Introducing Distributed Application Runtime (Dapr)

Simplifying Microservices Applications Development Through Proven and Reusable Patterns and Practices

—

Radoslav Gatev

Foreword by Yaron Schneider, Principal Software Engineer and Dapr co-founder, Microsoft

Apress®

Introducing Distributed Application Runtime (Dapr)

Simplifying Microservices Applications Development Through Proven and Reusable Patterns and Practices

Radoslav Gatev
Foreword by Yaron Schneider,
Principal Software Engineer and Dapr co-founder, Microsoft

Apress®

Introducing Distributed Application Runtime (Dapr): Simplifying Microservices Applications Development Through Proven and Reusable Patterns and Practices

Radoslav Gatev
Gorna Oryahovitsa, Bulgaria

ISBN-13 (pbk): 978-1-4842-6997-8 ISBN-13 (electronic): 978-1-4842-6998-5
https://doi.org/10.1007/978-1-4842-6998-5

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Joan Murray
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media LLC, 1 New York Plaza, Suite 4600, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484269978. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To my girlfriend Desislava who supports me unconditionally.

Table of Contents

About the Author

Radoslav Gatev is a software architect and consultant who specializes in designing and building complex and vast solutions in Microsoft Azure. He helps companies all over the world, ranging from startups to big enterprises, to have highly performant and resilient applications that utilize the cloud in the best and most efficient way possible. Radoslav has been awarded a Microsoft Most Valuable Professional (MVP) for Microsoft Azure for his ongoing contributions to the community in this area. He strives for excellence and enjoyment when working on the bleeding edge of technology and is excited to work with Dapr. He frequently speaks and presents at various conferences and participates in organizing multiple technical conferences in Bulgaria.

About the Technical Reviewer

As a freelance Microsoft technologies expert, **Kris van der Mast** helps his clients to reach their goals. Actively involved in the global community, he is a Microsoft MVP since 2007 for ASP.NET and since 2016 for two disciplines: Azure and Visual Studio and Development Technologies. Kris is also a Microsoft ASP Insider, Microsoft Azure Advisor, aOS ambassador, and Belgian Microsoft Extended Experts Team (MEET) member. In the Belgian community, Kris is active as a board member of the Belgian Azure User Group (AZUG) and is Chairman of the Belgian User Group (BUG) Initiative. Since he started with .NET back in 2002, he's also been active on the ASP.NET forums where he is also a moderator. His personal site can be found at `www.krisvandermast.com`. Kris is a public (inter) national speaker and is a co-organizer of the CloudBrew conference.

Personal note:

While doing a book review, I also like to learn new things on the go. With this book I sure did. I hope you will enjoy reading it at least as much as I did.

Foreword

In the year leading up to the first release of Dapr as an open source project in October 2019, Haishi Bai (my partner in co-founding Dapr) and I observed just how much the cloud-native space had matured. It had grown to provide ops and infrastructure teams with first-class tools to run their workloads either on premises or in the cloud.

With the rise of Kubernetes (K8s), an entire ecosystem of platforms has sprung up to provide the missing pieces for network security, traffic routing, monitoring, volume management, and more.

Yet, something was missing.

The mission statement to make infrastructure "boring" was being realized, but for developers, many if not all of the age-old challenges around distributed computing continued to exist in cloud-native platforms, especially in microservice workloads where complexity grows with each service added.

This is where Dapr comes in. First and foremost a developer-facing tool, Dapr focuses on solving distributed systems challenges for cloud-native developers. But just like any new technology, it's critical to be able to understand its uses, features, and capabilities.

This book by Radoslav Gatev is the authoritative, technical, hands-on resource you need to learn Dapr from the ground up. Up to date with version 1.0 of Dapr, this book gives you all you need to know about the Dapr building blocks and APIs (Application Programming Interfaces), when and how to use them, and includes samples in multiple languages to get you started quickly. In addition to the Dapr APIs, you'll also find important information about how to debug Dapr-enabled applications, which is critical to running Dapr in production.

Radoslav has extensive, in-depth knowledge of Dapr and is an active Dapr contributor, participating in the Dapr community and helping others learn to use it as well. He makes the project better by working with maintainers to report issues and contribute content.

You really can't go wrong with this book, and I highly recommend it to anyone who wants to start developing applications with Dapr.

Yaron Schneider

Principal Software Engineer and Dapr Co-founder, Microsoft

Acknowledgments

In every venture uncommon to a self, there should be a great catalyst. I would like to thank Apress and especially Joan Murray, Jill Balzano, Laura Berendson, Welmoed Spahr, and everyone else involved in the publishing of this book. I had been thinking about writing a book for quite some time, and I am grateful that Joan reached out to me. At that moment I had a few conferences canceled, a few professional opportunities lost because of the risks and the great uncertainty at the start of COVID-19. Fast-forward a year from then, the book has been finished, and I am writing this. A year of lockdowns spent in writing is a good year, after all.

Additionally, I would like to thank Mark Russinovich for being such an inspiration and knowledge source for me. Sometimes, it takes just a tweet to change the life of a person. He retweeted a blog post of mine about Dapr. It gained a lot of attention, and to a large extent, because of that, *Introducing Distributed Application Runtime (Dapr)* is now a reality. I would like to thank Yaron Schneider and all Dapr maintainers who are always friendly and supportive. They helped a lot by answering some of the questions I've had in the process.

I would also like to thank Kris van der Mast, the technical reviewer, for the excellent feedback and suggestions that added immense value to this book.

I would like to thank Mihail Mateev who gave me the opportunity to do my first public session a couple of years ago. Since then, we have been collaborating with a lot of other folks to make some of the biggest conferences in Bulgaria possible. Of course, thanks to the community that still finds them interesting, and from the fascinating discussions, they sparkle. I would like to thank Martin Tatar, Cristina González Herrero, and Irene Otero for their great help and continuous support to us, the Microsoft Most Valuable Professionals.

I would like to thank Dimitar Mazhlekov with whom we have been friends, teammates, business partners, and tech junkies. We have walked a long way and learned a lot together.

I would like to thank all my teachers, professors, and mentors who supported me a lot throughout the years. To find a good teacher is a matter of luck. And with you all, I am the lucky person for being your student.

I would like to express my gratitude for being able to work with organizations around the globe that gave me exposure to their unique and intriguing challenges that helped me gain so much knowledge and experience. Thanks to all the team members I met there and for what I was able to learn from every one of them.

And last but not least, to my girlfriend Desislava, my parents, extended family, and friends, thank you for the endless support throughout the years! Thank you for keeping me sane and forgiving my absence when I get to work on something challenging.

Introduction

Being able to work on various projects, one should be able to identify the common set of issues every project faces. It doesn't mean you can always apply the same solution over and over again, but it puts a good structure. I was very lucky that early in my career, I was pointed to the proper things to learn. I have already been using object-oriented programming (OOP) for some time, but I was stunned when I read the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides for the first time. It gave me the answers to some of the questions I'd been asking myself. From there on, I am a strong believer that patterns do not only serve as reusable solutions to common problems, they become a common lingo and teach you how to think in an abstract way. Being able to work at a conceptual level, instead of focusing too much on the details, I believe made me a better professional.

I heard of Distributed Application Runtime (Dapr) for the first time when it was announced at Microsoft Ignite 2019, Microsoft's annual conference for developers and IT professionals. At first, the idea of it resonated within me but I didn't completely understand it, and so I decided to start playing with it. In September 2020, a transition to an open governance model was announced to ensure that the project is open and vendor neutral. Fast-forward to February 2021 when Dapr v1.0 was released. Now that Dapr is stable and production-ready, it is also in the process of being donated to the Cloud Native Computing Foundation (CNCF) as an Incubation project. By the time you read this, it may be finalized.

Dapr greatly simplifies the development of Microservices applications. It works with any language and any platform. You can containerize your applications or not, you can use Kubernetes or not, you can deploy to the cloud or not. You can sense the freedom here. From a development perspective, Dapr offers a number of capabilities grouped and packaged as building blocks. Let's face it. You will have to use some services that are external to the application you are aiming to build. It is very normal to not try to reinvent the wheel and build everything from scratch. By using the building blocks provided by Dapr, you use those external services without thinking about any SDKs or specific concepts imposed by the external service you are trying to integrate with. You just have to know how to work with the building block. This simplifies the operations you want to

execute on the target external services, and Dapr serves as the common denominator. That's why you can swap one technology with another in the scope of the building block, that is, reconfiguring Dapr from persisting state to, say, Redis to MySQL, for example.

Some believe Dapr is the service mesh but done right. The reason for that is that service meshes rely on the sidecar architecture as Dapr does. However, service meshes are for network infrastructure, while Dapr provides reusable patterns that are easy to apply and repeatable. In the future, I expect building blocks to expand in functionality and maybe new building blocks to come to Dapr. With that, the reach to potential external services will become so wide. For greenfield projects, this will mean that Dapr can be put on the foundational level of decisions. Once you have it, you can, later on, decide what specific message broker or what specific persistence medium to use for state storage, for example. This level of freedom unlocks many opportunities.

Introducing Distributed Application Runtime (Dapr) aims to be your guide to learning Dapr and using it for the first time. Some previous experience building distributed systems will be helpful but is by no means required. The book is divided into three parts. In the first part before diving into Dapr, a chapter is devoted to set the ground for the basic concepts of Microservices applications. The following chapter introduces Dapr: how it works and how to initialize and run it locally. The next chapter covers the basics of containers and Kubernetes. Then all that knowledge is combined in order to explore how Dapr works inside Kubernetes. The part wraps up by exploring the various options to develop and debug Dapr applications, by leveraging the proper Visual Studio code extensions – both locally and inside Kubernetes.

The second part of the book has a chapter devoted to each building block that explores it in detail. The building blocks are:

- Service Invocation
- Publish and Subscribe
- State Management
- Resource Bindings
- The Actor model
- Secrets
- Observability

The final part of the book is about integrating Dapr with other technologies. The first chapter outlines what middleware can be plugged into the request pipeline of Dapr. Some of the middleware enable using protocols like the OAuth2 Client Credentials and Authentication Code grants and OpenID Connect with various Identity Providers that support them. The examples in the chapter use Azure Active Directory. The following chapter discusses how to use Dapr with ASP.NET Core by leveraging the useful attributes that come from the Dapr .NET SDK. The last two chapters cover how to combine Dapr with the runtimes of Azure Functions and Azure Logic Apps.

Code samples accompany almost every chapter of the book. Most of them are implemented in C#, but there are a few of them in Node.js, to emphasize the multiple-language approach to microservices. You can find them at https://github.com/Apress/introducing-dapr. You will need to have .NET and Node.js installed. Some of the tips and tricks in the book are applicable only to Visual Studio Code (e.g., the several extensions that are covered in Chapter 5: Debugging Dapr Applications), but you can also use any code editor or IDE like Visual Studio. For some of the examples, you will also need Docker on your machine and any Kubernetes cluster – either locally, as part of Docker Desktop, or somewhere in the cloud.

I hope you enjoy the book. Good luck on your learning journey. Let's start Dapr-izing! I am happy to connect with you on social media:

LinkedIn: www.linkedin.com/in/radoslavgatev/

Twitter: https://twitter.com/RadoslavGatev

Feel free to also check my blog: www.gatevnotes.com.

PART I

Getting Started

CHAPTER 1

Introduction to Microservices

Digitalization drives businesses in such a direction that every system should be resilient and available all the time. In order to achieve that, you have to make certain decisions about the application architecture. In this chapter, you will learn how systems evolved from calculation machines built to serve a specific purpose to general-purpose computers. Making the same parallel but on the software side, we will discuss what Monolithic applications are along with their pros and cons. Then, we will go through the need of having distributed applications dispersed across a network of computers. You will also learn about the Microservices architecture as a popular way for building distributed applications – how to design such applications, what challenges the Microservices architecture brings, and some of the applicable patterns that are often used.

A Brief History of System Design

It's always good to take a look back at history to understand how the concepts have evolved over time making almost every innovation by building on the existing knowledge and experience. There has been a long way for computers until you could walk with a computing device in the pocket or on the wrist.

The first computing devices were entirely mechanical,[1] operating with components like levers, belts, gears, and so on to perform their logic. That's a lot of moving parts that

[1]The first mechanical computer is considered to be the Difference Engine that was designed by Charles Babbage in the 1820s for calculating and tabulating the values of polynomial functions. Later on, he devised another machine called the Analytical Engine that aimed to perform general-purpose computation. The concepts it was to employ can be found in modern computers, although it was designed to be entirely mechanical.

R. Gatev, *Introducing Distributed Application Runtime (Dapr)*, https://doi.org/10.1007/978-1-4842-6998-5_1

take a lot of space. Apart from the slowness of their operation, each of them was a point of failure on its own. But over time, the mechanical parts started getting replaced by their electric counterparts.

Hardware Progress

To perform some logic, you need some way of representing state – like 1 and 0 in the modern binary computers. Likewise, the relay was identified as a viable component that was widely known and available to be utilized for performing the "on" and "off" switching. But still, relays were rather slow as they had a moving mechanical part – an electromagnet opens or closes a metal contact between two conductors. Then *vacuum tubes* gained traction as a way of switching. They didn't have any moving parts; however, they were still big, expensive, and nonefficient. Then they got replaced by *transistors*. But imagine what is soldering thousands of discrete transistors in a complex circuit! There will be faulty wirings that are hard to discover. Later on, the need for soldering discrete transistors was avoided with *integrated circuits* where thousands of tiny transistors were placed on small chips. This ultimately led the way to modern *microprocessor* technology. These evolutionary steps were restricted by the speed, size, and cost of a single bit.

Early computers used to be expensive and could easily fill an entire room. Because they weighed a lot, they were usually moved around using forklifts and transported via cargo airplanes. Announced in 1956, IBM 305 RAMAC was the first computer to use a random-access disk drive – the IBM 350 Disk Storage Unit that incorporated 50 24-inch-diameter rotating disks that could store 5 million 6-bit characters or the equivalent of whopping 3.75 MB of data. This is the ancestor of every hard drive produced ever since.

That's how typically technology evolves. Advances of knowledge are being used to build new things on top of the old knowledge base. Every decision at a time is restricted by various boundaries we face – physical limits, cost, speed, size, purpose, and so on. If you think about the purpose of the early computers, in the beginning, they were devised with a sole purpose – from solving polynomial functions to cracking secret codes ciphered by machines such as the German Enigma machine. There was a long way until we could use general-purpose computers that are highly programmable.

Without making any generalizations, it will be highly inconvenient to build anything. You have to manage all of the moving parts right from the beginning, which is a lot of effort. For example, you had to either be a genius or be among one of the inventors to be

able to use the computer in the 1950s. With the introduction of personal computers, it started to get easier. The same applies to software development.

Software Progress

While hardware tried to address the physical aspects of computer systems, kind of the same evolution happened with software. The early programs were highly dependent on the architecture of the computer executing them.

Applications Development

When writing low-level code, you have to think about everything – machine instructions and what registers to use and how. With the advances of modern compilers, we have a comfortable abstraction to express just what the program should do without thinking about what instructions will be executed by the *Central Processing Unit* (CPU).

Programs used to be a self-sustainable piece of code without any external dependencies. *Object-oriented programming* is a paradigm that essentially gave us yet another powerful abstraction. By utilizing the power of interfaces, we can reuse a lot of code. Every piece of code that we use out of the box has an interface (or a contract that it serves). Software libraries emerged, and they became the building blocks of modern software. Let's face it: system software, server software, frameworks, utilities, and all kinds of application software are all built using well-known and widely used libraries and components.

Some programs are still dependent on some operating system features or third-party components that were installed on the developer's machine. And the typical case is that the program you just downloaded does not run on your machine. "But it works on my machine," they would say.

A few years ago, containers started to gain more and more popularity. Containers are the solution to make your code easily transferable across machines and environments by packaging all application code along with all of its dependencies. By doing this, you are effectively isolating the host machines and their current state from your code.

Infrastructure and Scalability

In the past, programs were running on a single machine and were used only on that same machine. This was until computers could be connected to networks where they could talk to each other. The machine that hosts applications is called a server, and the other machines that use the applications are called clients.

Over time what happened with some applications is that the number of clients started to outgrow the capacity of the servers. To accommodate the ever-increasing load resulted in adding more resources to each server, the so-called *vertical scaling*. The application code is still the same but running on a beefier machine with a lot more processing power and memory. At some point, the technological limits will be reached, or it will become too expensive to continue adding power to a single machine. And in case something happens with this machine, your application will become inaccessible for all clients. The number of options to alleviate the issue was pretty much exhausted.

Then it became obvious that the applications should be distributed across different servers. There are more instances of your applications running across a set of machines instead of relying on a single big machine. The incoming load is typically distributed across all machines. That's called *horizontal scaling*.

To be able to achieve horizontal scaling, you have to make sure that every instance of the application doesn't hold any internal state, that is, your application is *stateless*. This is needed because each application replica should be able to respond to any request. As you replicate software across more instances, you are starting to treat your servers more like cattle as opposed to much-loved pets. You don't really care even if you lose an entire server if you have a couple of others that are still healthy and taking traffic. Of course, as with any decision, this comes with certain trade-offs.

Even if you achieve some level of scalability, it doesn't mean that your application is prepared to withstand future requirements. Not only traffic can grow but also the application functionality evolves and extends. Respectively with functionality, team size is also a subject of change.

In the next sections of this chapter, I will walk you through the two popular architectural styles for building an application – the Monolithic and the Microservices architecture. It doesn't make sense to explain one without mentioning the other because they have rather contradictory principles.

Monolithic Architecture

According to the Merriam-Webster dictionary, the definition of the word monolith is "a single great stone often in the form of an obelisk or column" or "a massive structure." Taking the broad meaning of a *massive structure*, a Monolithic application is built as a single unit. This single unit contains all your application logic. Internally, this Monolithic application can consist of different layers. One of the layers could be the presentation

layer, which deals with the user interface of the application; another layer can hold some business logic; a third layer can be used for accessing the database. But it doesn't mean that those layers cannot be separated from one another in terms of a codebase. For example, the business logic layer can spread across numerous small modules, each of them implementing just a small part of the overall functionality. This generally improves the quality of the code; however, those modules are still living in the same layer of the application.

Figure 1-1 shows what typically happens when a user invokes Function A in a Monolithic application. Function A passes the control to Function B, which depends on Functions C and D (which depends on Function E), and when they return a result, Function B will be able to pass the result to the user. Although the functionality is separated into discrete functions, which are likely placed in separate class libraries, they are all sharing common resources like the database and are running inside the same process on the same machine.

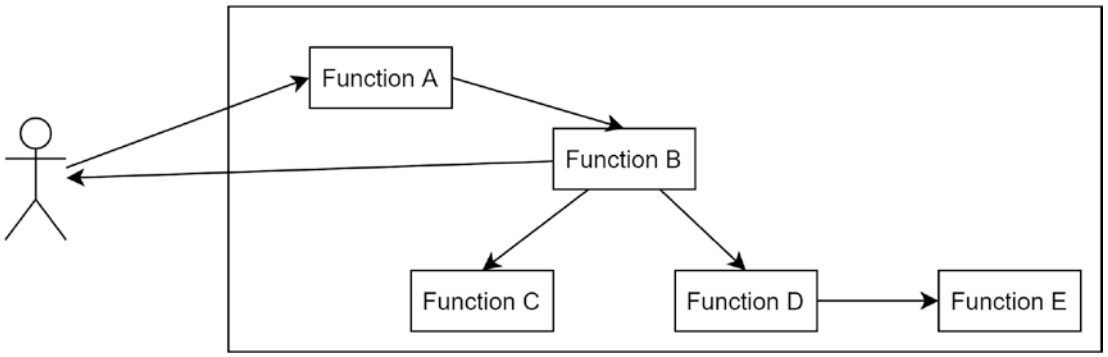

Figure 1-1. *Functions inside a Monolithic application*

Benefits of the Monolithic Architecture

However, "monolith" can be used as defamation nowadays for an application that doesn't follow modern trends that can result in a lot of prolonged discussions. At this point, we have to acknowledge that there are still loads of Monolithic applications out there that are business-critical. Developing monoliths has some benefits as well.

The first thing probably is the simplicity that comes with Monolithic applications. If you think about it, *Integrated Development Environments (IDEs)* and application frameworks are driving you in this direction. It's very easy to create a new web project

and implement its models, views, and controllers. You click the Run button of your IDE and voilà, it is up and running on your machine. Everything can be debugged end to end. It feels really natural and fast. The deployment is also easy – package the application and move it to the server that will host it. Done.

Later on, several other developers can join you and work on this application. And you probably won't have serious conflicts for most of the things you are implementing, as long as the application is not very big and you are just a handful of people.

When your server starts facing pressure from traffic growth, you can probably replicate the application to multiple servers and put a load balancer in front. This way each replica will receive a portion of all requests, and it won't get overwhelmed. This approach is the so-called *horizontal scaling*.

Drawbacks of the Monolithic Architecture

However, as time goes by, your application will have more and more functionality implemented. The code will become more complex with every passing day and every new requirement that is implemented. Although the code could have been separated into different modules, it is very easy to miss the fact there are so many cross-dependencies in place.

This means that the Monolithic architecture is easy in the beginning and becomes trickier over time. The more team members you assign to the project, the slower they work together. The impact of each change that gets implemented is difficult to be understood because of the dependencies. As shown in Figure 1-1, Function A depends on Functions B, C, D, and E, which are interconnected.

Imagine there is a serious performance issue in Function B that makes your CPU and memory go crazy high. This will likely bring down the whole process that is hosting it. But wait. We have several other instances, right? Well, the same piece of code is running in all the replicas, so it means that you are also replicating all performance issues and bugs. Although the issue comes from a tiny function that lives in a small module of your application, it impacts the availability of your whole application.

Let's say that your team fixed the issue in Function B. You cannot just deploy the changes made on Function B. Instead, you will have to package the whole application, deploy it, and wait for it to warm up and start serving again.

What if at some point in time you identify that most of the load in the application goes to a particular feature that the majority of clients use? You start wondering how to

scale out only the pieces of functionality in question to achieve a fine-grained density with the hardware you have at hand.

Or maybe you want to start implementing new features with some new technology that makes sense for the application you have and some of the team members are knowledgeable about it. Let's say it's a different framework in a different language than the ones you have based your Monolithic application on. Unfortunately, utilizing such technology won't be possible as you are locked in a certain execution model that spans the whole application. And generally speaking, attempts for mixing various technologies that bring different concepts don't end well in the long term.

It's a monolith. You cannot easily dismantle it as its components are *tightly coupled*. If you waited too long, it might be that your application is a homogeneous mixture of functions. Let's see what is the case with the popular Microservices architecture.

Microservices Architecture

The Microservices architecture is an architectural pattern for building distributed systems in such a way that they are comprised of different *loosely coupled* components called services that run in different processes and are independently deployed.

Figure 1-2 shows how the Monolithic application you saw earlier in Figure 1-1 can be shaped in the Microservices world. Each of the functions has its own service. The services can be independently deployed and distributed across different machines. But still, they are part of the same application and work as a whole. The services communicate with each other by using a clearly defined *Application Programming Interface* (API) via protocols such as HTTP, gRPC, AMQP, and others.

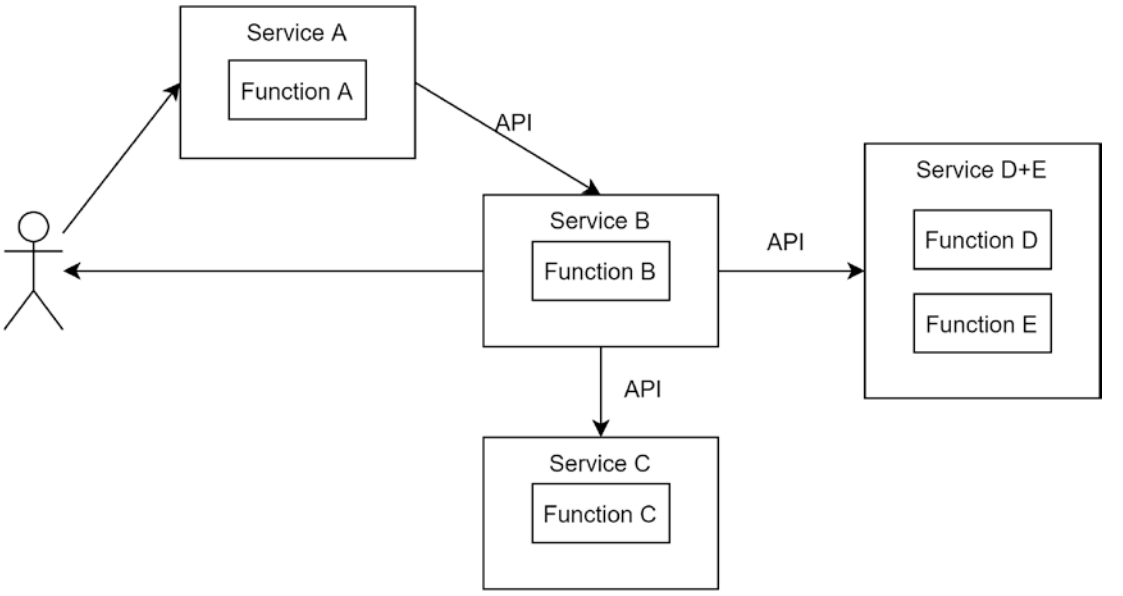

Figure 1-2. *The Monolithic application translated into a Microservices application*

For the sake of the example, I have oversimplified the process of converting a Monolithic application to one based on the Microservices architecture. It's not a straight two-step process. It takes careful analysis and thorough design beforehand.

Designing Microservices

Now that you know that an application should be split into multiple pieces, how do you define how big those pieces are? There are various approaches to tackle this. But designing microservices is pretty much a piece of art. You can start by understanding how the business that you are digitalizing works and what are its processes and rules. And then identify what are the *business capabilities* that the application should provide. You can probably group them into several categories and identify the main objects.

Alternatively, you can use some of the tactics from *Domain-Driven Design*. You can decompose the main domain into subdomains and identify the *Bounded Contexts* we have. This is a strategy to split a large problem into a set of small pieces with clear relationships. Instead of defining a large ubiquitous model that will end up representing a lot of perspectives, the Bounded Contexts enable you to project small parts of the whole domain from different lenses. Having outlined the Bounded Contexts and the

models, it's easier to start thinking about the functionality of the service. Let me give you an example. Let's say we are designing an online store. We may have a Cart microservice, a Payment microservice, and a Shipment microservice, among others. When a customer purchases an item from the store, the underlying work will be performed by those microservices. But each service sees the customer from a different perspective. The Cart microservice sees the user as a Customer who added some product into the cart. Customer is the entity representing the user in the context of managing a cart. To the Payment service, the user is a Payer who uses a specific payment method to remit the money. From the perspective of the Shipment service, the user is an entity named Receiver that contains the user's address. All three entities share the same identifier of a user but have different attributes depending on the problem being addressed.

The goal of identifying the boundaries of a service is not just to make it as small as possible. In the example in Figure 1-2, Function D and Function E are placed in the same microservice as their functions belong to the same business capability. You should not aim to create a microservice for the smallest piece of functionality. But ideally, a microservice should comply with the *Single-Responsibility Principle*.

The first design of your microservices won't stay forever. As the business evolves, you will likely do some refactoring to refine the size and granularity of the services. Getting back to the example with the online store, if you find yourself constantly merging the information about the user from Payment and Shipment services, for example, Payment calls the Shipment service upon every operation, there is a huge chance those two should be merged into one service.

Each service is responsible for storing its data in some persistence layer. Depending on the nature of data, it can be persisted in various types of databases. Some of the data can be cached to offload the performance hit on the services and their respective databases. The potential of using different technologies in each service depending on the case is enormous. In contrast to monoliths, you can choose whatever programming languages, frameworks, or databases that fit best the problem you are solving or the expertise of the team.

Benefits

The drawbacks of Monolithic applications, in general, are addressed by the Microservices architecture.