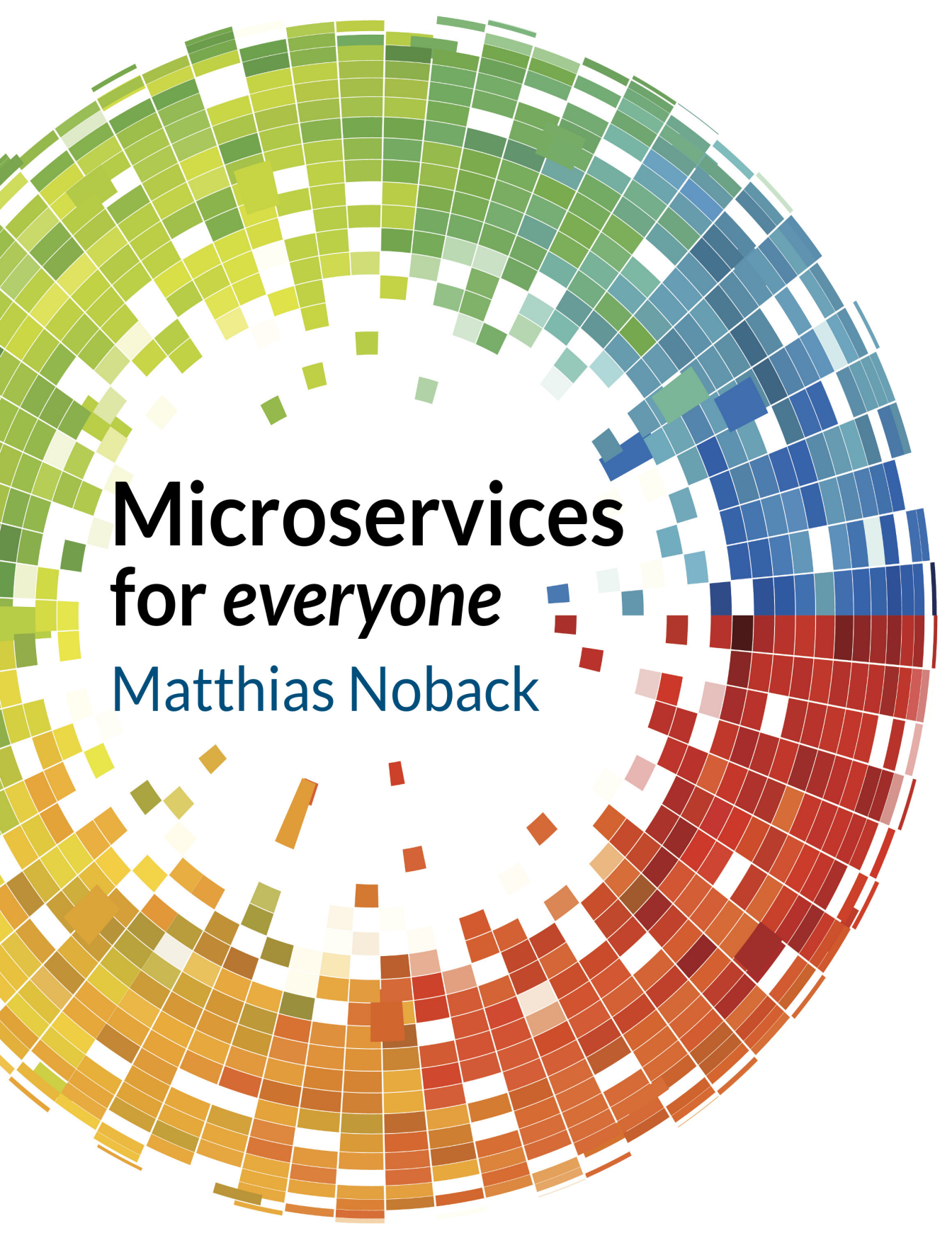

Microservices
for everyone
Matthias Noback

Microservices for everyone

Matthias Noback

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

This version was published on 2017-03-10

Contents

Introduction

Scepticism

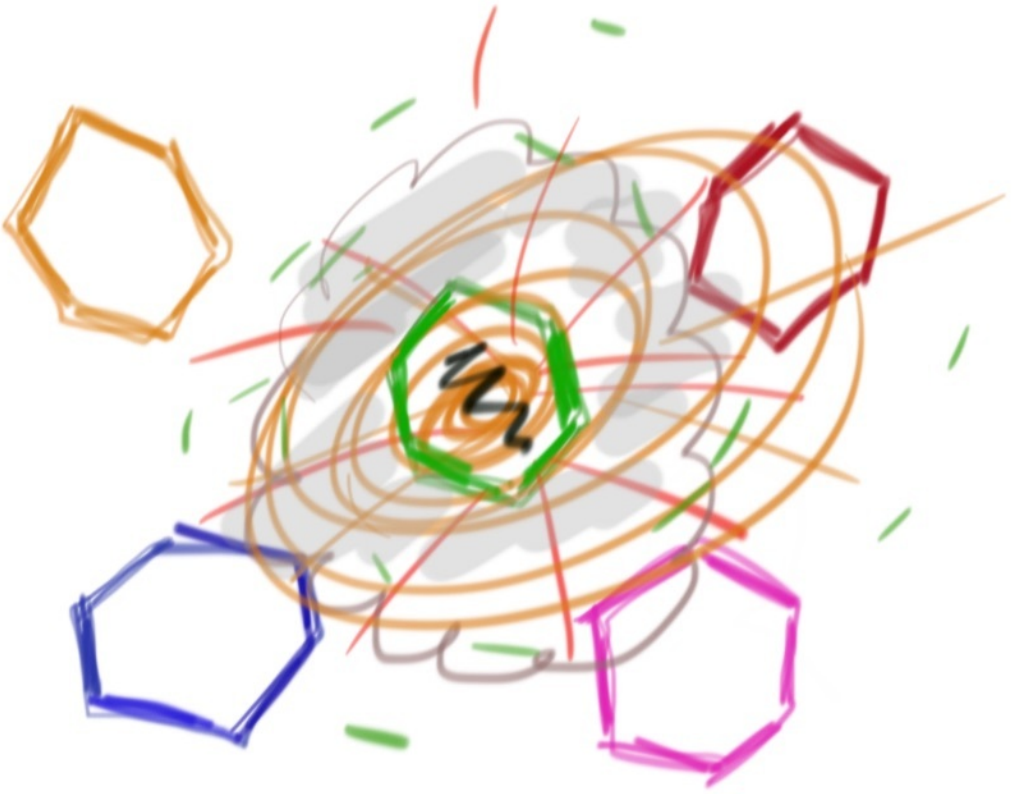
I can almost hear you think: "Bah, microservices. Nothing good could come from that!"…

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

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 company. Especially since you already 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]

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



Which makes me wonder: how does this differ from when we have a single application? If something goes wrong in a monolith we usually throw an exception, and let it crash the application, 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]

[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]

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. And 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 have decided to make their next project a microservices project, which took a lot of research and a lot of work and the project may have ended up in quite a bad shape after all. There are many reasons for this. Probably most of those reasons are the same as for any other kind of software project: the regular problems 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 microservices 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

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

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:

- 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 connected.
- 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 on Deliberate Discovery[6]), 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 the 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 or devops.

[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/

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 Europe local communities 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.

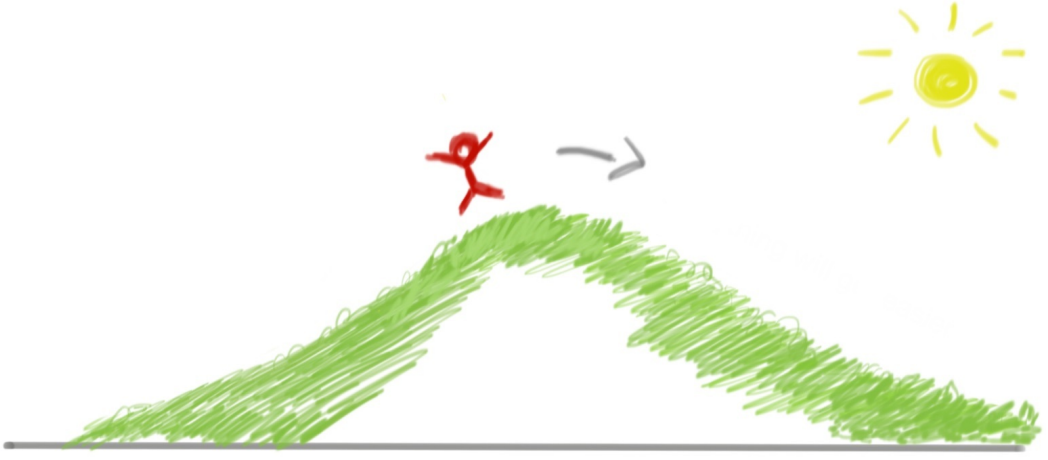
Why I have to write this book

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

This book is not so much about showing you in overly enthusiastic words that you're crazy if you don't do microservices. It's about my hypothesis that over the past few years the tech community has been working their way towards the peak of *Microservice Impediments*. I want to prove 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 microservices.*



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 find 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 stop writing.

Since I want to combine insights from Domain-Driven Design (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 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 every tech book, this book will 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, add a list of things to consider once you really start implementing microservices.

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

> Code samples will be written in PHP.

If you know Java: PHP is much like Java anyway, 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 they work with a language that may not be so well-designed; you can still do great things with it.

Rigor?

If you have read my previous book, Principles of Package Design[10], 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: 1. I don'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 *operations*. 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 will automatically fly back to me, the moment I publish something. I'm slowly, but surely, accepting 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*. 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

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

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 it:

> 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 you 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 you on a trail to some place else, or provide you 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.

Overview of the contents

The order in which I'm going to discuss all relevant topics is not certain yet, but this is a list of some of the keywords for this book:

Exploring the domain, event storming, bounded contexts, services, Docker, continuous delivery, CRUD, CQRS, event sourcing, shared database, synchronous HTTP calls, asynchronous messaging, ports & adapters, user interface, storage, serialization.

To be continued

For now, this is all that's available of *Microservices for everyone*. If you like it, please let me know.

- Tweet me at @matthiasnoback[11]
- Send an email[12]
- Discuss this book online[13]

[11] https://twitter.com/matthiasnoback
[12] https://leanpub.com/microservices-for-everyone/feedback
[13] https://leanpub.com/microservices-for-everyone/feedback

What are we talking about?

What is the promise of microservices?

The microservice maturity model

Taking a breath

Modularized Microservices Architecture

Independent deployability & Polyglotism

Introducing Docker Engine

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

Conclusion

Testability and independent manageability

Improving the safety of change with Continuous Delivery

Continuous delivery with Docker in a microservices 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 Microservices Architecture

Communication styles

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, synchronous protocol, mixed non-blocking IO

Example: Synchronous integration, synchronous protocol, 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, non-blocking IO