Pro Microservices in .NET 6

With Examples Using ASP.NET Core 6, MassTransit, and Kubernetes

—

Sean Whitesell
Rob Richardson
Matthew D. Groves

Foreword by Scott Hunter
VP Director, Azure Developer Experience
Microsoft

Apress®

Pro Microservices in .NET 6

With Examples Using ASP.NET Core 6, MassTransit, and Kubernetes

Sean Whitesell
Rob Richardson
Matthew D. Groves

Foreword by Scott Hunter
VP Director, Azure Developer Experience
Microsoft

Apress®

Pro Microservices in .NET 6: With Examples Using ASP.NET Core 6, MassTransit, and Kubernetes

Sean Whitesell
KIEFER, OK, USA

Rob Richardson
Gilbert, AZ, USA

Matthew D. Groves
Grove City, OH, USA

ISBN-13 (pbk): 978-1-4842-7832-1
ISBN-13 (electronic): 978-1-4842-7833-8
https://doi.org/10.1007/978-1-4842-7833-8

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

Cover designed by eStudioCalamar

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/9781484278321. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to you, the reader. Writing good software is hard enough. Learning and conquering the development of microservices is an even greater challenge. I hope this book serves you well in your journey to developing great solutions for your users.

Table of Contents

About the Authors

Sean Whitesell is a Microsoft MVP and cloud architect at TokenEx, where he designs cloud-based architectural solutions for hosting internal services for TokenEx. He serves as President of the Tulsa Developers Association. He regularly presents in the community at developer events, conferences, and local meetups.

Rob Richardson is a software craftsman, building web properties in ASP.NET and Node, React, and Vue. He is a Microsoft MVP; published author; frequent speaker at conferences, user groups, and community events; and a diligent teacher and student of high-quality software development. You can find his recent work at robrich.org/presentations.

Matthew D. Groves is a Microsoft MVP who loves to code. From C# to jQuery, or PHP, he will submit pull requests for anything. He got his start writing a QuickBASIC point-of-sale app for his parents' pizza shop back in the 1990s. Currently a Product Marketing Manager for Couchbase, he is the author of the book *AOP in .NET* and created the video "Creating and Managing Your First Couchbase Cluster."

About the Technical Reviewer

Mike Benkovich A developer, business owner, consultant, cloud architect, Microsoft Azure MVP, and an online instructor, **Mike Benkovich** is an alumni of Microsoft from 2004 to 2012 where he helped build developer communities across the United States, through work on Microsoft Across America, MSDN Events, MSDN Webcasts, DPE, and Channel 9. He's helped to create and grow developer conferences and user groups in various cities across the United States. While at Microsoft he helped create the Azure Boot Camp events that were run in cities across the United States and at PDC and TechEd before it was transferred to the community. In his spare time he helped start a Toastmaster club for Geeks called TechMasters in Minneapolis where we grow speakers for conferences. He's a LinkedIn Learning Instructor for Azure, having developed many online courses. Mike actively works in Azure Cloud Governance, Application Architecture, and Software Delivery consulting.

Acknowledgments

There are challenges, and there are life goals. Writing this book has certainly been an accomplishment of a life goal. I could not have done this alone. There are people in my life that have given phenomenal support, and I'm eternally grateful.

To the Lord for helping me through challenging times, especially the ones I get myself into.

To my wife and biggest cheerleader, Barb, thank you for your unceasing support. It is so much more than I deserve. You have been so understanding of my goals and the challenges that come along with them. To my daughter McKayla, you are my gem and are the reason I fight hard to be a good dad. Remember, the best goals are worth fighting for.

To Michael Perry, I can't thank you enough. Your willingness to help me is amazing. I appreciate our discussions, where I get to learn so much from you. I'm thankful I got to learn from your book *The Art of Immutable Architecture*. Details in your book really helped this book and me as an architect.

To Floyd May, thank you so much for your friendship and our time on the whiteboard discussing microservices. I really appreciate your guidance.

To Phil Japikse, thank you so much for helping me get started with this book project. I appreciate your guidance throughout this book.

To Josh Brown, thank you so much brother for helping to spur ideas and the great discussions about databases.

To Rob Richardson and Matt Groves, thank you for helping me get this book done. I appreciate being able to lean on your expertise.

—Sean Whitesell

I would like to thank the Lord whose inspiration I rely on daily. His help has been instrumental in accomplishing this work.

—Rob Richardson

I'd like to acknowledge my patient wife Ali, Kevin and Mary Groves, and all of my Twitch audience that helped me to learn this microservices stuff.

—Matt Groves

Foreword

Software development is in the middle of a revolution. Moving away from monolithic application development with a team working on a large project that ships on a slow cadence to microservice based development where the application is broken into smaller pieces, which version independently, are built by smaller teams and ship on a fast cadence. .NET 6 is part of the revolution of .NET that makes it the perfect framework for building these microservice based applications.

.NET was re-imagined starting in 2016 to be the highest performance full stack development framework running across Linux, macOS and Windows on x86, x64, Arm32, Arm64 and M1 architectures. It includes support for cross platform RPC with gRPC, support for API's with Web API and Minimal API's and support for services with Worker Template.

Sean, Rob, and Matt have been building microservices in .NET and speaking on this form of development for many years. This book will help you learn how to build modern applications with microservices using the latest version of .NET.

I'm excited to see what you will build!

Scott Hunter
VP Director, Azure Developer Experience
Microsoft

Introduction

The microservice architecture breaks software into smaller pieces that can be independently deployed, scaled, and replaced. There are many benefits to this modern architecture, but there are more moving pieces.

In the olden days, we compiled the entire software product into one piece and deployed it infrequently. Deployment was hard, so we opted not to do it very often. With the advent of containers, deployment has become much easier. We can now break our application into lots of little pieces – microservices. When one microservice needs more horsepower, we can scale up only this portion of the web property. If a feature needs to work differently, we can deploy only this microservice, avoiding the churn with the entire system.

With this power come some additional layers of complexity. In the legacy monolithic software applications, we merely made a function call if we wanted to call into another part of the system. Our internal methods now have IP addresses, multiple instances, maybe load balancers distributing the load, and many more moving pieces.

How do we discover the address of the microservice? How do we scale to just the right level of availability without wasted cost? This is the magic of microservices, and this is the purpose of this book. You'll learn how to design, architect, scale, monitor, and containerize applications to build robust and scalable microservices.

Who Should Read This Book

In some respect, anyone involved with software projects related to distributed architecture should read this book. Even if a software project is not a distributed architecture but may become one, this book will shed some light on understanding existing business processes that may need to be handled by microservices.

From development managers to product owners to developers will find this book useful in understanding many complexities of a microservices architecture. Application architects and developers will gain quick insight with the hands-on code samples. The step-by-step coding approach covers examples with direct microservice calls as well as by messaging communication.

Book Organization

The microservices architecture is multifaceted and complex. Chapter 1 covers many of the subjects involved in this architecture style. Chapter 2 covers the advancements of .NET 6. In Chapter 3, we use a fictional story to help convey the purpose of breaking apart a monolithic application to a microservices architecture. We cover using Event Storming and Domain-Driven Design tenants to help understand existing business processes to determine where and why to create a microservice.

In Chapter 4, we cover direct communication with microservices. This chapter is also where you begin creating microservices using Visual Studio 2022 with .NET 6. Chapter 5 covers the messaging communication style. Also, you will create more microservices that communicate using MassTransit for messaging.

Chapter 6 covers breaking apart data from a centralized data store to distributed data stores. We also cover Saga patterns for handling transactions across multiple systems.

In Chapter 7, we cover testing the microservices using direct communication. We also cover testing the microservices that communicate using messaging. You will create the test projects for both communication styles.

Chapter 8 covers hosting microservices in Docker containers as well as using Kubernetes. By understanding containerization options, you understand how to handle the scaling of microservices.

In Chapter 9, we cover health concerns for microservices. The microservices developed in earlier chapters only have business logic. This chapter covers logging concerns, gathering metrics, tracing, and points for debugging.

CHAPTER 1

Introducing Microservices

Twitter, PayPal, and Netflix had serious problems. Problems like scaling, quality, and downtime became common and increasing issues. Each had a large, single-code base application known as a “monolith.” And each hit different frustration points where a fundamental architecture change had to occur. Development and deployment cycles were long and tedious, causing delays in feature delivery. Each deployment meant downtime or expensive infrastructure to switch from one set of servers to another. As the code base grew, so did the coupling between modules. With coupled modules, code changes are more problematic, harder to test, and lower overall application quality.

For Twitter, scaling servers was a huge factor that caused downtime and upset users. All too often, users would see an error page stating Twitter is overcapacity. Many users would see the “Fail Whale” while the system administrators would reboot servers and deal with the demand. As the number of users increased, so did the need for architecture changes. From the data stores, code, and server topology, the monolithic architecture hit its limit.

For PayPal, their user base increased the need for guaranteed transactions. They scaled up servers and network infrastructure. But, with the growing number of services, the performance hit a tipping point, and latency was the result. They continuously increased the number of virtual machines to process the growing number of users and transactions. This added tremendous pressure on the network, thereby causing latency issues.

Netflix encountered problems with scaling, availability, and speed of development. Their business required 24 × 7 access to their video streams. They were in a position where they could not build data centers fast enough to accommodate the demand. Their user base was increasing, and so were the networking speeds at homes and on devices. The monolithic application was so complex and fragile that a single semicolon took down the website for several hours.

In and of themselves, there is nothing wrong with a monolith. Monoliths serve their purpose, and when they need more server resources, it is usually cheap enough to add more servers. With good coding practices, a monolith can sustain itself very well. However, as they grow and complexity increases, they can reach a point that feature requests take longer and longer to implement. They turn into "monolith hell." It takes longer to get features to production, the number of bugs increases, and frustration grows with the users. Monolith hell is a condition the monolith has when it suffers from decreased stability, difficulty scaling, and nearly impossible to leverage new technologies.

Applications can grow into a burden over time. With changes in developers, skillsets, business priorities, etc., those applications can easily turn into a "spaghetti code" mess. As the demands of those applications change, so do the expectations with speed of development, testing, and deployment. By pulling functionality away from monolithic applications, development teams can narrow their focus on functionality and respective deployment schedule. This allows a faster pace of development and deployment of business functionality.

In this chapter, you will learn about the benefits of using a microservices architecture and the challenges of architecture changes. You will then learn about the differences between a monolithic architecture and a microservices architecture. Next, we will begin looking at microservices patterns, messaging, and testing. Finally, we will cover deploying microservices and examine the architectured infrastructure with cross-cutting concerns.

Benefits

For large applications suffering from "monolith hell," there are several reasons they may benefit by converting to a microservice architecture. Development teams can be more focused on business processes, code quality, and deployment schedules. Microservices scale separately, allowing efficient usage of resources on infrastructure. As communication issues and other faults occur, isolation helps keep a system highly available. Lastly, with architectural boundaries defined and maintained, the system can adapt to changes with greater ease. The details of each benefit are defined in the following.

Team Autonomy

One of the biggest benefits of using a microservice architecture is team autonomy. Companies constantly need to deliver more features in production in the fastest way possible. By separating areas of concern in the architecture, development teams can have autonomy from other teams. This autonomy allows teams to develop and deploy at a pace different than others. Time to market is essential for most companies. The sooner features are in production, the sooner they may have a competitive edge over competitors.

It also allows for but does not require different teams to leverage different programming languages. Monoliths typically require the whole code base to be in the same language. Because microservices are distinctly different applications, they open the door to using different languages, allowing flexibility in fitting the tool to the task at hand.

With data analytics, for example, Python is the most common programming language used and works well in microservice architectures. Mobile and front-end web developers can leverage languages best suited for those requirements, while C# is used with back-end business transaction logic.

With teams dedicated to one or more microservices, they only hold the responsibility for their services. They only focus on their code without the need to know details of code in other areas. Communication will need to be done regarding the API endpoints of the microservices. Clients need to know how to call these services with details such as HTTP verb and payload model, as well as the return data model. There is an API specification available to help guide the structure of your API. Consider the OpenAPI Initiative (`https://www.openapis.org/`) for more information.

Service Autonomy

As team autonomy focuses on the development teams and their responsibilities, service autonomy is about separating concerns at the service layer. The "Single Responsibility Principle" applies here as well. No microservice should have more than one reason to change. For example, an Order Management microservice should not also consist of business logic for Account Management. By having a microservice dedicated to specific business processes, the services can evolve independently.

Not all microservices exist alone with the processing of business logic. It is common to have microservices call others based on the data to process. The coupling is still loose and maintains the flexibility of code evolution.

With loose coupling between microservices, you receive the same benefits as when applied at the code level. Upgrading microservices is easier and has less impact on other services. This also allows for features and business processes to evolve at different paces.

The autonomy between microservices allows for individual resiliency and availability needs. For example, the microservice handling credit card payment has a higher availability requirement than handling account management. Clients can use retry and error handling policies with different parameters based on the services they are using.

Deployment of microservices is also a benefit of service autonomy. As the services evolve, they release separately using "Continuous Integration/Continuous Deployment" (CI/CD) tools like Azure DevOps, Jenkins, and CircleCI. Individual deployment allows frequent releases with minimal, if any, impact on other services. It also allows separate deployment frequency and complexity than with monolithic applications. This supports the requirement of zero downtime. You can configure a deployment strategy to bring up an updated version before taking down existing services.

Scalability

The benefit of scalability allows for the number of instances of services to differentiate between other services and a monolithic application. Generally, monolithic applications require larger servers than those needed for microservices. Having microservices lets multiple instances reside on the same server or across multiple servers, which aids in fault isolation. Figure 1-1 shows a relationship between the number of code instances and the size of the code.

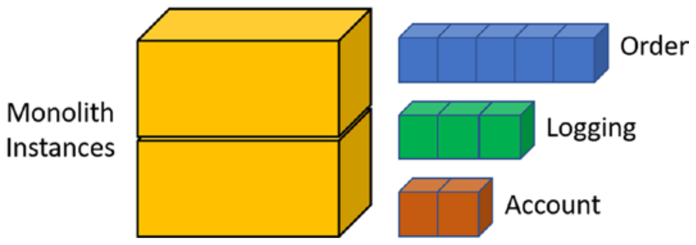

Figure 1-1. *Example of instance and size of code*

By utilizing a microservice architecture, the applications can leverage servers of diverse sizes. One microservice may need more CPU than RAM, while others require more in-memory processing capabilities. Other microservices may only need enough CPU and RAM to handle heavy I/O needs.

Another benefit of having microservices on different servers than the monolith is the diversity of programming languages. For example, assuming the monolith runs .NET Framework, you can write microservices in other programming languages. If these languages can run on Linux, then you have the potential of saving money due to the operating system license cost.

Fault Isolation

Fault isolation is about handling failures without them taking down an entire system. When a monolith instance goes down, all services in that instance also go down. There is no isolation of services when failures occur. Several things can cause failure:

- Coding or data issues
- Extreme CPU and RAM utilization
- Network
- Server hardware
- Downstream systems

With a microservice architecture, services with any of the preceding conditions will not take down other parts of the system. Think of this as a logical grouping. In one group are services and dependent systems that pertain to a business function. The functionality is separate from those in another group. If a failure occurs in one group, the effects do not spread to another group. Figure 1-2 is an oversimplification of services dependent on other services and a dependency on a data store.

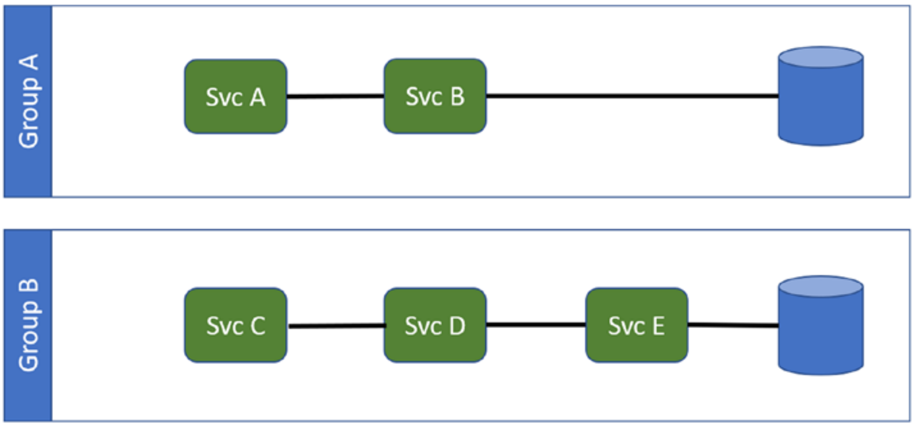

Figure 1-2. *Depiction of fault isolation*

As with any application that relies on remote processing, opportunities for failures are always present. When microservices either restart or are upgraded, any existing connections will be cut. Always consider microservices ephemeral. They will die and need to be restarted at some point. This may be from prolonged CPU or RAM usage exceeding a threshold. Orchestrators like Kubernetes will "evict" a pod that contains an instance of the microservice in those conditions. This is a self-preservation mechanism, so a runaway condition does not take down the server/node.

An unreasonable goal is to have a microservice with an uptime of 100% or 99.999% of the time. If a monolithic application or another microservice is calling a microservice, then retry policies must be in place to handle the absence or disappearance of the microservice. This is no different than having a monolithic application connecting with a SQL Server. It is the responsibility of the calling code to handle the various associated exceptions and react accordingly.

Retry policies in a circuit breaker pattern help tremendously in handling issues when calling microservices. Libraries such as Polly (`http://www.thepollyproject.org`) provide the ability to use a circuit breaker, retry policy, and others. This allows calling code to react to connection issues by retrying with progressive wait periods, then using an alternative code path if calls to the microservice fail X number of times.

Data Autonomy

So far, there have been many reasons presented for using a microservice architecture. But they focus on the business processes. The data is just as important, if not more so. Monolithic applications with the symptoms described earlier most certainly rely on a data store. Data integrity is crucial to the business. Without data integrity, no company will stay in business for long. Can you imagine a bank that “guesses” your account balance?

Microservices incorporate loose coupling, so changes deploy independently. Most often, these changes also contain schema changes to the data. New features may require new columns or a change to an existing column, as well as for tables. The real issue occurs when the schema change from one team impacts others. This, in turn, requires the changes to be backward compatible. Additionally, the other team affected may not be ready to deploy at the same time.

Having data isolated per microservice allows independent changes to occur with minimal impact on others. This isolation is another factor that encourages quicker time to production for the business. Starting a new feature with a new microservice with new data is great. Of course, that is easy to implement.

With separate databases, you also get the benefit of using differing data store technologies. Having separate databases provides an opportunity for some data to be in a relational database like SQL Server, while others are in non-relational databases like MongoDB, Azure Cosmos DB, and Azure Table Storage. Having a choice of different databases is another example of using the right tool for the job.

Challenges to Consider

Migrating to a microservice architecture is not pain-free and is more complex than monoliths. You will need to give yourself room to fail. Even with a small microservice, it may take several iterations to get to exactly what you need. And you may need to complete many rounds of refactoring on the monolith before you can support relocating functionality to a microservice. Developing microservices requires a new way of thinking about the existing architecture, such as the cost of development time and infrastructure changes to networks and servers.

If coming from a monolith, you will need to make code changes to communicate with the new microservice instead of just a simple method call. Communicating with microservices requires calls over a network and, most often, using a messaging broker. You will learn more about messaging later in this chapter.

The size of the monolithic applications and team sizes are also factors. Small applications, or large applications with small teams, may not see the benefits. The benefits of a microservice architecture appear when the overwhelming problems of "monolith hell" are conquered by separating areas.

Many companies are not ready to take on the challenges and simply host monolithic applications on additional servers and govern what business logic they process. Servers are relatively cheap, so spreading the processing load is usually the easiest "quick" solution. That is until they end up with the same issues as PayPal, Twitter, and others.

Developers may push back on the idea of microservice development. There is a large learning curve for the intricate details that need to be understood. And many developers will remain responsible for various parts of the monolithic applications, so it may feel like working on two projects simultaneously. There will be the ongoing question of quality vs. just getting something to production. Cutting corners will only add code fragility and technical debt and may prolong a successful completion.

A challenge every team will face is code competency. Developers must take the initiative to be strong with the programming language chosen and embrace distributed system design. Design patterns and best practices are great as they relate to the code in monoliths and inside the microservices. But new patterns must also be learned with how microservices communicate, handling failures, dependencies, and data consistency.

Another challenge for teams developing microservices is that there is more than code to consider. In the later section on "Cross-Cutting Concerns," items are described that affect every microservice, therefore every developer. Everyone should be involved in understanding (if not also creating) the items that help you understand the health of the architectural system. User stories or whatever task-based system you use will need additional time and tasks. This includes helping with testing the system and not just the microservices.

Microservice Beginning

With a primary system needing to work with other systems, there arose an issue of the primary system being required to know all the communication details of each connected system. The primary system, in this case, is your main application. Since each connected system had its own way of storing information, services it provided, and communication method, the primary system had to know all these details. This is a "tightly coupled" architecture. Suppose one of the connected systems changes to another system, a tremendous amount of change was required. Service-Oriented Architecture (SOA) aimed to eliminate the hassle and confusion. By using a standard communication method, each system could interact with less coupling.

The Enterprise Service Bus (ESB), introduced in 2002, was used to communicate messages to the various systems. An ESB provides a way for a "Publish/Subscribe" model in which each system could work with or ignore the message as they were broadcasted. Security, routing, and guaranteed message delivery are also aspects of an ESB.

When needing to scale a service, the whole infrastructure had to scale as well. With microservices, each service can scale independently. By shifting from ESB to protocols like HTTP, the endpoints become more intelligent about what and how to communicate. The messaging platform is no longer required to know the message payload, only the endpoint to give it to. "Smart Endpoints, Dumb Pipes" is how Martin Fowler succinctly stated.

So why now, in the last few years, have microservices gained attention? With the cost of supportive infrastructure, it is cheaper to build code and test to see if one or more microservices are the right way to go. Network and CPU have tremendously increased in power and are far more cost-effective today than yesteryear. Today, we can crunch through substantial amounts of data using mathematical models with data analytics and are gaining knowledge at a faster rate. For only $35, a Raspberry Pi can be bought and utilized to host microservices!

Cost is a huge factor, but so are the programming languages and platforms. Today, more than a handful of languages like C#, Python, and Node are great for microservices. Platforms like Kubernetes, Service Fabric, and others are vastly capable of maintaining microservices running in Docker containers. There are also far more programmers in the industry that can quickly take advantage of architectural patterns like SOA and microservices.

With the ever-increasing demand for software programmers, there also exists the demand for quality. It is way too easy for programmers to solve simple problems and believe they are "done." In reality, quality software is highly demanding of our time, talents, and patience. Just because microservices are cheaper and, in some cases, easier to create, they are by no means easy.

Architecture Comparison

Since most microservices stem from a monolithic application, we will compare the two architectures. Monoliths are the easiest to create, so it is no surprise this architecture is the de facto standard when creating applications. Companies need new features quickly for a competitive edge over others. The better and quicker the feature is in production, the sooner anticipated profits are obtained. So, as nearly all applications do, they grow. The code base grows in size, complexity, and fragility. In Figure 1-3, a monolith is depicted that contains a user interface layer, a business logic layer with multiple services, and a persistence layer.

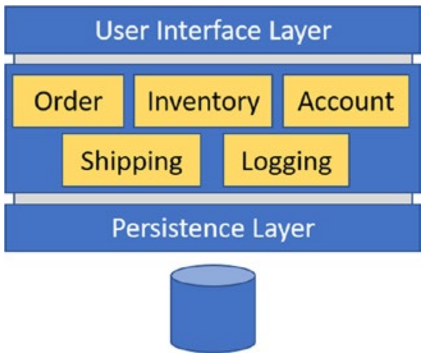

Figure 1-3. *Depiction of three-tier architecture*

A monolith, in the simplest term, is a single executable containing business logic. This includes all the supportive DLLs. When a monolith deploys, functionality stops and is replaced. Each service (or component) in a monolith runs "in process." This means that each instance of the monolith has the entire code base ready for instantiation.

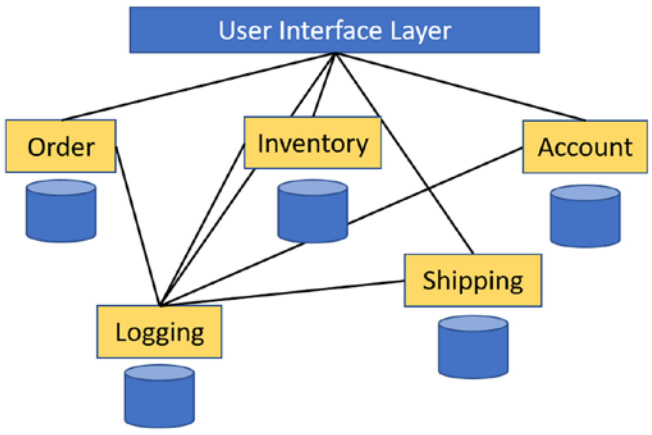

Figure 1-4. *Example of microservice architecture*

With the microservice architecture, shown in Figure 1-4, business logic is separated out into out-of-process executables. This allows them to have many instances of each running on different servers. As mentioned earlier, fault isolation is gained with this separation. If, for example, shipping was unavailable for a while, orders would still be able to be taken.

What is most realistic is the hybrid architecture, shown in Figure 1-5. Few companies fully transition to a microservice architecture completely. Many companies will take a sliver of functionality and partially migrate to a microservice solution.

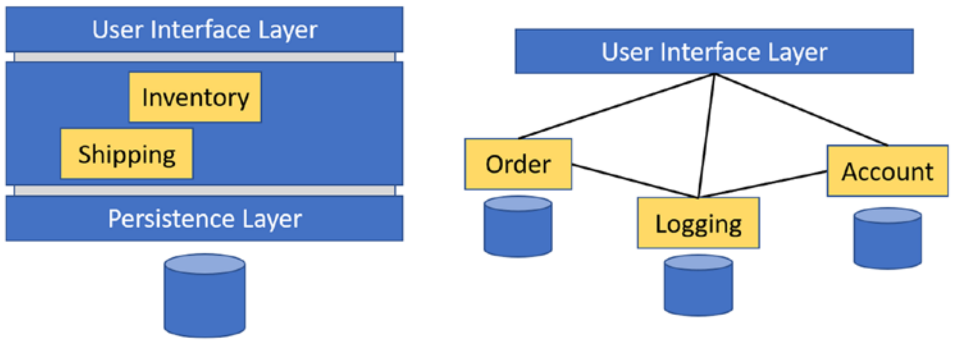

Figure 1-5. *Depiction of hybrid architecture*

When migrating from a monolithic to a microservice architecture, there is a huge danger when too much business functionality is in one microservice. For example, if the order microservice has tight coupling in the code with inventory management, and all of that logic was brought over, then you end up with a distributed monolith. You have gained some separation benefits while retaining many of the burdens the monolith has.

When you decide to venture down the path of creating microservices, start small. By starting with a small code base, you allow a way back. If the microservice is beyond time, cost, or patience, you will need to undo or abort changes to the monolith. While making these changes, continuously execute tests on the monolith looking for breaking code you did not expect.

Microservice Patterns

Every microservice architecture has challenges such as accessibility, obtaining configuration information, messaging, and service discovery. There are common solutions to these challenges called patterns. Various patterns exist to help solve these challenges and make the architecture solid.

API Gateway/BFF

The API Gateway pattern provides a single endpoint for client applications to the microservices assigned to it. Figure 1-6 shows a single API Gateway as an access point for multiple microservices. API Gateways provide functionality such as routing to microservices, authentication, and load balancing.

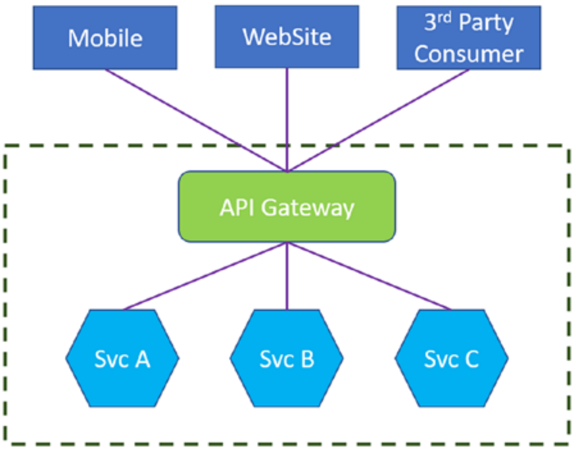

Figure 1-6. *Single API Gateway access point*