming languages, programming techniques, OOP principles, patterns, frameworks and libraries. Take a look at lists of resources like Java Annotated Monthly[7] from IntelliJ, or in my own community, PlanetPHP[8] or PHPDeveloper.org[9], and you'll notice that almost nobody seems to concern themselves with Domain-Driven Design, even less so with DevOps.

My second comforting message to you is: *it's not too late to catch up*. In fact, at this very moment *it's easier than ever before*. All over the world local communities

[5] http://www.slideshare.net/ziobrando/optimized-for-what/30?src=clipshare
[6] https://dannorth.net/2010/08/30/introducing-deliberate-discovery/
[7] https://blog.jetbrains.com/idea/tag/java-annotated/
[8] http://www.planet-php.net/
[9] http://phpdeveloper.org/

are gathering in meetup groups about Domain-Driven Design and DevOps. It's not just local meetups, there are international conferences on these topics too, like DDD Europe, DDDx, DockerCon, etc. And besides a large number of learning initiatives, we now have a lot of powerful yet easy-to-use tools available. We can create stand-alone deployable artifacts for our software using Docker, and deploy them using Docker Swarm, or Kubernetes, or integrated, even more user-friendly solutions on top of these container orchestration tools.

I'm confident that learning about microservices will pay off. It should shake out many issues that had so far been hidden from sight, swept under the carpet, or worked around for ages. Think of issues like:

1. Spaghetti code; everything knows about everything and can use any function or piece of data available in the entire system.
2. Single-person, delayed deployments; only one person in the organization knows how to deploy the application, and does so only every week, month or quarter.
3. Teams breaking the applications of other teams; they have trouble integrating their applications, which almost never succeeds in one go. Hence, they fear releasing their software (or only dare to do it in a coordinated fashion, late at night).
4. Teams not being able to decide upon the best course of action, hence doing a lot of rework, or delivering sub-optimal solutions.
5. Vendor lock-in; hosting providers that offer only a certain set of services (e.g. Nginx, MySQL, Memcache; while you would like to use Apache, Cassandra and Redis).
6. Scaling issues; in order to accommodate higher demand, you've only focused on performance optimizations in the request-response flow, applying patches everywhere, caching results, etc. There is a limit to what your current vertically scaled setup can handle, but you don't know how to prepare for the next step.

Adopting microservices is going to make all these issues clear, out in the open and ready to get fixed:

1. You can and need to start isolating data, and related behaviors.

2. You and everyone on your team will be able to release their software and, if you want, even deploy it to production servers.
3. You will be forced to explicitly define contracts for each service: how can other services communicate with it, which events does this service expose, etc. You have to think harder about explicit interfaces and focus on use cases first.
4. Because the size of each service is relatively small, most of your design decisions only reach as far as the boundaries of the service itself. This means you can try radically new approaches and fall back on more traditional solutions if it doesn't work out as expected.
5. Working with microservices allows you to try different types of databases and different technologies in general. This offers even more opportunities to experiment.
6. With microservices, scaling gets another dimension. Instead of looking for bigger, stronger machines, you can now invest in more, yet simpler machines. Resource usage will be distributed more evenly, in particular when you start using asynchronous communication.

Whether or not you are actually going to create microservices, the things you're going to learn about modularization, team work, domain modelling, and operations is useful either way.

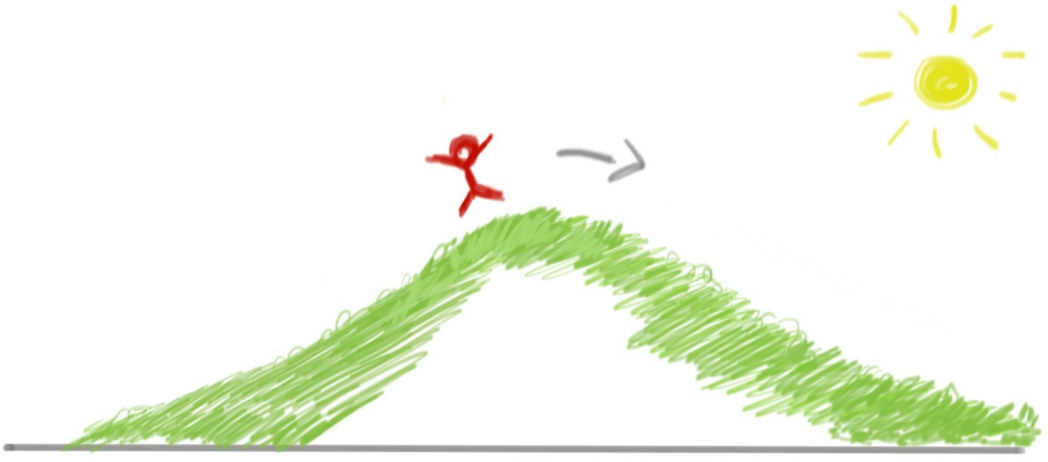
Why I have to write this book

I've been developing web application since 2003. At the risk of sounding like an old man: I've seen many things come and go. A couple of years ago I realized that my work as a software developer has become much more interesting than it was before. My activities started to stretch further than my keyboard could reach. With the advent of Domain-Driven Design and the Docker ecosystem, I feel more empowered to deliver useful software than I ever did before.

I feel that I'm the right person to write this book, as I enjoy writing, but I also enjoy reading. I've read a lot about microservices and adjacent topics, like Domain-Driven Design (DDD), continuous delivery, messaging integration, Docker, etc. It's crucial to note though that so far I have not had the opportunity to work on a large microservice system. So this book won't contain wild stories from the trenches. Instead, this

is a book about the technologies involved in a microservice architecture, focusing mostly on the software development involved, and how you can make the best design decisions.

My hypothesis is that over the past few years the tech community has been working their way towards the peak of *Microservice Impediments*. I want to show you that we are at a point where the overhead of implementing a microservice architecture starts being smaller and smaller, and is currently at least small enough to justify it, even for smaller teams. *You don't have to be Netflix or Amazon to benefit from building your software as (micro)services.*



We've reached the peak of Microservice Impediments

Design guidelines for this book

Since writing a book is a daunting task which can quickly get out of hand, I know that I'm better off with an explicit list of guidelines. This should help me decide on a case-by-case basis if I should write more, or if I should just stop writing.

Because I want to combine insights from DDD with practices of continuous delivery we're going to use Docker to create containers running single services. Each service encapsulates part of the overall domain model, so they are bounded contexts. This isn't a book about Docker though, nor about DDD, so I won't explain everything to you. Instead, I'll give you:

> Short summaries, a little bit of background, quotes, and pointers.

Microservices come with an entire ecosystem. It would be impossible to provide you with the best solutions, the ultimate or ideal ones. In fact, I couldn't do that, because each situation requires a different solution, to be determined based on the particular context. In this book I want to provide simple solutions (to prove my point that you don't really need to put a lot of work in it to arrive at a *minimum viable solution*). So:

> A few lines of code should be enough to show that it can work.

Like almost every other tech book, this book will only show you the happy path. Since many people have been talking quite negatively about microservices, warning about their dangers—and rightly so—it would be unfair to ignore the problems. So:

> For each overly simplistic solution, mention a few things to consider once you really start implementing microservices.

I'd like to make the code examples in this book as generic as possible, in order to be read and understood by people who are familiar with any programming language. Also, like in my previous books:

> Code samples will be written in PHP.

If you know Java: PHP is much like Java, just ignore the dollar signs. The main reason to choose PHP is that it's my "native" language. However, an important second reason is that the PHP community needs to be shown that, although they work with a language that is not so well-designed, you can still do great things with it.

Rigor?

If you have read my previous book, "Principles of Package Design"[10] (2015), you know that I'm generally a man of rigor. I want things to be exactly right. Given a technical subject, I want to know what I'm talking about, so I'll investigate it until I'm sure that:

[10] https://leanpub.com/principles-of-package-design/

1. I won't say anything about it that's incorrect, and
2. I won't give anyone bad advice about it.

So far, this approach has worked out well. I'm not a troubled perfectionist. I just know that the only way to go fast is to go well. However, I must admit that sometimes I get lost in a subject. In particular in the areas in which I'm less well versed, like *system operations* (SysOps). I'm learning my way in infrastructure-land, but the situation for me is sub-optimal at the moment and everything is still a time-sink, like programming was when I first started with it. In fact, *I know that I do not know*. The danger is, of course, that what I do or preach in this area is not the best thing one could do or preach. I've decided to let go of the feeling of uneasiness coming with that and to rely more on the feedback that should come back to me, the moment I publish something. I accept the fact that I don't have to know everything about everything, and that help will always be given to those who ask for it.

Ethics

If I ever want to finish this book, I can't make it complete, rigorous, nor entirely correct. I'll have to cut some corners. In order to keep things moral though, for the both of us, I have to define my own ethics first. I like how Nassim Nicholas Taleb explicitly defines his own *ethos* in the introduction of his pretty heavy book "Antifragile—things that gain from disorder" (2014). I never finished that one, but nevertheless got some interesting ideas from it (this is the least a book should offer to me, if it stops doing that, I'll put it away).

Nassim writes that "If the subject is not interesting enough for me to look it up *independently*, for my own curiosity or purposes, and I have not done so before, then I should not be writing about it at all, period." Of course, external sources are not banned, nor deemed worthless. But he doesn't want his writing to be directed by these. "Only distilled ideas, ones that sit in us for a long time, are acceptable—and those that come from reality." This is something I'd like to do myself too. I've read many books on software development and have developed lots of software, and would like to speak both from experience and existing knowledge about ideas that have been boiling for quite some time now.

In order to keep myself high-spirited, I'll use my internal compass, which revolves around *procrastination*. I know it's a pretty negative concept and I'm sure you all have some experience with it. There are times when I think I have to do activity *A*, while I'm craving to do activity *B* and often just start doing *B* anyway. As long as the lives around me don't completely derail, there are many good aspects about doing *B*, while not doing *A*. Again, Nassim has some interesting words about this:

> A very intelligent group of revolutionary fellows in the United Kingdom created a political movement [...] based on opportunistically delaying the revolution. [...] In retrospect, it turned out to be a very effective strategy, not so much as a way to achieve their objectives, but rather to accommodate the fact that these objectives are moving targets. Procrastination turned out to be a way to let events take their course and give the activists the chance to change their minds before committing to irreversible policies.

I often find that not doing *A* will let me discover things about *A* that were wrong about it. Sometimes *A* isn't the best thing I can do to achieve a certain goal. There may be more effective ways. Maybe *A* is not helpful at all, or even harmful. Besides, *B* is nicer to do, more energizing at this moment. And it might put me on a trail to some place else, or provide me with a more interesting and compelling journey.

This is why I'll make writing as fun and interesting as possible. I'll likely come up with exotic topics, interesting implementation discussions, fun little open source libraries, and if I notice my attention drifting away, or if I find myself bored by the writing itself, I'll focus on some other topic, trusting that you would otherwise get bored too.

Some meta-comments

The code and configuration samples in this book should be pleasant to read. I aim for short lines to make them fit on a book page. Sometimes I skip lines containing trivial or annoyingly complicated code, to allow you to focus on the parts that matter. Whenever I do so, I make sure to comment on what's left out, always in a syntactically correct way. I recommend you to take a look at the full code and

configuration, all of which is publicly and freely available under an MIT license on GitLab. You'll find a project for most of the chapters under the `microservices-for-everyone` organization[11].

There will be many quotes and references to relevant sources. Whenever possible I add a hyperlink, but some sources are books, which can't be easily linked to. For every source I add the year of publication too.

This book contains many illustrations. They're drawn by myself, unless otherwise stated. In case you're interested: I've used Autodesk Sketchbook for Mac and a simple Wacom Bamboo tablet to create them.

Overview of the contents

The *Introduction* starts with a brief discussion of the term "microservice" and the properties by which a microservice architecture are generally defined. We learn about the microservice maturity model, which gives us some ideas on where to start and how to improve.

We continue with two chapters building up the first pillar of a successful, *modularized microservice architecture*. We need to make sure that our separately designed modules (also known as "services") can be independently deployed. In the chapter *Independent deployability and Polyglotism* we take a look at Docker and its ecosystem of tools for managing container images and deploying containers to a cluster. To make independent deployment of services safer, we discuss continuous delivery and different styles of automated testing in the chapter *Testability and independent manageability*.

What follows constitutes the bulk of this book; the topic of *Cohesive microservice architecture*. We consider the basics of *Communication styles* between applications and we take a look at some *Implementation examples* of those communication styles. The following chapters cover different sub-topics of service integration: *Data and data formats*, *Persistence*, *Data locality*.

The latter chapters being very technical in nature, the next three chapters discuss the bigger picture of a microservice system. They shed some light on the topic of *Service boundaries*, *Process management* and *Consistency*.

[11] https://gitlab.com/microservices-for-everyone/

The knowledge from all these chapters combined should help you make good design decisions for your next microservice system. It's then time for some closing remarks in the *Conclusion*.

There's still several hours of reading left before that, so let's start right away.

Acknowledgments

Introduction

We're talking about "microservices" (for everyone), but, as Thomas Ploch[12] reminds me on Twitter:

> "Microservices" is so overloaded, I'd rather hear titles like "Designing digital products for autonomy and scale".

And of course, the little philosopher inside me kicks in too, urging me to explain: what *are* microservices, really? The biggest problem in the literature seems to be that we don't have a firm grasp of the concept. There are no strict rules for how big or small these "micro"-services are supposed to be. How is a microservice architecture different from a service-oriented one? And, when are we supposed to call a system a "microservice system" instead of just a frontend with a backend and some cron jobs?

I'll take it from Sam Newman, author of "Building Microservices" (2015), that:

> Microservices are small, autonomous services that work together.

These are still rather generic terms though. The authors[13] of "Microservice Architecture" (2016) have compared several definitions, as well as many war stories from the field. They've come up with a distilled list of characteristics that all microservices share:

- Small in size
- Messaging enabled
- Bounded by contexts
- Autonomously developed
- Independently deployable
- Decentralized
- Built and released with automated processes

[12] https://twitter.com/tPl0ch/status/832490297443676160

[13] Mike Amundsen, Matt McLarty, Ronnie Mitra, Irakli Nadareishvili.

What is the promise of microservices?

The microservice maturity model

Taking a breath

Modularized microservice architecture

When we discussed the microservice maturity model we learned that that *modularization* is the mandatory first step to achieve a microservice architecture. This modularization step gets broken down into two parts that will bring a greater speed of change, namely:

1. Polyglotism, and
2. Independent deployability

In this chapter we will see how easy it can be to achieve this independent deployability, and with it, polyglotism. This chapter will mainly depend on Docker technology to prove my claim that it is *easy*. Maybe a better word would be *simple*: you'll just need several lines of configuration and a small number of lines-of-code (LOC) to set up two services that can be composed to implement a simple use case. Though Docker and related tools are very user-friendly in my experience, you will find that you have to learn several new concepts, before you will be really successful working with them. Besides, you'll have to think about things you may not usually need to think about, like DNS and service discovery, distribution of data, health checks, restart policies, etc. I've come to like this subject matter, but it doesn't mean it's easy. On top of that, in my experience, every step you take will make you aware of several more things you need to know more about.

Independent deployability & Polyglotism

Introducing Docker Engine

The tool that immediately comes to (my) mind when we're talking about independent deployability, independent manageability, as well as polyglotism (working with different technologies, even programming languages, in different parts of a system) is *Docker*. According to Docker's website[14] (2017), the question "What is Docker?" should be answered like this:

> Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools, system libraries – anything that can be installed on a server. This guarantees that the software will always run the same, regardless of its environment.

[14] https://www.docker.com/what-docker

Managing multiple containers with Docker Compose

Overriding Compose configuration

Environment variables

Volumes

Build configuration

Deploying containers with Docker Machine and Docker Swarm Mode

Setting the stage for a multi-service polyglot deployment

Docker Machine and Docker Compose

A quick project tour

Introducing Docker Swarm

Independent deployability—at last

Rolling updates

Conclusion

Testability and independent manageability

Improving the safety of change with Continuous Delivery

In the previous chapter we've enjoyed the fact that it's very easy to define a new service, build a container image for it, and push it to a server, if not a cluster. This can give us tremendous speed gains during development. Releasing a new version of a service can really be a matter of seconds.

While we were happily creating, updating and scaling services using Docker Swarm, something might have felt a bit off though. As you know from experience, deploying applications always comes with a certain risk:

- You could break existing, expected behaviors of the system.
- You could introduce new, unexpected behaviors to the system.
- You could corrupt data because of a schema design mistake or a failed data migration.
- You could introduce a performance problem, which may eventually cause the system to become unresponsive.

Continuous delivery with Docker in a microservice architecture

An example of a build pipeline for one microservice

Running the unit tests

Building the `service` image

Running the service tests

What else do we need in a build pipeline?

End-to-end tests

Conclusion

Cohesive microservice architecture

> No man is an island, entire of itself… any man's death diminishes me, because I am involved in mankind;
>
> — John Donne[15] (1624)

In today's software landscape most applications are in some way connected to other applications. For example, an application:

- uses another application as its backend, serving data it receives from it,
- automatically shares published blog posts on Twitter,
- offers single-sign-on functionality by means of an OpenID integration, etc.

What is true for a monolithic application is most certainly true for a microservices system. Microservices need each other to do any meaningful work. *No microservice is an island*; if any service crashes, its effect should be somehow noticeable in the emergent behavior of the entire system.

However, we know from the microservices literature (as well as popular statements on social media) that microservices should be *independent* of each other too. They should be independently deployable and they should be resilient: when another service breaks, it shouldn't bring down other services with it.

Traditionally the topic of how applications communicate with each other, and how they can retrieve the data they need to do their work, has been called "application

[15] http://www.phrases.org.uk/meanings/no-man-is-an-island.html

integration". The problem I just described could therefore be considered the *paradox of microservice integration*: microservices work together, but independently so.

In this part I'll explain some of the architectural options you have to resolve this paradox. First, we'll look at different communication styles and how you can make services interact in ways that keep them relatively independent. Then we'll take a look at data, and how you can make sure that it ends up inside the services that need it. Good communication styles and data management will lead to cohesive microservices that can work together, *independently*.

Communication styles

The seminal work on application integration is the book “Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions” by Gregor Hohpe and Bobby Woolf. It’s mainly a catalogue of implementation patterns you can rely on when you’re integrating applications. I wouldn’t recommend reading it cover-to-cover, but the first few and final chapters are worth reading, because they give a useful overview of the topic. A lot of this book’s advice can be copied one-on-one to a microservices world.

Integration requirements

Integration styles

File transfer

Shared database

Remote procedure invocations, or: service API integration

Messaging integration

Characteristics of integration solutions

Blocking versus non-blocking IO

Synchronous versus asynchronous protocols

Synchronous versus asynchronous integration

Implementation examples

The setup

Example: Synchronous integration, synchronous protocol, blocking IO

Intermediate example: Synchronous integration, mixed non-blocking IO

Example: Synchronous integration, fully non-blocking IO

The need for statelessness

Example: A circuit breaker for synchronous communication

A flaky service

The circuit breaker in action

Limitations

Example: Asynchronous integration, asynchronous protocol, blocking IO

Setting up RabbitMQ

The producer

Data and data formats

A suitable integration style plays an important part in the success of your microservices system. Equally important is the data and the format of the data that services will send to each other. Communication protocols like HTTP for synchronous web requests and AMQP for asynchronous message passing typically leave it up to the implementer to choose a format. This leaves you with a lot of options, from the more familiar JSON or XML, with or without a schema definition, to special-purpose formats like Protocol Buffers or Thrift.

The general pattern is: data needs to be *serialized* in order to be sent across the network. At the receiving end the data needs to be *deserialized* before it can be used to interact with domain objects (creating new objects, modifying existing ones, etc.). This (de)serialization task contains several sub-tasks:

1. Converting the raw (textual or binary) data to an intermediate data structure,
2. Validating the structure and basic types of the raw data while doing so, and
3. Using the intermediate data structure to interact with rich domain objects, or create them.

Introducing the example

Schema-less (de)serialization

Custom (de)serialization

Schema validation

Upcasting

Generated (de)serializers

Validation of values

Persistence

In the previous chapter we've looked at data formats, and mainly how we can deal with serializing data that gets sent to a service. These techniques will become relevant once again when we arrive at the next chapter about *Data locality*. First we continue discussing the journey of data *flowing into* the service.

Data gets sent to a service as, for example, the body of an HTTP request, or the content of an AMQP message. Once the service has converted the serialized data into something it can handle—a custom data structure often called "DTO"—it will usually interpret that data as a *request for change*. This is why these DTOs are often called "command objects" or simply "commands".

A request for change here means that the application gets asked to modify its *internal state*. After a command has been processed, the application's state will be different, and consequently its observable behavior will be different too. As an example: before someone has signed up, it won't be possible for them to log in to a service. After the application processes a "sign up" command, the state of the application will have been modified, enabling the user to log in successfully, the next time they try.

It's not enough for an application's state and behavior to change. An application can always clean up its memory, or shut down and restart entirely. When this happens, it needs to make sure that its observable behavior isn't different compared to how it was before. In other words: the changes it makes to its state should be made *persistent*.

Mutable state

Anemic domain model

Rich domain model

Persisting mutable-state objects

Observations

Data loss

Identity generation

CRUD

Event sourcing

Event-sourced object design

Event-sourced persistence

Conclusion

Data locality

In the previous chapter we considered the two main strategies for services to persist their data: storing only the current state of its objects, or storing all the events *leading up to* the current state of its objects. Even when you picked the right strategy for a service, that service won't always be completely self-sufficient and stand-alone with regard to its data. A service often needs other services for more data, to update or enrich its own data.

Within a microservice architecture there is always the question about *data locality*—a term borrowed[16] from the domain of "big data": how can we bring the data as close to the processing unit as possible? Related questions are: how far away is the data? How easily can it be retrieved? Do we even need to fetch it, or can it be sent to us?

[16] https://www.quora.com/What-does-the-term-data-locality-mean-in-Hadoop

Fetching information when needed

REST

Caching HTTP responses

Client-side cache

Network-level cache

GraphQL

Get notified about new information

CQRS part I

Example 1: Filters

Example 2: Accumulations

Alternative setups

Notifications as RESTful resources

Streams

When you don’t want to send notifications

Conclusion

Service boundaries

Boundaries

In previous chapters we've discussed many of the technical aspects of service cohesion. To improve service cohesion, all the techniques we covered were aimed at decoupling services: making them work together, while not knowing too much about each other. Decoupling could take place at the communication level, by switching from synchronous to asynchronous communication. It could also occur at the data level, where parts of the data are specifically owned by one service, then distributed (ideally using domain events) to other services. By defining specific integration points and message schemas, services can provide proper encapsulation for their own internal data, including the schema for that data.

By using all these techniques, we've basically been enforcing service *boundaries*. These boundaries are conceptually much like boundaries at a package or module level. It shouldn't be a big surprise that some of the *principles of package design* (see also my book about this topic[17]) are applicable to services too. With package design we look for small, cohesive, stable, separately reusable and releasable packages. With packages though, there's always the option to use code from another package, even though you didn't intend it to be used that way. With services, it won't be possible to simply use another service's code, because of the *physical* boundaries of the service: the code lives in a different process, maybe on a different machine, and it can only be used by a different service if the service offers an explicit API endpoint for it.

This clean separation between services is both the gift and the curse of a microservice architecture. If laid out well, boundaries will provide services and their development teams with great autonomy. It will enable them to move fast, not affected by bottlenecks in other teams. If the boundaries are in the wrong place though, implementing a feature will require changes across many services, and this will

[17] https://leanpub.com/principles-of-package-design/

slow down the entire organization. Even worse, if releasing such a multi-service-spanning feature means a *coordinated deployment* of multiple services, you'll end up with an even more dangerous bottleneck. This will take away a team's potential for independent deployability and continuous delivery. They'll be back at "level 0" of the maturity model and it will take a lot of effort to climb up the ladder again.

Monolith first?

Bounded contexts

Finding service boundaries

Subdomains

Existing departments

Existing applications

Process stages

Team throughput

Independence requires alignment

Conclusion

Process management

Our journey so far has led us to consider many strategies for decoupling services. In the previous chapter we considered the question how you can determine the dimensions of a service: how big it should be and where its boundaries should lie. Getting those boundaries right might take some time, but eventually you'll end up with several independent services that communicate (mainly) indirectly, using an asynchronous integration style. In order to do this, you let services publish events, to which other services can respond, for example by updating their own read models. These outgoing events often correspond to actual domain events that occur inside the aggregates that are managed by those services.

All of this will get us quite high on the maturity model ladder: finding the right service boundaries will automatically make services *modularized*, as well as *cohesive*. Letting them talk only indirectly will keep them decoupled, and therefore easier to *deploy independently* and *manage separately*. The system will be *composable*, since we can easily rearrange services, make them subscribe to other events, or let them publish new types of events. It's relatively easy to build new functionality and hook new services into the system, tapping into the streams of messages that are already flowing between services.

Things that transcend service boundaries

Process managers

Conclusion

Consistency

State changes

Let's take another look at the *Command-Query Separation* (CQS) principle. We discussed it in a previous chapter already: a method is either a *command* method or a *query* method. Calling a *command* method (often) has an *observable side effect*. This means that after calling such a method, from that moment on the behavior of the application will have changed in some observable way. Maybe a *query* starts returning different data, maybe some other behavior will be different. Consider again the aggregates `Order` and `SeatsAvailability` which we've encountered in the previous chapter. Calling methods on them produced state changes (as well as new domain events). Afterwards, the application didn't behave the same way as it did before.

Zooming out a bit, we notice that a command method on an aggregate usually gets called when the application itself receives a command *message* (for example an HTTP `POST` request). This way, the application as a whole resembles an object: they both expose a programming interface which allows clients to manipulate their internal state. Both applications and objects should encapsulate these state changes in such a way that the result of processing a command doesn't bring the application (nor the objects) in an invalid, incomplete or inconsistent state.

Transactions

Aggregate design

Several ways to work with events

Dispatch events internally or always publish them too?

Eventual consistency

Conclusion—Programming in the fourth dimension

Conclusion

With everything this book has covered so far, you should have some solid ideas on how to design your (micro)services in an autonomous fashion. I'm sure there will remain many challenges for you to run into. There will be technological and organizational impediments. There will be new tools and breaking changes to existing tools. You may have to put in some extra effort to train everybody on the team. And you should look into standardizing the development workflow across teams, which may take a considerable amount of your energy too. Then, you may also get the design of your service boundaries wrong, and you may end up with a lot of extra work. Also, what happens when a service goes down? Does the system as a whole go down?

Opposition

Consider building a monolith

Opposites and trade-offs

Closing words