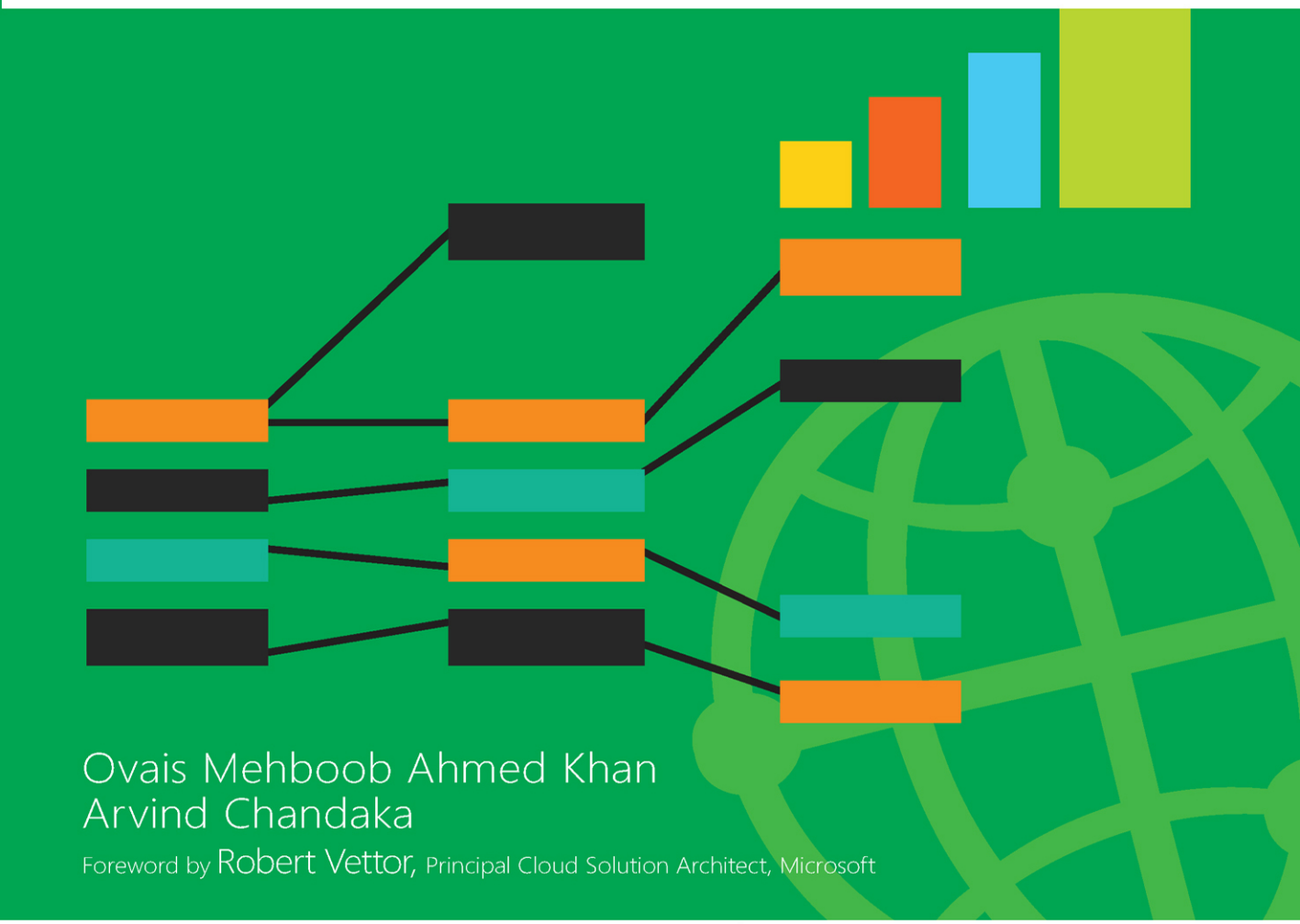
Microsoft

Developing Microservices Architecture on Microsoft Azure with Open Source Technologies



Ovais Mehboob Ahmed Khan
Arvind Chandaka

Foreword by Robert Vettor, Principal Cloud Solution Architect, Microsoft

Microsoft

Developing Microservices Architecture on Microsoft Azure with Open Source Technologies

Ovais Mehboob Ahmed Khan
Arvind Chandaka

Developing Microservices Architecture on Microsoft Azure with Open Source Technologies

Published with the authorization of Microsoft Corporation by:
Pearson Education, Inc.

ISBN-13: 978-0-13-681938-7
ISBN-10: 0-13-681938-9

Library of Congress Control Number: 2021937949

ScoutAutomatedPrintCode

CREDITS

EDITOR-IN-CHIEF
Brett Bartow

EXECUTIVE EDITOR
Loretta Yates

DEVELOPMENT EDITOR
Rick Kughen

SPONSORING EDITOR
Charvi Arora

MANAGING EDITOR
Sandra Schroeder

SENIOR PROJECT EDITOR
Tracey Croom

COPY EDITOR
Rick Kughen

INDEXER
Cheryl Ann Lenser

PROOFREADER
Abigail Manheim

TECHNICAL EDITORS
Doug Holland
Thomas Palathra

EDITORIAL ASSISTANT
Cindy Teeters

COVER DESIGNER
Twist Creative, Seattle

COMPOSITOR
codeMantra

I would like to thank my family for supporting and encouraging me in every goal of my life.

—Ovais Mehboob Ahmed Khan

I dedicate this book to my family and friends for their everlasting encouragement and support.

—Arvind Chandaka

This page intentionally left blank

Contents

Acknowledgments

This book involved great effort on our part. With the recent growth in microservices architecture, we saw the importance of developing a sample case study on microservices to empower our customers to build better applications. We created a use case to illustrate microservices, created a functional application, and evangelized it to our customers—all before even putting pen to paper to write this book. Our friends and family were the spectators during these laborious endeavors, and they showed nothing but encouragement and understanding during long nights and tiring weekends. We couldn't have done it without them!

A big thanks goes to our managers for their everlasting support. Barbaros Gunay, Andrew McCreary, Mekonnen Kassa—thank you! Without your support, this book could not have been done.

When we both started working at Microsoft, we were quickly taken under the OSS Enablement Team's wing. Rick Hines and Bahram Rushenas are the team leads who encouraged us to pursue the project that ultimately became this book. Whether it was discussing a design or architecture of an application, brainstorming ideas, or presenting at events, their help and support were tremendous. We want to thank you for everything you've done.

We also want to thank Sally Brennan for preparing our session outline for Microsoft Ready, our premier go-to-market conference, which led us to further refine the content we used within our book.

We also want to thank our team at Pearson. Thank you so much, Loretta Yates, Charvi Arora, Rick Kughen, Doug Holland, Thomas Palathra, Abigail Manheim, and Tracey Croom for supporting us during our journey. We would like to thank Rob Vettor for lending his time to review our book and providing a foreword. Rob is an extremely experienced individual in microservices at Microsoft and we greatly appreciate his support.

About the authors

Ovais Mehboob Ahmed Khan is a seasoned programmer and solutions architect with nearly 20 years of experience in software development, consultancy, and solution architecture. He has worked with various clients across the United States, Europe, Middle East, and Africa, and he currently works as a Senior Customer Engineer at Microsoft, based in Dubai. He specializes mainly in application development using .NET and other OSS technologies, Microsoft Azure, and DevOps.

He is a prolific writer and has published several books on enterprise application architecture, .NET Core, VS Code, and JavaScript, and he has written numerous technical articles on various sites. He likes to talk about technology and has delivered various technical sessions around the world.

Arvind Chandaka is a product manager at Microsoft and has led products across Azure and Cloud + AI Engineering. He has worked with strategy executive teams to grow Microsoft Azure as well as built IP at the company resulting in several services and products. He is a recognized SME in enterprise technology and has advised many Fortune 100 companies and international clients around the world. He specializes across infrastructure, identity, cyber security, open source, and more. A strong believer in empowerment, he works closely with startups, is a founder himself, invests in the entrepreneurial ecosystem, and serves on the board of non-profits in New York City.

Arvind earned his Bachelor of Science at Cornell University where he studied several disciplines ranging from computer science to business. In his spare time, you will likely find him at a tennis court or traveling the world for the next great foodie destination.

Foreword

For many years, the guidance for building business applications was clear-cut:

- Create a large, monolithic core that contains business logic
- Persist data to a shared relational database
- Expose functionality through front-end UIs and API endpoints
- Build, package, and deploy the application, and you're in business.

While often challenging, such monolithic applications were *straightforward* to build, test, deploy, and troubleshoot. With code executing in a single process, performance could be good. When the need arose, you could scale the application up (vertically) by adding more server resources.

This model, however, had limitations. When an application proved popular, the monolithic architecture would eventually break down. As the app grew larger, more complex, and more "coupled," it became less agile:

- Feature updates and fixes would cause unintended and costly side effects, such as breaking existing functionality.
- One unstable component could crash the entire system.
- Scaling any component required scaling the entire application.
- New technologies and frameworks were not an option.
- Each change required a full deployment of the application.

Perhaps the biggest challenge was *agility*. Monolithic applications prevented the business from moving fast.

Fast-forward into the world of modern apps and microservices. They are all about speed and agility. By decomposing the business domain into independent services, microservices enable business systems to strategically align with business capabilities. They embrace feature releases with high confidence. They allow for updating small areas of a live application without downtime. They empower the business to adapt rapidly to evolving market and competitive forces.

While microservice adoption is rapidly increasing, not everybody is adopting it correctly. Implementing a microservices architecture requires substantial investment. For it to be successful, developers and architects must understand the principles, patterns, and best practices of distributed microservices architecture.

This book *spoon-feeds* microservices to you. From design to development and from deployment to operation, it provides a comprehensive education into the world of microservices. Moreover, the book wraps microservice construction into

the context of cloud native design and open-source technologies, all hosted in the Azure cloud. Kudos to authors Ovais Mehboob Ahmed Khan and Arvind Chandaka for making this reference available. Kudos to you, the reader, for investing the time to study and master the architecture.

—Robert Vettor
Principal Cloud Solution Architect
Microsoft

Introduction

Welcome to *Developing Microservices Architecture on Microsoft Azure with Open Source Technologies*. Today, organizations are modernizing application development by integrating open-source technologies into a holistic architecture for delivering high-quality workloads to the cloud. This book is a complete, step-by-step guide to building an application based on microservices architecture by leveraging services provided by the Microsoft Azure cloud platform and a litany of open-source technologies such as Java, Node.js, .NET Core, and Angular.

This will be a reference guide to learn key building blocks of microservices architecture by representing a real-life case study of building a trading system to auction items. This book will lead readers through a step-by-step journey to learn modern application development scenarios and leverage cloud-native capabilities in Microsoft Azure to provide distinct value to customers.

This topic is in very high demand based on statistics and our own experiences from the field. There have been books made on microservices, but they almost always cover business value proposition alone rather than technical implementation on how to establish end to end infrastructure, application development, deployment automation and to realize actual value through constructing it using Azure and open source.

We start with an introduction and modeling of microservices and then move on to build a cloud-native microservices architecture by developing services, containerizing services, and deploying them on AKS. We also cover various topics related to communication patterns, security, and API gateways, and we touch on topics related to automating builds and deployments using CI/CD and monitoring.

Who is this book for?

This book is for architects and developers. It especially targets those who have some development background to build applications or have worked in the capacity of designing and architecting applications. It will also be useful for those with some skills with Azure.

How is this book organized?

This book is organized into three parts:

- Part I: Introduction and modeling of microservices
- Part II: Designing and building microservices architecture
- Part III: Implementing patterns, security, DevOps, and monitoring

The first part of this book consists of two chapters that focus on a thorough introduction to microservices and how you can decompose or model microservices using domain-driven design (DDD).

The second part of this book consists of three chapters that focus on designing an architecture and building an application that follows this architecture with various open-source technologies such as Java, .NET Core, Node.js, and Angular. The architecture is cloud-native and leverages multiple managed services within Azure.

The third part of this book consists of five chapters that focuses on topics related to communication patterns, security within microservices, API gateways, automating builds and deployments, and monitoring.

System requirements

The requirements for running through the examples in this book should be:

- Any Linux distros or Windows operating systems
- 4 GB of RAM
- 1 GB of available disk space
- Docker with Linux Containers
- Microsoft Azure subscription
- Azure DevOps account
- Visual Studio Code or other editors of your choice

About the companion content

The companion content for this book can be downloaded from the following page: *MicrosoftPressStore.com/DevMicroservices/downloads*

Errata, updates, & book support

We've made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at:
MicrosoftPressStore.com/DevMicroservices/errata

If you discover an error that is not already listed, please submit it to us at the same page.

For additional book support and information, please visit *MicrosoftPressStore.com/Support*.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to http://support.microsoft.com.

Stay in touch

Let's keep the conversation going! We're on Twitter:
http://twitter.com/MicrosoftPress.

This page intentionally left blank

Chapter 1

Introduction to microservices

In this chapter, you will:

- Learn about the history and evolution of software architecture
- Understand the differences between monolithic and microservices architectures
- Learn about the core benefits and challenges of using a microservices architecture

Microservices is a buzzword that is thrown around frequently in the technology community. Many infer it as being a further dissection of modular componentization, and although that addresses part of it, it doesn't encompass the full picture. Microservices are a group of back-end services that provide business operations to form an application. When combined with other pieces such as a front-end and various communication platforms, it creates what is known as a *microservices architecture*. Although that is a simplistic way to approach this architecture, it is a vital design orientation and metaphor to learn for implementing modern application development scenarios geared to achieving better results.

This particular chapter will focus on the core fundamentals of microservices, including the transition from historical architecture to present, benefits and challenges, and a comparison of a monolithic application architecture with microservices architecture.

Our journey with microservices

In recent years, microservices architecture have become very popular with firms. Many organizations and enterprises have expressed a desire and need to move their applications to a microservices architecture.

We have worked with customers that were very concerned with increasing development velocity to ship code faster and at more frequent intervals. Other customers are focused on modularizing their codebase further to iterate on particular services for better application performance and additional support. Some are jaded by financial and time constraints that constantly plague their organization due to large scale migrations or refactoring and want to put an end to it. We have seen these pains, frustrations, complex scenarios, and wishes from a myriad of customers time and time again.

We have worked at Microsoft and in the IT space with many enterprise customers. We met with various internal open-source groups to find solutions to these problems of customers. Microservices architecture was the key behind this, and ultimately, we wanted to have a streamlined way to demonstrate the theoretical value of microservices and how to build one, too.

As a result, we created an application called the Online Auction System (OAS) to serve as a real-life case study to teach and empower customers to build this architecture for their enterprise scenarios. We accomplished this by using a variety of open-source technologies, and we used Microsoft Azure as the bedrock for the application. Throughout this book, we will use the approach to provide this background knowledge in conjunction with the parallels to this case study, which helps map theoretical and experiential learning together.

Evolution of software architecture

Before we can understand the details behind microservices architecture, let's take a brief look at how software architectures have evolved.

Monolithic architecture

About a decade ago, the monolithic application was very popular, and consequently its N-tiered architecture backbone was a norm. This consisted of an application decomposed into several layers. These layers vary depending on the complexity of the application, but traditionally, there were three main components. See Figure 1-1.

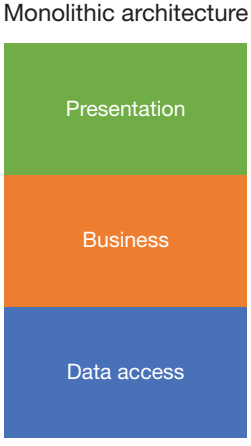


FIGURE 1-1 The monolithic architecture model

- **Presentation layer** The presentation layer traditionally represents the front-end. Some of the artifacts include view models, view pages, user interfaces, and other artifacts that are oriented toward front-end development.

- **Business layer** The middle layer is a business layer that holds all the logical flows of the application. This includes most of what we consider as the back-end code necessary for the primary function of the overall application. This area is where languages such as .NET or even Java are used to build back-end services that contain business managers and business objects consisting of the application logic.
- **Data access layer** Finally, the data access layer is responsible for all database operations. This consists of a database and interface to perform read/write operations on data collected and used within the application as part of its overall functionality.

  This layered architecture provides a basic separation of concerns and decomposition of applications into functional components. However, this architecture results in tightly coupled services that result in many dependencies.

Thinking of some of the scenarios from earlier, we can see why tightly coupled services and dependencies cause problems. Dependencies present drawbacks where it is difficult to make any changes to the application easily. This includes complexity in the form of large migrations and refactoring projects where many teams are involved and many times agility is replaced with a potential for bureaucracy. This introduces longer timelines because simple quick deployments cannot be made and ultimately all these problems cascade into a single point where management is then required to make massive expenditures.

Service-oriented architecture (SOA)

The transition away from monolithic architecture was driven by these reasons. The heavyweight nature of this architecture was slowly decomposed and eventually led to the formation of a new architecture known as service-oriented architecture (SOA), which is illustrated in Figure 1-2.

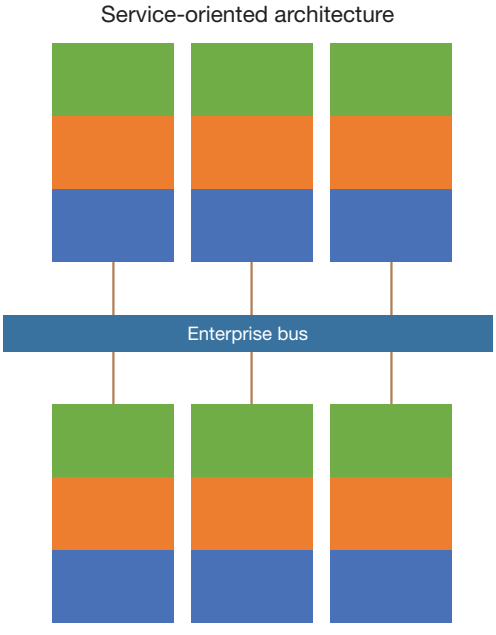


FIGURE 1-2 SOA architecture

This model was geared toward decomposing an application into even more granular modules to account for the drawbacks in monolithic architectures. Essentially, SOA was a way to connect different monoliths across a consistent messaging channel. Thus, as you can see in Figure 1-2, with SOA, you can have a set of services that are serving various functions connect and communicate with each other over a service bus. This service bus essentially is the communication tool that assists with message routing, transformations, security improvement, logging, and much more.

Because the services are broken down by function, it is slightly easier to deploy changes that don't affect the whole application. So, if you were building an application with a monolithic architecture, you could refactor it by using SOA architecture and addressing the litany of cascading problems we had with the monolithic architecture. Although SOA addresses some of these problems, we haven't solved a primary pain point in either architecture model—assuaging application scalability concerns.

Application scalability is an important design consideration when thinking about the growth of the application over time. With a larger user base, you will likely have more complex requirements, and thus considerations must be in place for handling additional requests and traffic. The clunky nature of the monolithic architecture doesn't address this and neither does SOA. Ultimately, with more scale, SOA breaks down internally because the single enterprise bus is overwhelmed, which slows (or throttles) during periods of increased requests and traffic.

That being said, the SOA concept ultimately introduced an important integration solution that helped applications talk to each other, and the additional consideration of scalability gave way to the expansion of the microservices architecture.

Comparing monolith with microservices

A microservices architecture is a series of services that communicate with one another to achieve complete application functionality. Each microservice has a similar pattern to a monolithic architecture but only covers one function within the entire application functionality.

This means that each service can have a data access layer that is customized to use several kinds of database technologies and a business layer that is customized to use wide varieties of technology to perform the functional aspects of the microservice. To elaborate on the business layer using our case study detailed in future chapters, this means that instead of having all the OAS back-end code in a single layer, we have individual services for setting up an auction or placing a bid.

The microservices architecture iterates on previous models with more communication and an all-encompassing front-end functionality. All the back-end services now communicate with each other to represent actions that users can experience through a single point of entry within the front-end, as illustrated in Figure 1-3.

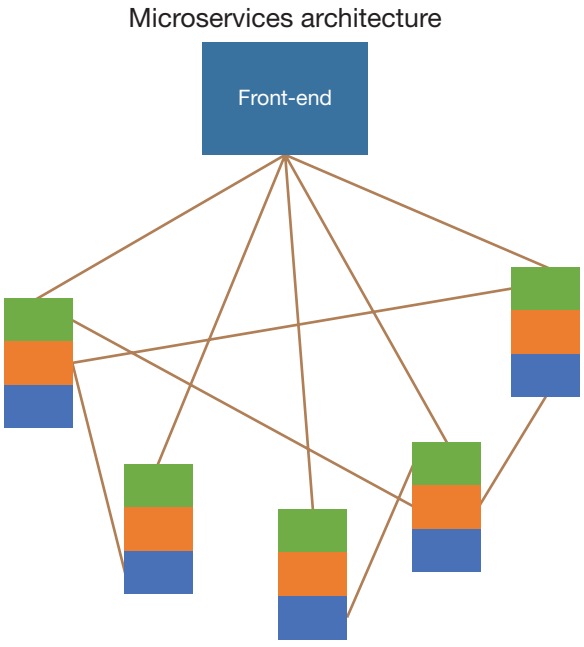


FIGURE 1-3 Microservices architecture

SOA versus microservices

You might also wonder how SOA differs from microservices. Microservices are fine-grained services where each service is modeled for a business domain or capability. SOA services are usually coarse-grained services that are not completely decomposed based on the business domain. Microservices are more abstracted and segmented by the individual business functionality. We will cover this in detail in future chapters, but the use of domain-driven design is a major design factor in microservices that is distinct from other architectures. Domain-driven design specifically aligns services to individual business functions in an application.

Also, microservices have more complex patterns when it comes to communication and messaging. Where we once had an enterprise bus as one of the innovative features of an SOA, we now have a broader set of technologies and mechanisms, such as request-response and publish-subscribe communications. These patterns will help distribute communication channels for better performance in an application, rather than making the service bus the bottleneck of the system.

Another advantage of microservices is rooted in using lightweight protocols, such as HTTP, whereas with SOA, we use other protocols such as TCP or MSMQ. There are many examples in which microservices are the presumptive choice for your application architectures. To help illustrate this better, we will dive deeper into some of the issues with legacy architectures.

Monolith example

Now that we have talked about the high-level differences, let's run through an exercise to understand the minute differentiations between monolithic and microservices architectures. To help illustrate this, see Figure 1-4, which illustrates how the monolithic architecture requires a full application deployed in each host, making it difficult to scale.

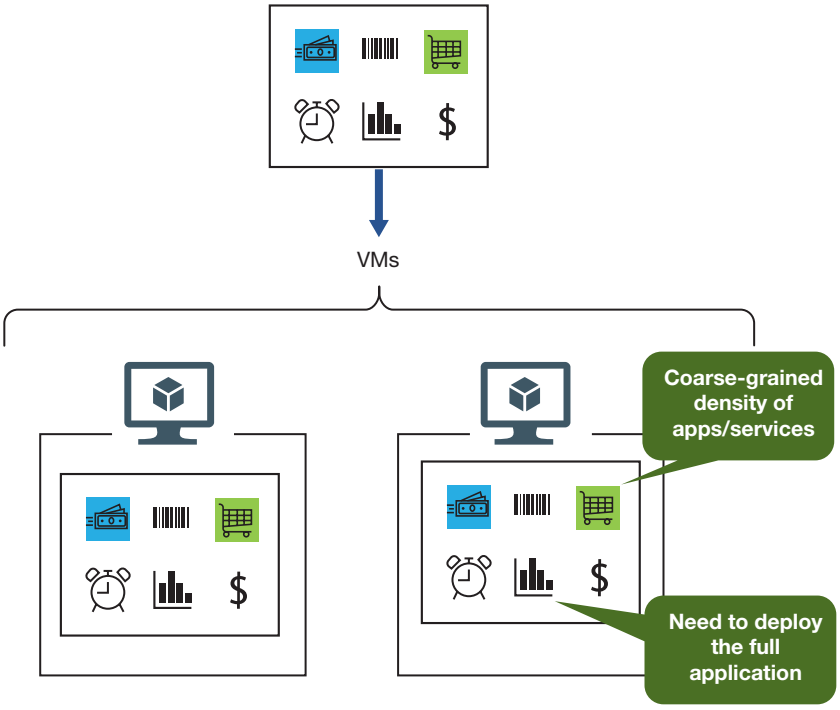


FIGURE 1-4 The monolithic architecture is difficult to scale

Figure 1-4 helps clarify the nuances of a monolithic application. Here, we have a collection of services that are ultimately deployed on two virtual machines (VMs)—all containing the same services within each host. As you can see, the scaling solution in place essentially deploys the full application on multiple VMs.

> ***NOTE*** **We know the monolithic architecture is scalable, however, it is expensive. We need to have the prerequisite specifications for these VMs with the same memory, compute power, and so on. Even though the deployment of the monolithic architecture might be simple because it is consistent, this can be expensive, whether you are utilizing your on-premises datacenter for the infrastructure or you are using a cloud provider with a pay-as-you-go subscription.**

Microservices example

Now let's compare the monolithic architecture with a microservices approach. Figure 1-5 illustrates how the microservices architecture can be deployed independently in several different hosts and is more capable of scaling.

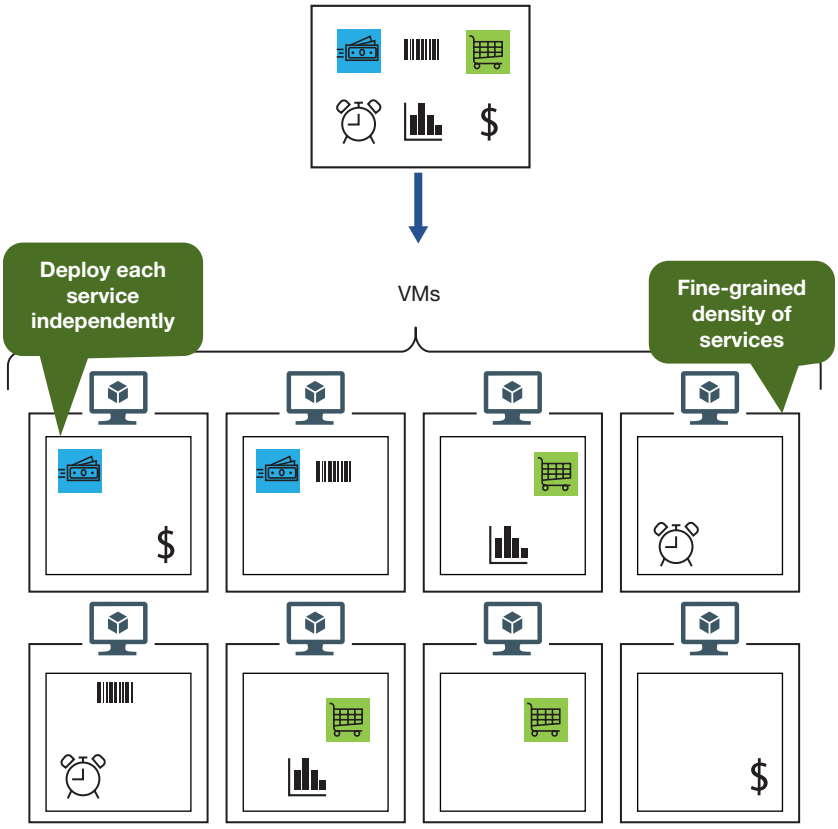


FIGURE 1-5 Microservices architecture

Figure 1-5 shows how microservices architecture differs from the monolithic approach. Virtual machines can be deployed in the microservices architecture, but instead of deploying all components of the application in a single VM, we see that each service is deployed independently on several VMs. Furthermore, with a monolithic application, any deployment that might have issues or bugs would have to be rolled back and analyzed carefully for mistakes. Not only is this tedious, but it provides a bad user experience for your app's consumers because of bugs and unexpected downtime. This allows us to scale based on functional need, doesn't result in overexpenditure of your resources, and enables a more fine-grained control of your services. If you experience deployment issues, you can easily find errors and redeploy with flexibility and velocity.

To understand how microservices promote scalability, suppose we are running an order management system in which users can place orders. On weekends, the system load is increased by a high number of user orders, which causes response timeouts and improper handling of resources. With the appropriate monitoring framework in place, we can identify when the order service is not handling a larger amount of requests. Based on this, we can just scale out the order service, rather than scaling out the whole system. This allows us to save our resources inexpensively.

> ***NOTE*** **Although we are demonstrating some of these basic concepts with microservices spread across VMs, a more regimented enterprise approach will use infrastructure such as Kubernetes, containers, and continuous integration and continuous development (CI/CD) as components to bolster some of the microservices benefits.**

Databases in a monolithic architecture

Let's focus more on the data tier of the architecture. Figure 1-6 shows that a monolithic database might not be a good design choice when it is shared across many services.

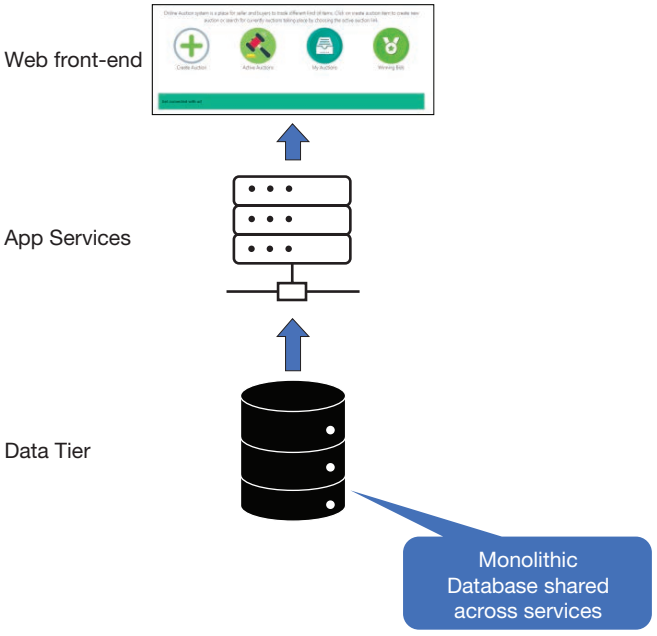


FIGURE 1-6 Monolithic database when shared across many APIs

As you can see, the data access layer in Figure 1-6 utilizes a single monolithic database that is shared across multiple services. This database also tends to be a relational database to store information, and all the app services that are operating in the middle tier perform actions directly on this single data tier.

If your application continues to grow, you might start noting issues with the scaling of your database and with maintaining this data. All the different services are recording particular information relevant to their service's operations, and it becomes very difficult to segregate this within your database.

Databases in microservices architecture

The microservices perspective is shown in Figure 1-7.

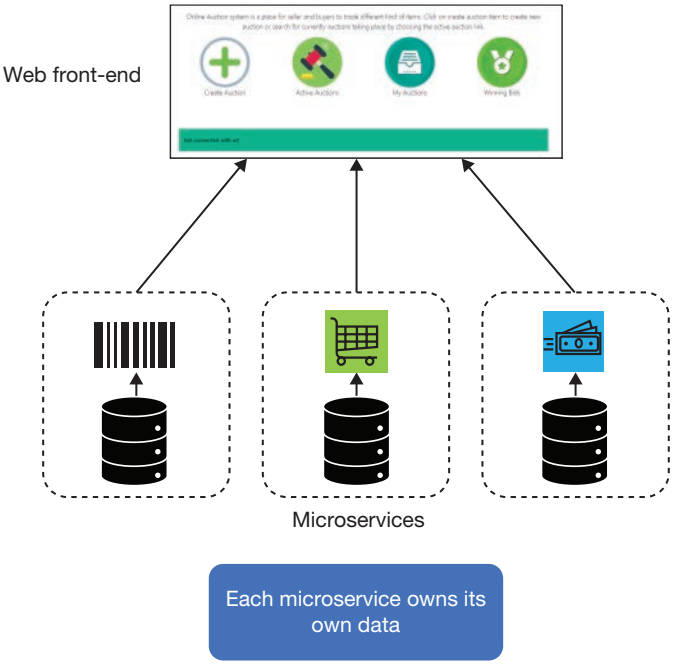


FIGURE 1-7 The microservices architecture

In microservices, you design with the domain or business functionality in mind. This means that you can segregate the services by business functionality, so each service can have its own particular database. Furthermore, you can specifically pick the technology behind the database of your choosing based on the need of the service. This provides even more customization, and it is a great benefit to leverage. Granted, when we abstract these services even more, we face difficulties with database consistency. Thankfully, as we mentioned earlier, there are many patterns in place to address that concern, which shows how microservices can truly make a difference in your application architecture.

Micro front-ends

When discussing the architectural features of microservices, we should also be aware of new movements and trends that continue to positively affect this paradigm, such as micro front-ends.

A micro front-end is similar to having various back-end microservices because the front-end is segmented along the same verticals. In the future, we will see more micro front-ends as we continue to evolve software architecture (see Figure 1-8).

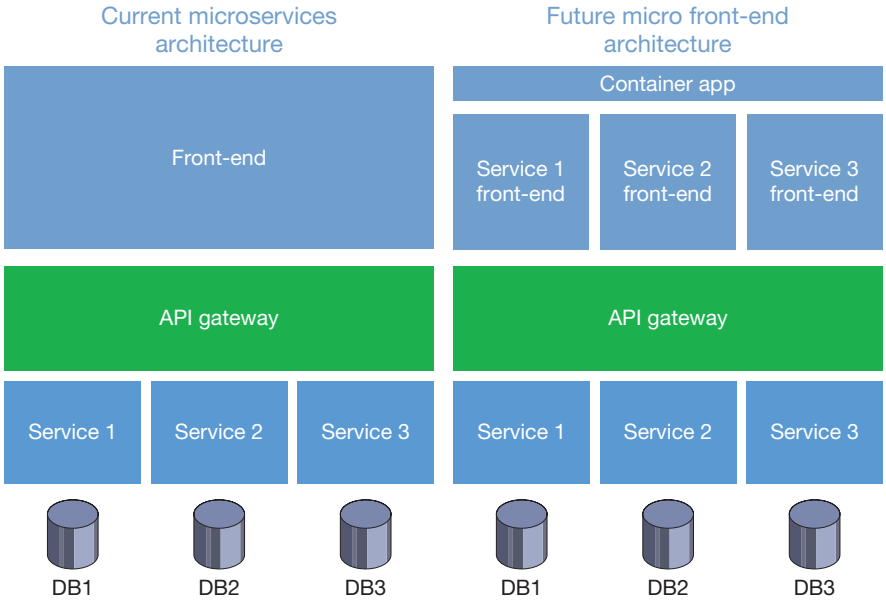


FIGURE 1-8 Microservices architecture versus micro front-end architecture

Figure 1-8 shows the current evolution of microservices architecture. When it comes to micro front-ends, we see that the front-end is further abstracted and connected to the back-end microservices over the same API gateway. This means that the UI/UX components of each of the services are segmented to those applications only. Then, all these separate components are placed together in a container app that serves as a placeholder for these HTML components with embedded data. When deployed, the components are rendered into a single front-end that is seen when the user accesses the overall application, say, through a web browser (assuming it was a web app). In traditional microservices, data components are received as JSON payloads, but in micro front-ends, data components are received as HTML components.

This componentization has similar benefits to those discussed in the next section, but this example helps illustrate its usefulness.

Let's take three teams—named Service 1, Service 2, and Service 3—operating in distinct service verticals. This means the Service 1 team can build and deploy the Service 1 full stack microservice, meaning they can build and deploy its respective data layer, business layer, and front-end. Likewise, the Service 2 and Service 3 teams can do the same, without any dependency on the other verticals.

Many of the benefits and challenges of micro front-end are strongly correlated to their respective back-end microservices. We will cover this in more detail in the next section. In this book, we use the OAS as a way to illustrate many of these concepts in practice, though we did

not use this particular feature in our case study. We still want you to be aware of micro front-ends because you will likely see their underlying patterns time and time again. Different layers and components of application architecture will always be further abstracted in order to create simplified, yet richer, scenarios for developers, engineers, and more. This is something that will always grow in popularity, and is a trend to observe and track as we continue evolving our application design strategies.

Core fundamentals of microservices

A fundamental understanding of microservices is vital to understanding the examples used throughout this book, as well as the real-life case study.

First, it's important to know that microservices are autonomous, independent, and loosely coupled services that cover business scenarios and collaborate with each other to form an application. As we know, each service will have the back-end code needed to perform a particular operation. In addition, each service will most likely have an endpoint to communicate with other services for entire application functionality. Finally, these services will also have a data access layer for maintaining service-specific data in individual databases.

Benefits

We have seen some of the scenarios and understood some of the pains that enterprise architects, developers, and consultants face when it comes to application architecture. Based on our experiences, we divided these benefits into three different categories:

- Agility
- Scalability
- Technology diversity

Agility

As shown in Figure 1-9, microservices are representative of individual functions within an overall application, and as such, business functionality being encapsulated into small targeted services is one of the major points of agility. In a traditionally monolithic application, there are three layers: presentation layer, business layer, and the data access layer. In this form of architecture, all the business logic is packed together in a single layer. With microservices, this is abstracted so that each microservice is representative of one component of the overall business logic, and a single team can be responsible for building that out. In fact, this attribute alone is considered one of the major hallmarks of microservices.

This is a method to ensure logical separation of concerns. When individual teams are aligned to microservices, this tends to lend itself to more agile development methodologies and ultimately, sections of the app can be changed without regard to issues stemming from tightly coupled functions.

The second benefit in agility is that each service can evolve independently and can be deployed frequently. This is of course with the assumption that the service endpoints are well defined and agreed upon by the teams building and consuming the services. In the team dynamic setting mentioned earlier in this chapter, we noted that each service can be built out by different teams because the overall business logic of the application is further abstracted. Additionally, there isn't dependency on the deployment side either, and separate continuous integration and deployment pipelines can be set up to make changes to each team's respective service. Ultimately, one team's work on a particular service will not affect another team's work. This independence ultimately reduces the mean time to delivery from a time-to-market perspective.

This results in development teams being more agile and being able to produce at a high velocity. Ultimately, they can focus better on their individual services. Development teams are now abstracted from the internal intricacies present when working with other teams from both a technological and process standpoint, which means they can be more productive overall.

Scalability

Microservices are very scalable in relation to an application. They can scale on respective demands without affecting overall performance. This occurs because of the abstraction of business functions combined with components of the overall architecture that build out the ability to handle more traffic and requests, and it has high availability. In fact, scalability was the feature that drove the transition from SOA to microservices.

We touched on some of these components earlier. Granular services mean that requests are oriented to particular areas within the architecture. However, when we have services communicating with others for an end-to-end scenario, communication patterns and messaging channels between individual services help streamline logic to complete scenarios. For microservices architecture, we have individual communications and patterns such as publish-subscribe which makes communication easier and doesn't throttle the performance of the application.

Also, microservices tend to be containerized in enterprise scenarios and thus can be deployed efficiently and frequently with the use of infrastructure to containerize the services (like Docker), deploy with scale with Kubernetes, and be able to deploy quickly with pipelining and continuous integration and deployment.

Technology Diversity

Often, balance and flexibility are overlooked in developer environments, which gives way to some rigid architecture that has specific problems and thus requires specific answers.

Developers who work on applications that utilize microservices architectures can mix and match different programming platforms and storage technologies. This offers great flexibility for developers, architects, and consultants to design with almost limitless permutations for a final business solution. This flexibility to leverage numerous technologies is an added benefit that we see time and time again in enterprise environments.

Consequently, applications can be refactored, modernized, or experimented on with new languages and practices, one service at a time. However, the introduction of new technology into longstanding enterprise environments tends to create new problems and headaches for incumbent IT professionals. This signals the start of large-scale refactoring and modernization projects, which equate to massive expenditures in time and money. Despite this, microservices change the way we think about these initiatives and instead of generating more problems, provides solutions for IT managers.

To understand this better, suppose we started with a monolithic application a decade ago to develop a product. In the last 10 years, we find that the application code base has become huge, and the database has become monolithic, too. Today, we realized that the technology we used earlier is getting outdated and making modifications to the existing application is difficult as one change will potentially break other pieces of the monolithic application. We want to provide a modern user experience to the consumer with new technologies and applications. This is where microservices solves these problems. With microservices, because the application is segregated into set of different services, the front-end could be a micro front-end or even a basic terminal that contains some HTML pages and consumes those services to represent application features. The segregation allows teams to independently make improvements to their service with minimal impact to the rest of the application. Adding new functionality is not only easy, but the adoption of cutting-edge technologies are faster and complementary.

Challenges

We've seen some of the fantastic benefits of using microservices architecture, many of which have themes related to componentization and flexibility. On the other hand, many of these same benefits pose new challenges for your IT professionals as well.

These challenges include:

- Learning curve
- Deployment
- Interaction
- Monitoring

Learning curve

In the modern enterprise environment, we have seen the shift from a traditional team setting to a DevOps-oriented team. A traditional company might have individualized teams of developers, QA engineers, an infrastructure team, an information security team, an operations team, and database admins—all in individual silos that have to work together in a waterfall-oriented project.

Today, agile projects are the norm and teams in a DevOps setting are almost mandatory, which means that teams are composed of a smattering of people from different traditional teams to make a completely iterative team that is capable of building out and maintaining some function.

However, this means that everyone on that agile team that is assigned to building out a particular microservice must ramp up based on the different kinds of technology that are selected for the final design.

The number of technologies here poses a problem, where more options cause decision fatigue and might exacerbate the learning curve. From the developer perspective, a design choice could be made to use Java for the business logic in a particular microservice. If this isn't the primary language that developers are used to building on, then ramping up to learn the language from a syntax, semantic, and even a knowledge perspective (considering libraries available) could be a significant barrier, which can pose both time and fiscal challenges when training time and resources are taken into consideration.

This isn't a situation that is unique to developers; it can happen to any role within your agile team. Let's take an infrastructure team who is responsible for deploying the underlying components needed to build out the application. The decision on what tool to use for infrastructure-as-code (IaC)—such as Terraform, ARM Templates, and so on—and for pipelines (build and deployments, i.e. Jenkin, Azure DevOps etc.) could cause delays and expenditures out of budget for training.

Building expertise on many different kinds of technology is difficult for teams in general, and it is even more dynamic when working with the microservices architecture. That being said, good planning and an understanding of your team's strengths and weaknesses can enable you to make the right decisions to balance the time and money to make the team productive when using microservices, despite these challenges.

Deployment

Agility is one of the big benefits of microservices, particularly because builds and deployments are faster and more frequent because teams own smaller parts of an overall application. However, this bonus comes with difficulty, where one needs a strong focus and expertise in automation for faster and frequent deployments.

A manual process to address builds and deployments would be cumbersome and painstaking. Thus, automation is required to reduce this overhead. The problem is deeper here because automation of pipelines in continuous integration and continuous deployment requires specialized technologies and skills, and automated pipelines are complex to set up based on how customized your environment is. Again, this could hinder the potential pros of using microservices.

Let's understand this better by walking through a scenario. With the assumption that the code base for your service is already complete, we need to ascertain where we will be storing the code. Traditionally, tools such as GitHub provide repos and source code version control, which is useful for developers to push code, generate builds, or make improvements to the service. Primarily, we want to focus on the integration of these pushes to the repo in kicking off the workflow for a build. There are many ways to do this with your tools, but then there is a need to create the build using the artifacts created from the code and running through tests

to ensure the build is working before moving it over for automatic deployment to your various environments. Each layer and step of this is complex and requires a deep understanding of multiple roles to work. This is why we see there is a high demand for DevOps and automation engineers. Furthermore, the tools that you could use for this end-to-end workflow can all vary in capability and have their own associated learning curve.

That being said, many enterprise IT professionals are observing and reacting to this need by seeking out people to help accomplish this in their environments based on incumbent talent, hired talent, and consultants they might be bringing in for these particular tasks. Microservices aside, to be successful in an agile IT world, we see this as being a scenario you cannot ignore.

Interaction

Adding on to some of the knowledge base needed to work in microservices are patterns and messaging channels. In a microservices architecture, although services are disparate and independent, the overall application functionality requires services to talk and communicate with one another for triggering of actions and functions to conceive an end-to-end cohesive workflow. Therefore, knowing messaging patterns and related technology is vital.In a traditional monolithic application, all the layers are stacked on top of each other, allowing interfacing between all parts of the application. In microservices, each function is a miniaturized monolithic application, but now it needs to communicate with several others to complete an end-to-end scenario and provide a rich user scenario/experience. Ultimately, seamless communication and independent scaling can only be achieved if services can asynchronously communicate with one another. This gave rise to technologies such as service buses and queues to help delineate communication between services.

Let's walk through an example that will reappear in detail in Chapter 6, which is how we implemented interaction with our OAS case study. The goal of this application is to provide a one-stop shop for people to auction off items, bid for items, pay once a bid is won, and subsequently receive the items they've purchased. This is a simplification of the scenario, but as you can see, with the moving parts of an application in place, several systems must talk to each other. We can see that the auction and bid services must communicate with each other, the front-end would have to communicate to all the back-end services, and the payment service must communicate with the bid services.

A service-oriented architecture has a single enterprise bus and doesn't account for all of these scenarios. Thus, a more loosely coupled architecture will require you to learn more patterns (such as request/response and pub/sub) and messaging technologies (such as Apache Kafka and RabbitMQ) to be productive. Again, this is a burden on the developer and infrastructure teams.

However, there are standard ways of learning this information, so although there is a learning curve, the information you gather will be useful in understanding how to design further architecture to handle various interactions.

Monitoring

Monitoring is a core component of many developers' day-to-day jobs. Monitoring is vital for understanding the performance of your particular application and helping identity bottlenecks, and it is used for overall troubleshooting. This is something already ingrained in the mindset of most developers. However, monitoring gets more difficult and is a strict requirement when it comes to microservices.

> ***NOTE*** **It is vital to log each and every detail relevant to these services, and you must be able to visualize these results. This involves the use of potential log ingestion/collection engines, visualization tools, alerting systems, and much more. Again, this overhead needs to be considered when shifting to this paradigm.**

There are several choices when it comes to monitoring, but we have found certain technologies and frameworks that work particularly well for microservices. One of the most useful things to do would be to understand the native features available from the cloud provider you are using. Today, it is common for many businesses to build modern, cloud-native applications. Because of this, there are likely cloud architects on teams that are experts on the monitoring capabilities native to your cloud platform. It is extremely valuable to leverage their insights and skills to building this piece of the puzzle.

In the OAS, we leveraged Azure and so our application metrics and performance are recorded by tools such as Azure Monitor, Log Analytics, Application Insights, and much more. Whatever your platform might be, the key to addressing a potential challenge here is being informed and intelligent about the different options available to you, whether it be cloud-native, third-party tools, or open-source tools, to make good decisions for your monitoring frameworks.

Moving forward with the microservices architecture, open source, and Azure

We hope that this provides a level set for readers to understand microservices architecture as we go deeper into our case study and the decisions we've had to make with Open Source Software (OSS) tooling and using Microsoft Azure.

Moving forward, we will go into greater depth in this book, describing different concepts in detail that we've touched on in this chapter. Our case study—the OAS shown in Figure 1-9—will be the context for understanding and using these concepts as we go through this journey together.

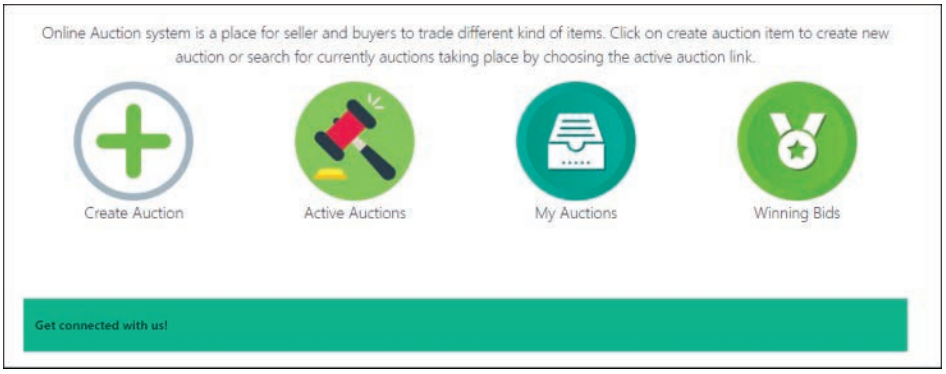


FIGURE 1-9 The Online Auction System is our real-life case study to help you understand how to use microservices architecture to build out your own modern, cloud-native applications

This book's goals

Ultimately, we hope this book will arm you—whether you are a developer, architect, consultant, or simply someone who wants to dive deeper into this topic—with the information you need to better understand the microservices architecture.

After reading this book, we hope you are sufficiently prepared to take on the challenge of building this out in a cloud-native setting such as Azure to solve your business need—whether it be through modernizing your application, migrating your application workloads, changing management, and so on. We hope that the world of microservices and open source can be the catalyst for making this a reality for you. Let's dive in!

Summary

In this chapter we covered:

- The history and evolution of software architecture from monolith all the way to microservices
- We compared different architecture models to help you understand the differences and basic inner workings of each
- We reviewed the benefits and challenges of microservices to understand the trade-offs when it comes to using this architecture

In the next chapter, we will elaborate on our real-life online auction system (OAS) case study. We will start by describing the base scenario and business requirements of the OAS and will then transition to the feature breakdown. We will discuss some decomposition principles and strategies with an emphasis on domain-driven design to demonstrate how we model our microservices.

This page intentionally left blank

Index

A

B

C

D

N

O

P

Q-R

S

T

U-V

W

X-Y-Z