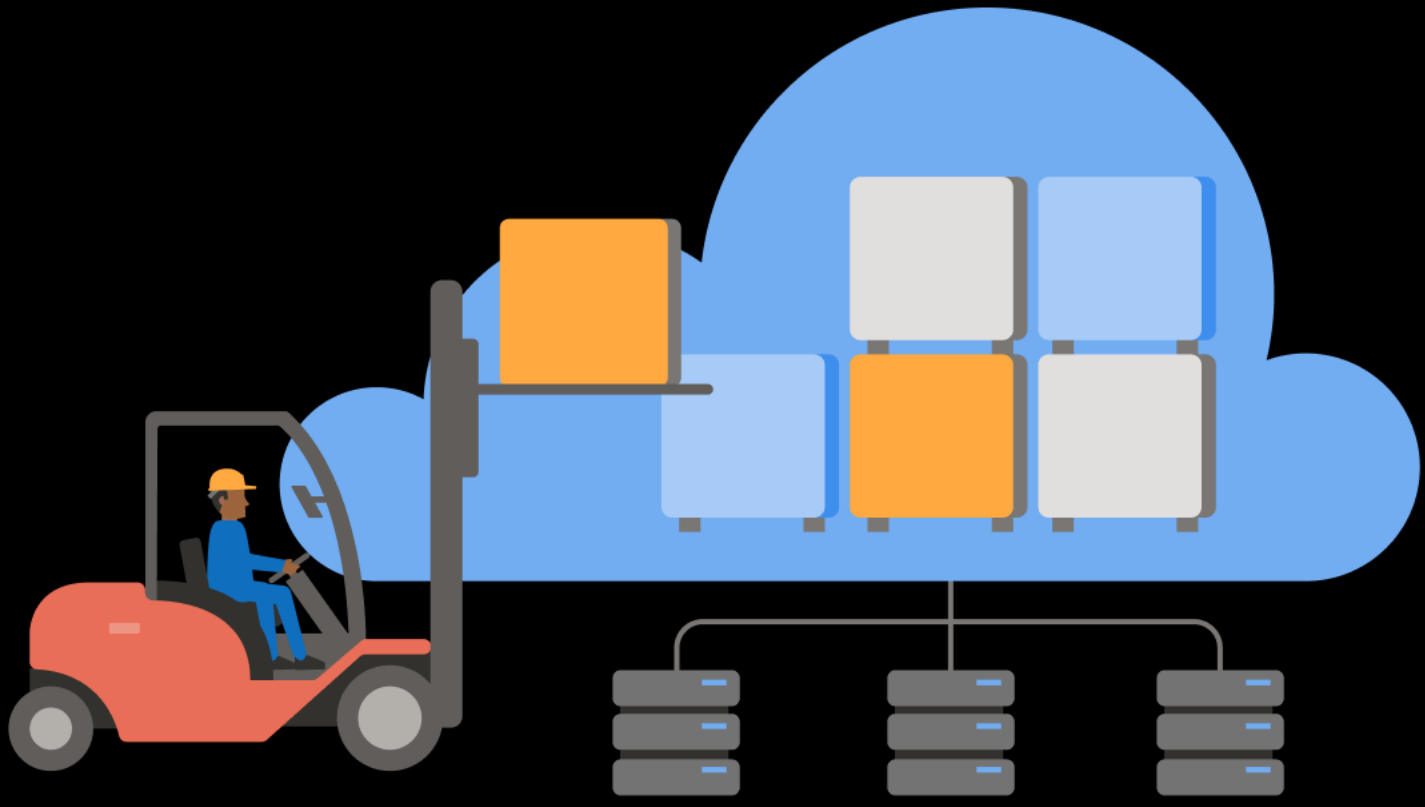
Microsoft

Architecting Cloud-Native .NET Apps for Azure



Robert Vettor

Steve "ardalis" Smith

Preview Edition

PUBLISHED BY

Microsoft Developer Division, .NET, and Visual Studio product teams

A division of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Important

PREVIEW EDITION

This book is a preview edition and is currently under construction. If you have any feedback, submit it at https://aka.ms/ebookfeedback.

Authors:

Steve "ardalis" Smith - Software Architect and Trainer - Ardalis.com

Rob Vettor - Microsoft - Principal Cloud System Architect/IP Architect - RobVettor.com

Participants and Reviewers:

Cesar De la Torre, Principal Program Manager, .NET team, Microsoft

Nish Anil, Sr. Program Manager, .NET team, Microsoft

Editors:

Maira Wenzel, Sr. Content Developer, .NET team, Microsoft

Who should use this guide

The audience for this guide is mainly developers, development leads, and architects who are interested in learning how to build applications designed for the cloud.

A secondary audience is technical decision-makers who plan to choose whether to build their applications using a cloud-native approach.

How you can use this guide

This guide begins by defining cloud native and introducing a reference application built using cloud-native principles and technologies. Beyond these first two chapters, the rest of the book is broken up into specific chapters focused on topics common to most cloud-native applications. You can jump to any of these chapters to learn about cloud-native approaches to:

- Data and data access
- Communication patterns
- Scaling and scalability
- Application resiliency
- Monitoring and health
- Identity and security
- DevOps

This guide is available both in PDF form and online. Feel free to forward this document or links to its online version to your team to help ensure common understanding of these topics. Most of these topics benefit from a consistent understanding of the underlying principles and patterns, as well as the trade-offs involved in decisions related to these topics. Our goal with this document is to equip teams and their leaders with the information they need to make well-informed decisions for their applications' architecture, development, and hosting.

Contents

Introduction to cloud-native applications

Another day, at the office, working on "the next big thing."

Your cellphone rings. It's your friendly recruiter - the one who calls you twice a day about new jobs.

But this time it's different: Start-up, equity, and plenty of funding.

The mention of the cloud and cutting-edge technology pushes you over the edge.

Fast forward a few weeks and you're now a new employee in a design session architecting a major eCommerce application. You're going to complete with other leading eCommerce sites.

How will you build it?

If you follow the guidance from past 15 years, you'll most likely build the system shown in Figure 1.1.

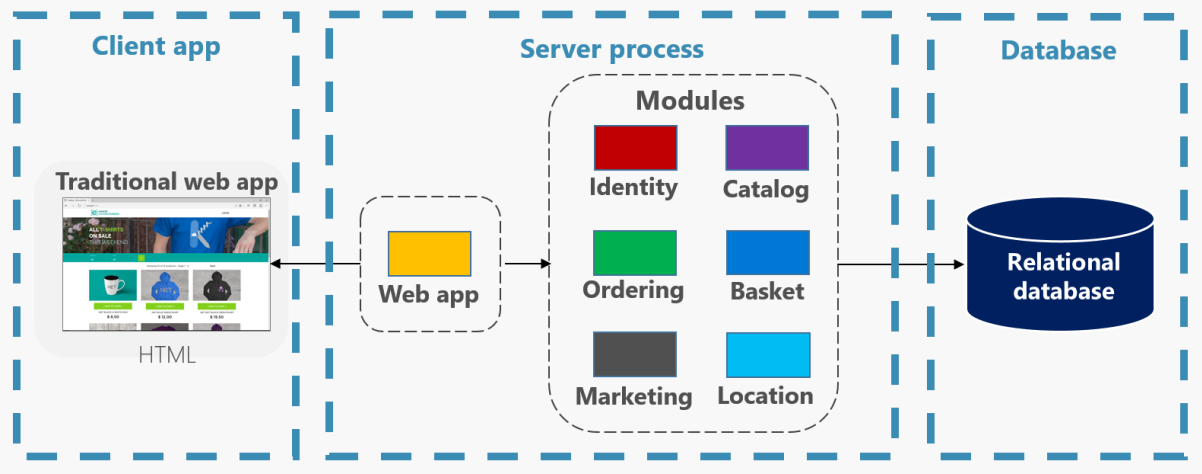


Figure 1-1. Traditional monolithic design

You construct a large core application containing all of your domain logic. It includes modules such as Identity, Catalog, Ordering, and more. The core app communicates with a large relational database. The core exposes functionality via an HTML interface.

Congratulations! You just created a monolithic application.

Not all is bad. Monoliths offer some distinct advantages. For example, they're straightforward to…

- build
- test
- deploy
- troubleshoot
- scale

Many successful apps that exist today were created as monoliths. The app is a hit and continues to evolve, iteration after iteration, adding more and more functionality.

At some point, however, you begin to feel uncomfortable. You find yourself losing control of the application. As time goes on, the feeling becomes more intense and you eventually enter a state known as the **Fear Cycle**.

- The app has become so overwhelmingly complicated that no single person understands it.
- You fear making changes - each change has unintended and costly side effects.
- New features/fixes become tricky, time-consuming, and expensive to implement.
- Each release as small as it may be requires a full deployment of the entire application.
- One unstable component can crash the entire system.
- New technologies and frameworks aren't an option.
- It's difficult to implement agile delivery methodologies.
- Architectural erosion sets in as the code base deteriorates with never-ending "special cases."
- The consultants tell you to rewrite it.

Many organizations have addressed the monolithic fear cycle by adopting a cloud-native approach to building systems. Figure 1-2 shows the same system built applying cloud-native techniques and practices.

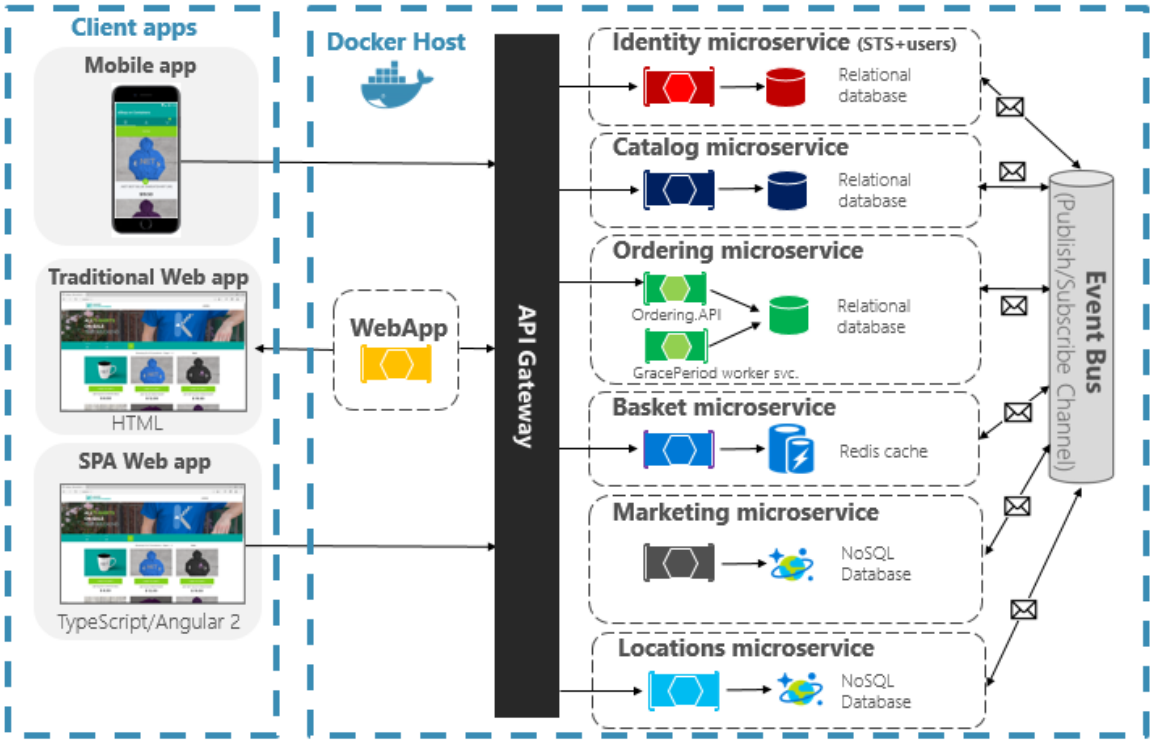


Figure 1-2. Cloud-native design

Note how the application is decomposed across a set of small isolated microservices. Each service is self-contained and encapsulates its own code, data, and dependencies. Each is deployed in a software container and managed by a container orchestrator. Instead of a large relational database, each service owns it own datastore, the type of which vary based upon the data needs. Note how some services depend on a relational database, but other on NoSQL databases. One service stores its state in a distributed cache. Note how all traffic routes through an API Gateway service that is responsible for routing traffic to the core back-end services and enforcing many cross-cutting concerns. Most importantly, the application takes full advantage of the scalability and resiliency features found in modern cloud platforms.

Cloud-native computing

Hmm… We just used the term, "*Cloud Native*." You first thought might be, "What exactly does that mean?" Another industry buzzword concocted by software vendors to market more stuff?"

Fortunately it's far different and hopefully this book will help convince you.

Within a short time, cloud native has become a driving trend in the software industry. It's a new way to think about building large, complex systems, an approach that takes full advantage of modern software development practices, technologies, and cloud infrastructure. The approach changes the way you design, implement, deploy, and operationalize systems.

Unlike the continuous hype that drives our industry, cloud native is "*for-real*." Consider the Cloud Native Computing Foundation (CNCF), a consortium of over 300 major corporations with a charter to

make cloud-native computing ubiquitous across technology and cloud stacks. As one of the most influential open-source groups, it hosts many of the fastest-growing open source-projects in GitHub. They include projects such as Kubernetes, Prometheus, Helm, Envoy, and gRPC.

The CNCF fosters an ecosystem of open-source and vendor-neutrality. Following that lead, we present cloud-native principles, patterns, and best practices that are technology agnostic. At the same time, we discuss the services and infrastructure available in the Microsoft Azure cloud for constructing cloud-native systems.

So, what exactly is Cloud Native? Sit back, relax, and let us help you explore this new world.

Defining cloud native

Stop what you're doing and text 10 of your colleagues. Ask them to define the term "Cloud Native." Good chance you'll get eight different answers. Interestingly, six months from now, as cloud-native technologies and practices evolve, so will their definition.

Cloud native is all about changing the way we think about constructing critical business systems.

Cloud-native systems are designed to embrace rapid change, large scale, and resilience.

The Cloud Native Computing Foundation provides an official definition:

Cloud-native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

Applications have become increasingly complex with users demanding more and more. Users expect rapid responsiveness, innovative features, and zero downtime. Performance problems, recurring errors, and the inability to move fast are no longer acceptable. They'll easily move to your competitor.

Cloud native is much about *speed* and *agility*. Business systems are evolving from enabling business capabilities to weapons of strategic transformation, accelerating business velocity and growth. It's imperative to get ideas to market immediately.

Here are some companies who have implemented these techniques. Think about the speed, agility, and scalability they've achieved.

Company	Experience
Netflix	Has 600+ services in production. Deploys a hundred times per day.
Uber	Has 1,000+ services stored in production. Deploys several thousand builds each week.
WeChat	Has 300+ services in production. Makes almost 1,000 changes per day.

As you can see, Netflix, Uber, and WeChat expose systems that consist of hundreds of independent microservices. This architectural style enables them to rapidly respond to market conditions. They can

instantaneously update small areas of a live, complex application, and individually scale those areas as needed.

The speed and agility of cloud native come about from a number of factors. Foremost is cloud infrastructure. Five additional foundational pillars shown in Figure 1-3 also provide the bedrock for cloud-native systems.

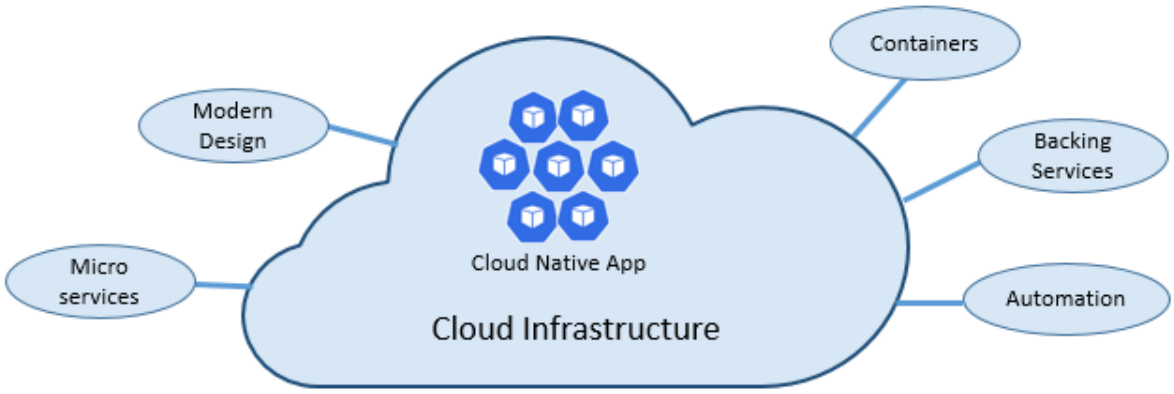


Figure 1-3. Cloud-native foundational pillars

Let's take some time to better understand the significance of each pillar.

The cloud...

Cloud-native systems take full advantage of the cloud service model.

Designed to thrive in a dynamic, virtualized cloud environment, these systems make extensive use of Platform as a Service (PaaS) compute infrastructure and managed services. They treat the underlying infrastructure as *disposable* - provisioned in minutes and resized, scaled, moved, or destroyed on demand – via automation.

Consider the widely accepted DevOps concept of Pets vs. Cattle. In a traditional data center, servers are treated as *Pets*: a physical machine, given a meaningful name, and cared for. You scale by adding more resources to the same machine (scaling up). If the server becomes sick, you nurse it back to health. Should the server become unavailable, everyone notices.

The *Cattle* service model is different. You provision each instance as a virtual machine or container. They're identical and assigned a system identifier such as Service-01, Service-02, and so on. You scale by creating more of them (scaling out). When one becomes unavailable, nobody notices.

The cattle model embraces *immutable infrastructure*. Servers aren't repaired or modified. If one fails or requires updating, it's destroyed and a new one is provisioned – all done via automation.

Cloud-native systems embrace the Cattle service model. They continue to run as the infrastructure scales in or out with no regard to the machines upon which they're running.

The Azure cloud platform supports this type of highly elastic infrastructure with automatic scaling, self-healing, and monitoring capabilities.

Modern design

How would you design a cloud-native app? What would your architecture look like? To what principles, patterns, and best practices would you adhere? What infrastructure and operational concerns would be important?

The Twelve-Factor Application

A widely accepted methodology for constructing cloud-based applications is the Twelve-Factor Application. It describes a set of principles and practices that developers follow to construct applications optimized for modern cloud environments. Special attention is given to portability across environments and declarative automation.

While applicable to any web-based application, many practitioners consider it as a solid foundation for building cloud-native apps. Systems built upon these principles can deploy and scale rapidly and add features to react quickly to market changes.

The following table highlights the Twelve-Factor methodology:

	Factor	Explanation
1	Code Base	A single code base for each microservice, stored in its own repository. Tracked with version control, it can deploy to multiple environments (QA, Staging, Production).
2	Dependencies	Each microservice isolates and packages its own dependencies, embracing changes without impacting the entire system.
3	Configurations	Configuration information is moved out of the microservice and externalized through a configuration management tool outside of the code. The same deployment can propagate across environments with the correct configuration applied.
4	Backing Services	Ancillary resources (data stores, caches, message brokers) should be exposed via an addressable URL. Doing so decouples the resource from the application, enabling it to be interchangeable.
5	Build, Release, Run	Each release must enforce a strict separation across the build, release, and run stages. Each should be tagged with a unique ID and support the ability to roll back. Modern CI/CD systems help fulfill this principle.
6	Processes	Each microservice should execute in its own process, isolated from other running services. Externalize required state to a backing service such as a distributed cache or data store.
7	Port Binding	Each microservice should be self-contained with its interfaces and functionality exposed on its own port. Doing so provides isolation from other microservices.
8	Concurrency	Services scale out across a large number of small identical processes (copies) as opposed to scaling-up a single large instance on the most powerful machine available.

	Factor	Explanation
9	Disposability	Service instances should be disposable, favoring fast startups to increase scalability opportunities and graceful shutdowns to leave the system in a correct state. Docker containers along with an orchestrator inherently satisfy this requirement.
10	Dev/Prod Parity	Keep environments across the application lifecycle as similar as possible, avoiding costly shortcuts. Here, the adoption of containers can greatly contribute by promoting the same execution environment.
11	Logging	Treat logs generated by microservices as event streams. Process them with an event aggregator and propagate the data to data-mining/log management tools like Azure Monitor or Splunk and eventually long-term archival.
12	Admin Processes	Run administrative/management tasks as one-off processes. Tasks can include data cleanup and pulling analytics for a report. Tools executing these tasks should be invoked from the production environment, but separately from the application.

In the book, Beyond the Twelve-Factor App, author Kevin Hoffman details each of the original 12 factors (written in 2011). Additionally, the book provides three additional factors that reflect today's modern cloud application design.

	New Factor	Explanation
13	API First	Make everything a service. Assume your code will be consumed by a front-end client. gateway or another service.
14	Telemetry	On a workstation, you have deep visibility into your application and its behavior. In the cloud, you don't. Make sure your design includes the collection of monitoring, domain-specific, and health/system data.
15	Authentication/ Authorization	Implement identity from the start. Consider RBAC (role-based access control) features available in public clouds.

We'll refer to many of the 12+ factors in this chapter and throughout the book.

Critical Design Considerations

Beyond the guidance provided from the twelve-factor methodology, there are several critical design decisions you must make when constructing distributed systems.

Communication

How will front-end client applications communicate with backed-end core services? Will you allow direct communication? Or, might you abstract the back-end services with a gateway façade that provides flexibility, control, and security?

How will back-end core services communicate with each other? Will you allow direct HTTP calls that lead to coupling and impact performance and agility? Or might you consider decoupled messaging with queue and topic technologies?

Communication is covered in detail Chapter 4, *Cloud-Native Communication Patterns*.

Resiliency

A microservices architecture moves your system from in-process to network communication. In a distributed environment, what will you do when Service B isn't responding to a call from Service A? What happens when Service C becomes up temporarily unavailable and other services calling it stack and degrade system performance?

Resiliency is covered in detail Chapter 6, *Cloud-Native Resiliency*.

Distributed Data

By design, each microservice encapsulates its own data, exposing operations via its public interface. If so, how do you query data or implement a transaction across multiple services?

Distributed data is covered in detail Chapter 5, *Cloud-Native Data Patterns*.

Identity

How will your service identify who is accessing it and what permissions they have?

Identity is covered in detail Chapter 8, *Identity*.

Microservices

Cloud-native systems embrace microservices, a popular architectural style for constructing modern applications.

Built as a distributed set of small, independent services that interact through a shared fabric, microservices share the following characteristics:

- Each implements a specific business capability within a larger domain context.
- Each is developed autonomously and can be deployed independently.
- Each is self-contained encapsulating its own data storage technology (SQL, NoSQL) and programming platform.
- Each runs in its own process and communicates with others using standard communication protocols such as HTTP/HTTPS, WebSockets, or AMQP.
- They compose together to form an application.

Figure 1-4 contrasts a monolithic application approach with a microservices approach. Note how the monolith is composed of a layered architecture, which executes in a single process. It typically consumes a relational database. The microservice approach, however, segregates functionality into independent services that include logic and data. Each microservice hosts its own datastore.

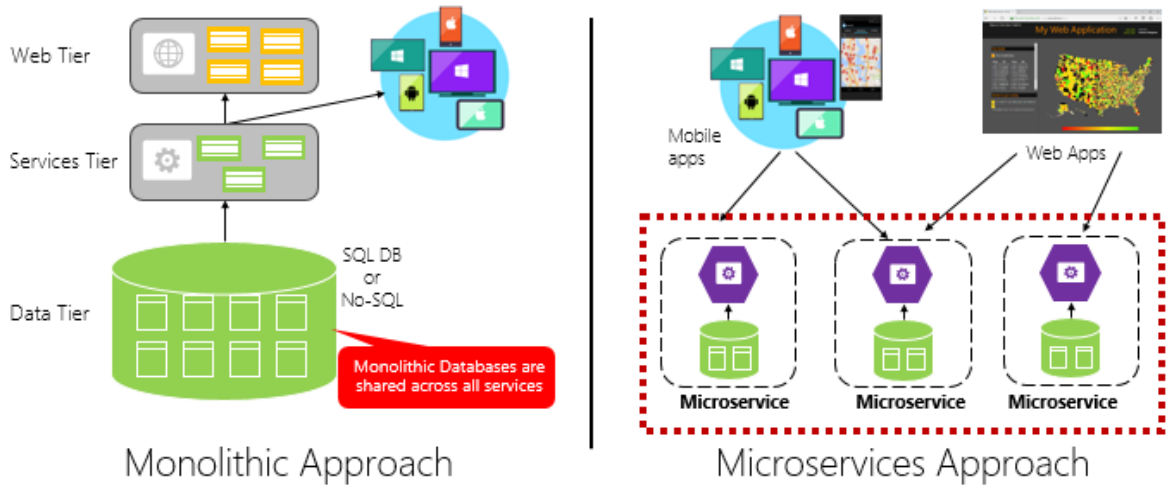


Figure 1-4. Monolithic deployment versus microservices

Note how microservices promote the "One Codebase, One Application" principle from the Twelve-Factor Application, discussed earlier in the chapter.

Factor #1 specifies "A single code base for each microservice, stored in its own repository. Tracked with version control, it can deploy to multiple environments"

Why microservices?

Microservices provide agility.

Earlier in the chapter, we compared an eCommerce application built as a monolith to that with microservices. In the example, we saw some clear benefits:

- Each microservice has an autonomous lifecycle and can evolve independently and deploy frequently. You don't have to wait for a quarterly release to deploy a new features or update. You can update a small area of a complex application with less risk of disrupting the entire system.
- Each microservice can scale independently. Instead of scaling the entire application as a single unit, you scale out only those services that require more processing power or network bandwidth. This fine-grained approach to scaling provides for greater control of your system and helps to reduce overall costs as you scale portions of your system, not everything.

An excellent reference guide for understanding microservices is .NET Microservices: Architecture for Containerized .NET Applications. The book deep dives into microservices design and architecture. It's a companion for a full-stack microservice reference architecture available as a free download from Microsoft.

Developing microservices

Microservices can be created with any modern development platform.

The Microsoft .NET Core platform is an excellent choice. Free and open source, it has many built-in features to simplify microservice development. .NET Core is cross-platform. Applications can be built and run on Windows, macOS, and most flavors of Linux.

.NET Core is highly performant and has scored well in comparison to Node.js and other completing platforms. Interestingly, TechEmpower conducted an extensive set of performance benchmarks across many web application platforms and frameworks. .NET Core scored in the top 10 - well above Node.js and other competing platforms.

.NET Core is maintained by Microsoft and the .NET community on GitHub.

Containers

Nowadays, it natural to hear the term *container* mentioned in any conversation concerning *cloud native*. In the book, Cloud Native Patterns, Author Cornelia Davis observes that, "Containers are a great enabler of cloud-native software." The Cloud Native Computing Foundation places microservice containerization as the first step in their Cloud-Native Trail Map - guidance for enterprises beginning their cloud-native journey.

Containerizing a microservice is simple and straightforward. The code, its dependencies, and runtime are packaged into a binary called a container image. Images are stored in a container registry, which acts as a repository or library for images. A registry can be located on your development computer, in your data center, or in a public cloud. Docker itself maintains a public registry via Docker Hub. The Azure cloud features a container registry to store container images close to the cloud applications that will run them.

When needed, you transform the image into a running container instance. The instance runs on any computer that has a container runtime engine installed. You can have as many instances of the containerized service as needed.

Figure 1-5 shows three different microservices, each in its own container, running on a single host.

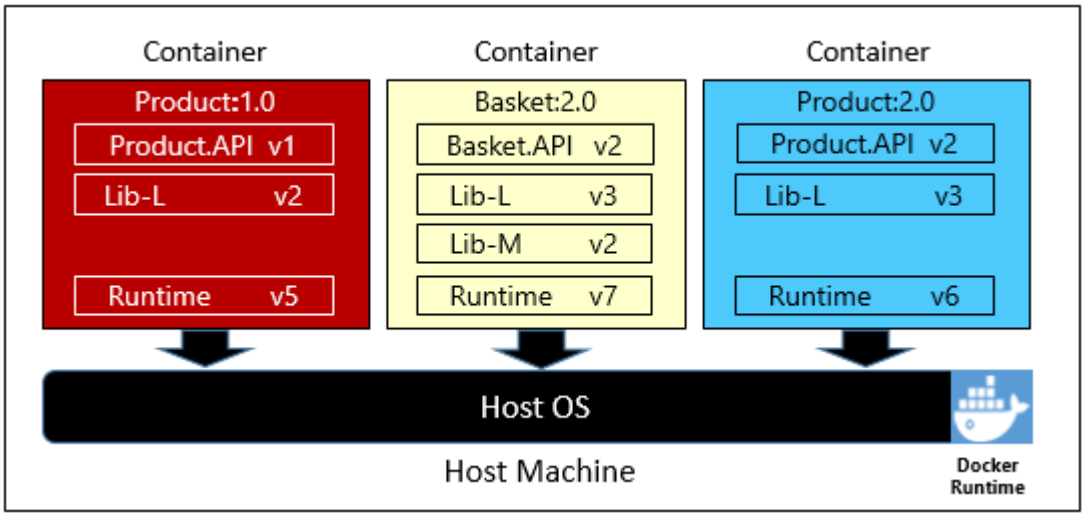


Figure 1-5. Multiple containers running on a container host

Note how each container maintains its own set of dependencies and runtime, which can be different. Here, we see different versions of the Product microservice running on the same host. Each container shares a slice of the underlying host operating system, memory, and processor, but is isolated from one another.

Note how well the container model embraces the "Dependencies" principle from the Twelve-Factor Application.

Factor #2 specifies that "Each microservice isolates and packages its own dependencies, embracing changes without impacting the entire system."

Containers support both Linux and Windows workloads. The Azure cloud openly embraces both. Interestingly, it's Linux, not Windows Server, that has become the most popular operating system in Azure.

While several container vendors exist, Docker has captured the lion's share of the market. The company has been driving the software container movement. It has become the de facto standard for packaging, deploying, and running cloud-native applications.

Why containers?

Containers provide portability and guarantee consistency across environments. By encapsulating everything into a single package, you *isolate* the microservice and its dependencies from the underlying infrastructure.

You can deploy that same container in any environment that has the Docker runtime engine. Containerized workloads also eliminate the expense of pre-configuring each environment with frameworks, software libraries, and runtime engines.

By sharing the underlying operating system and host resources, containers have a much smaller footprint than a full virtual machine. The smaller size increases the *density*, or number of microservices, that a given host can run at one time.

Container orchestration

While tools such as Docker create images and run containers, you also need tools to manage them. Container management is done with a special software program called a container orchestrator. When operating at scale, container orchestration is essential.

Figure 1-6 shows management tasks that container orchestrators provide.

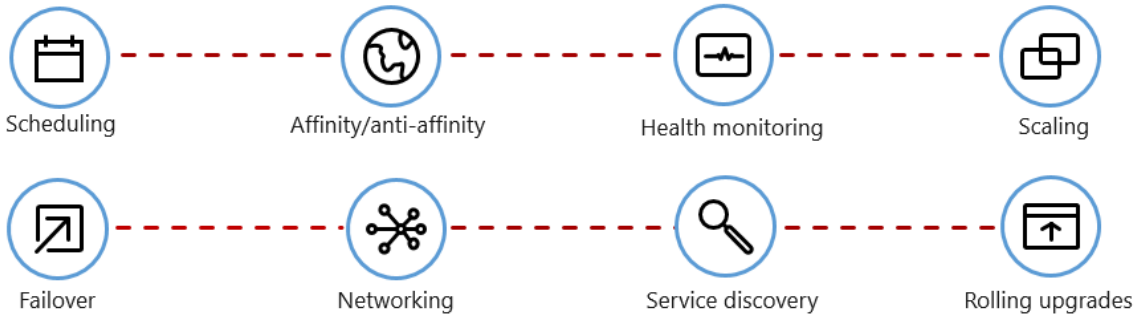


Figure 1-6. What container orchestrators do

The following table describes common orchestration tasks.

Tasks	Explanation
Scheduling	Automatically provision container instances.
Affinity/anti-affinity	Provision containers nearby or far apart from each other, helping availability and performance.
Health monitoring	Automatically detect and correct failures.
Failover	Automatically reprovision failed instance to healthy machines.
Scaling	Automatically add or remove container instance to meet demand.
Networking	Manage a networking overlay for container communication.
Service Discovery	Enable containers to locate each other.
Rolling Upgrades	Coordinate incremental upgrades with zero downtime deployment. Automatically roll back problematic changes.

Note how orchestrators embrace the disposability and concurrency principles from the Twelve-Factor Application, discussed earlier in the chapter.

Factor #9 specifies that "Service instances should be disposable, favoring fast startups to increase scalability opportunities and graceful shutdowns to leave the system in a correct state. Docker containers along with an orchestrator inherently satisfy this requirement."

Factor #8 specifies that "Services scale out across a large number of small identical processes (copies) as opposed to scaling-up a single large instance on the most powerful machine available."

While several container orchestrators exist, Kubernetes has become the de facto standard for the cloud-native world. It's a portable, extensible, open-source platform for managing containerized workloads.

You could host your own instance of Kubernetes, but then you'd be responsible for provisioning and managing its resources - which can be complex. The Azure cloud features Kubernetes as a managed service, Azure Kubernetes Service (AKS). A managed service allows you to fully leverage its features, without having to install and maintain it.

Azure Kubernetes Services is covered in detail Chapter 2, *Scaling Cloud-Native Applications*.

Backing services

Cloud-native systems depend upon many different ancillary resources, such as data stores, message brokers, monitoring, and identity services. These services are known as backing services.

Figure 1-7 shows many common backing services that cloud-native systems consume.

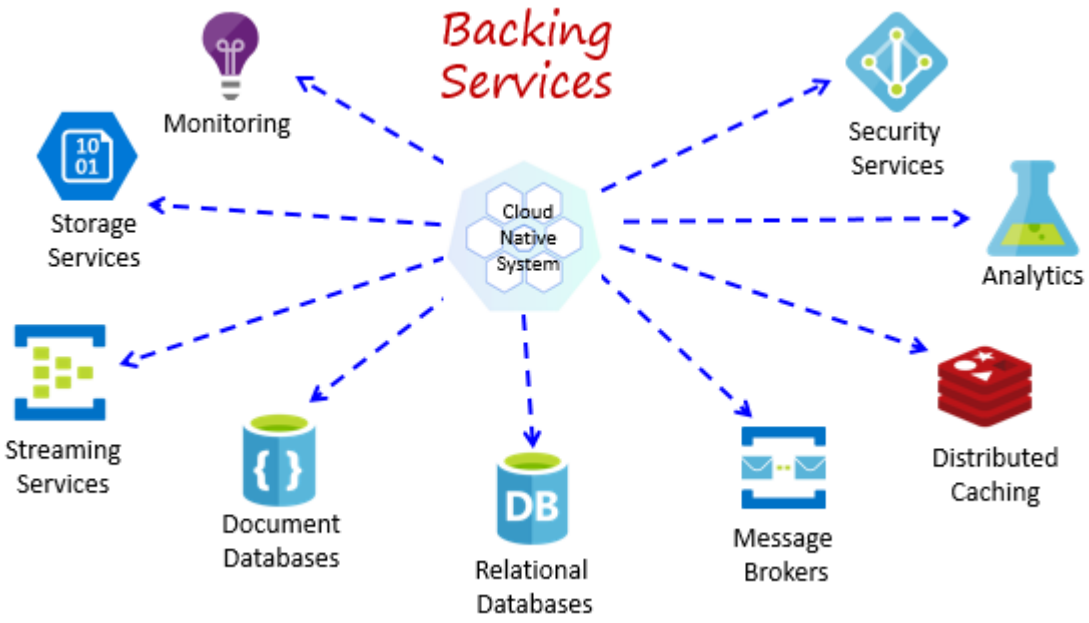


Figure 1-7. Common backing services

Backing services promote the "Statelessness" principle from the Twelve-Factor Application, discussed earlier in the chapter.

Factor #6 specifies that, "Each microservice should execute in its own process, isolated from other running services. Externalize required state to a backing service such as a distributed cache or data store."

You could host your own backing services, but then you'd be responsible for licensing, provisioning, and managing those resources.

Cloud providers offer a rich assortment of *managed backing services*. Instead of owning the service, you simply consume it. The provider operates the resource at scale and bears the responsibility for performance, security, and maintenance. Monitoring, redundancy, and availability are built into the service. Providers fully support their managed services - open a ticket and they fix your issue.

Cloud-native systems favor managed backing services from cloud vendors. The savings in time and labor are great. The operational risk of hosting your own and experiencing trouble can get expensive fast.

A best practice is to treat a backing service as an *attached resource*, dynamically bound to a microservice with information (a URL and credentials) stored in an external configuration. This guidance is spelled out in the Twelve-Factor Application, discussed earlier in the chapter.

Factor #4 specifies that backing services "should be exposed via an addressable URL. Doing so decouples the resource from the application, enabling it to be interchangeable."

Factor #3 specifies that "Configuration information is moved out of the microservice and externalized through a configuration management tool outside of the code."

With this pattern, a backing service can be attached and detached without code changes. You might promote a microservice from QA to a staging environment. You update the microservice configuration to point to the backing services in staging and inject the settings into your container through an environment variable.

Cloud vendors provide APIs for you to communicate with their proprietary backing services. These libraries encapsulate the plumbing and complexity. Communicating directly with these APIs will tightly couple your code to the backing service. It's a better practice to insulate the implementation details of the vendor API. Introduce an intermediation layer, or intermediate API, exposing generic operations to your service code. This loose coupling enables you to swap out one backing service for another or move your code to a different public cloud without having to make changes to the mainline service code.

Backing services are discussed in detail Chapter 5, *Cloud-Native Data Patterns*, and Chapter 4, *Cloud-Native Communication Patterns*.

Automation

As you've seen, cloud-native systems embrace microservices, containers, and modern system design to achieve speed and agility. But, that's only part of the story. How do you provision the cloud environments upon which these systems run? How do you rapidly deploy app features and updates? How do you round out the full picture?

Enter the widely accepted practice of Infrastructure as Code, or IaC.

With IaC, you automate platform provisioning and application deployment. You essentially apply software engineering practices such as testing and versioning to your DevOps practices. Your infrastructure and deployments are automated, consistent, and repeatable.

Automating infrastructure

Tools like Azure Resource Manager, [Terraform, and the Azure CLI, enable you to declaratively script the cloud infrastructure you require. Resource names, locations, capacities, and secrets are parameterized and dynamic. The script is versioned and checked into source control as an artifact of your project. You invoke the script to provision a consistent and repeatable infrastructure across system environments, such as QA, staging, and production.

Under the hood, IaC is idempotent, meaning that you can run the same script over and over without side effects. If the team needs to make a change, they edit and rerun the script. Only the updated resources are affected.

In the article, What is Infrastructure as Code, Author Sam Guckenheimer describes how, "Teams who implement IaC can deliver stable environments rapidly and at scale. Teams avoid manual configuration of environments and enforce consistency by representing the desired state of their environments via code. Infrastructure deployments with IaC are repeatable and prevent runtime issues caused by configuration drift or missing dependencies. DevOps teams can work together with a unified set of practices and tools to deliver applications and their supporting infrastructure rapidly, reliably, and at scale."

Automating deployments

The Twelve-Factor Application, discussed earlier, calls for separate steps when transforming completed code into a running application.

Factor #5 specifies that "Each release must enforce a strict separation across the build, release and run stages. Each should be tagged with a unique ID and support the ability to roll back."

Modern CI/CD systems help fulfill this principle. They provide separate deployment steps and help ensure consistent and quality code that's readily available to users.

Figure 1-8 shows the separation across the deployment process.

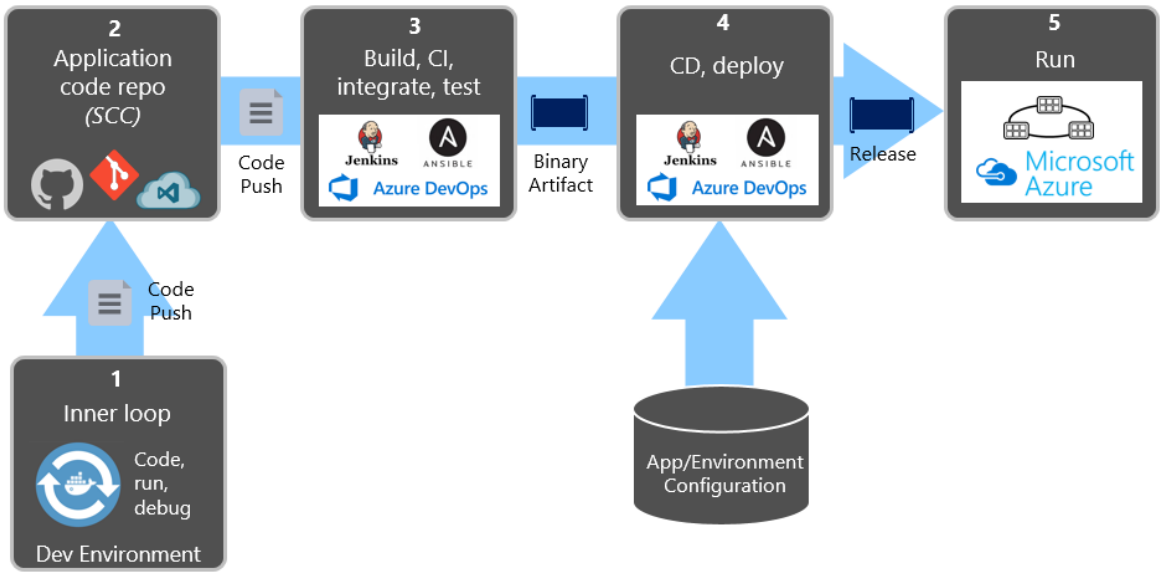


Figure 1-8. Deployment steps in a CI/CD Pipeline

In the previous figure, pay special attention to separation of tasks.

The developer constructs a feature in their development environment, iterating through what is called the "inner loop" of code, run, and debug. When complete, that code is *pushed* into a code repository, such as GitHub, Azure DevOps, or BitBucket.

The push triggers a build stage that transforms the code into a binary artifact. The work is implemented with a Continuous Integration (CI) pipeline. It automatically builds, tests, and packages the application.

The release stage picks up the binary artifact, applies external application and environment configuration information, and produces an immutable release. The release is deployed to a specified environment. The work is implemented with a Continuous Delivery(CD) pipeline. Each release should be identifiable. You can say, "This deployment is running Release 2.1.1 of the application."

Finally, the released feature is run in the target execution environment. Releases are immutable meaning that any change must create a new release.

Applying these practices, organizations have radically evolved how they ship software. Many have moved from quarterly releases to on-demand updates. The goal is to catch problems early in the development cycle when they're less expensive to fix. The longer the duration between integrations, the more expensive problems become to resolve. With consistency in the integration process, teams can commit code changes more frequently, leading to better collaboration and software quality.

Azure Pipelines

The Azure cloud includes a new CI/CD service entitled Azure Pipelines, which is part of the Azure DevOps offering shown in Figure 1-9.

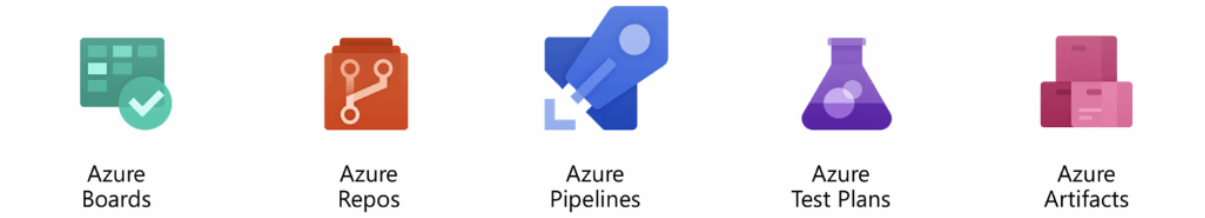


Figure 1-9. Azure DevOps offerings

Azure Pipelines is a cloud service that combines continuous integration (CI) and continuous delivery (CD). You can automatically test, build, and ship your code to any target.

You define your pipeline in code in a YAML file alongside the rest of the code for your app.

- The pipeline is versioned with your code and follows the same branching structure.
- You get validation of your changes through code reviews in pull requests and branch build policies.
- Every branch you use can customize the build policy by modifying the azure-pipelines.yml file.
- The pipeline file is checked into version control and can be investigated if there's a problem.

The Azure Pipelines service supports most Git providers and can generate deployment pipelines for applications written on the Linux, macOS, or Windows platforms. It includes support for Java, .NET, JavaScript, Python, PHP, Go, XCode, and C++.

Candidate apps for cloud native

Look at the apps in your portfolio. How many of them qualify for a cloud-native architecture? All of them? Perhaps some?

Applying a cost/benefit analysis, there's a good chance that most wouldn't support the hefty price tag required to be cloud native. The cost of being cloud native would far exceed the business value of the application.

What type of application might be a candidate for cloud native?

- A large, strategic enterprise system that needs to constantly evolve business capabilities/features
- An application that requires a high release velocity - with high confidence
- A system with where individual features must release *without* a full redeployment of the entire system
- An application developed by teams with expertise in different technology stacks
- An application with components that must scale independently

Then there are legacy systems. While we'd all like to build new applications, we're often responsible for modernizing legacy workloads that are critical to the business. Over time, a legacy application could be decomposed into microservices, containerized, and ultimately "replatformed" into a cloud-native architecture.

Modernizing legacy apps

The free Microsoft e-book Modernize existing .NET applications with Azure cloud and Windows Containers provides guidance for migrating on-premises workloads into cloud. Figure 1-10 shows that there isn't a single, one-size-fits-all strategy for modernizing legacy applications.

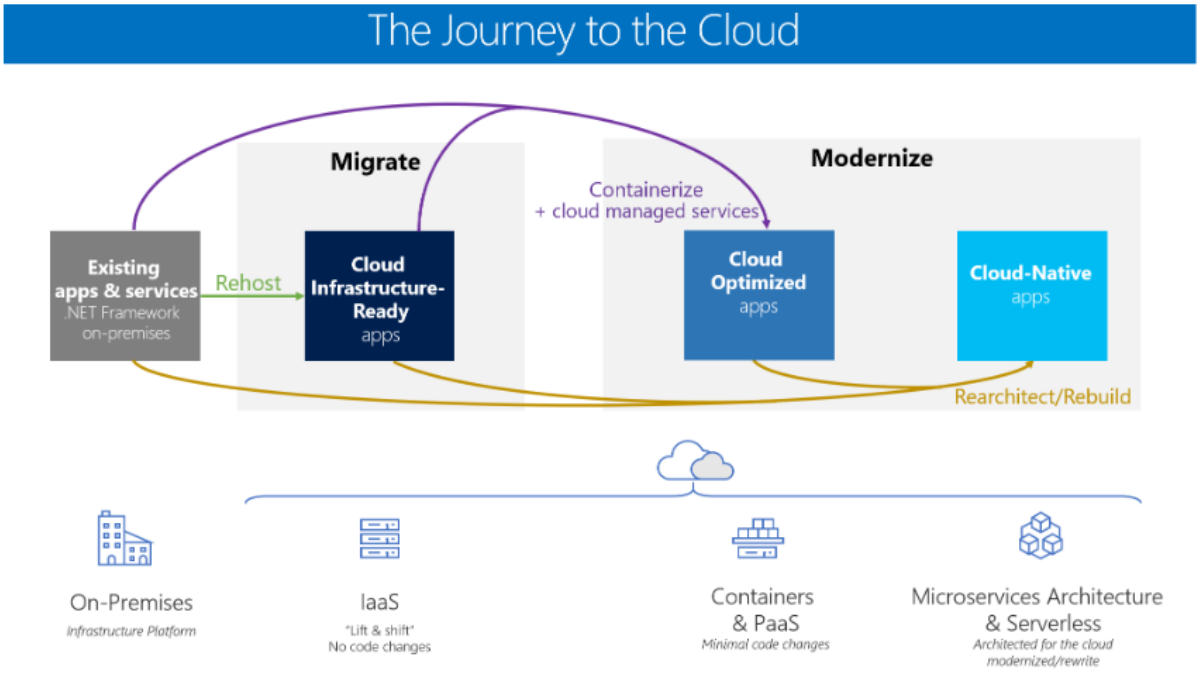


Figure 1-10. Strategies for migrating legacy workloads

Monolithic apps that are non-critical largely benefit from a quick lift-and-shift (Cloud Infrastructure-Ready) migration. Here, the on-premises workload is rehosted to a cloud-based VM, without changes. This approach uses the IaaS (Infrastructure as a Service) model. Azure includes several tools such as (Azure Migrate, Azure Site Recovery, and Azure Database Migration Service) to make such a move easier. While this strategy can yield some cost savings, such applications typically weren't architected to unlock and leverage the benefits of cloud computing.

Monolithic apps that are critical to the business oftentimes benefit from an enhanced lift-and-shift (*Cloud Optimized*) migration. This approach includes deployment optimizations that enable key cloud services - without changing the core architecture of the application. For example, you might containerize the application and deploy it to a container orchestrator, like Azure Kubernetes Services, discussed later in this book. Once in the cloud, the application could consume other cloud services such as databases, message queues, monitoring, and distributed caching.

Finally, monolithic apps that perform strategic enterprise functions might best benefit from a *Cloud-Native* approach, the subject of this book. This approach provides agility and velocity. But, it comes at a cost of replatforming, rearchitecting, and rewriting code.

If you and your team believe a cloud-native approach is appropriate, it behooves you to rationalize the decision with your organization. What exactly is the business problem that a cloud- native approach will solve? How would it align with business needs?

- Rapid releases of features with increased confidence?
- Fine-grained scalability - more efficient usage of resources?
- Improved system resiliency?
- Improved system performance?
- More visibility into operations?
- Blend development platforms and data stores to arrive at the best tool for the job?
- Future-proof application investment?

The right migration strategy depends on organizational priorities and the systems you're targeting. For many, it may be more cost effective to cloud-optimize a monolithic application or add coarse-grained services to an N-Tier app. In these cases, you can still make full use of cloud PaaS capabilities like the ones offered by Azure App Service.

Summary

In this chapter, we introduced cloud-native computing. We provided a definition along with the key capabilities that drive a cloud-native application. We looked the types of applications that might justify this investment and effort.

With the introduction behind, we now dive into a much more detailed look at cloud native.

References

- Cloud Native Computing Foundation
- .NET Microservices: Architecture for Containerized .NET applications
- Modernize existing .NET applications with Azure cloud and Windows Containers
- Cloud Native Patterns by Cornelia Davis
- Beyond the Twelve-Factor Application
- What is Infrastructure as Code
- Uber Engineering's Micro Deploy: Deploying Daily with Confidence
- How Netflix Deploys Code
- Overload Control for Scaling WeChat Microservices
- RayGun - Case Study

CHAPTER 2

Introducing eShopOnContainers reference app

Microsoft, in partnership with leading community experts, has produced a full-featured cloud-native microservices reference application, eShopOnContainers. This application is built to showcase using .NET Core and Docker, and optionally Azure, Kubernetes, and Visual Studio, to build an online storefront.

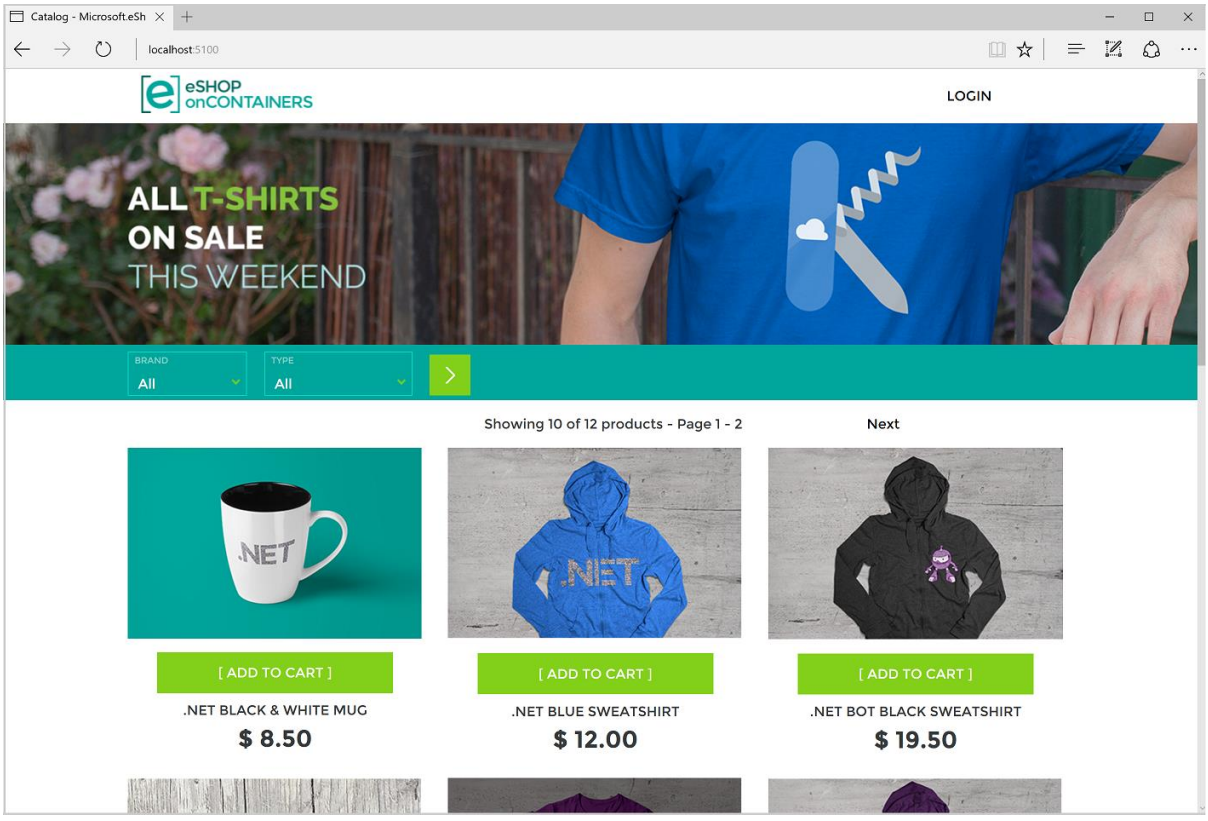


Figure 2-1. eShopOnContainers Sample App Screenshot.

Before starting this chapter, we recommend that you download the eShopOnContainers reference application. If you do so, it should be easier for you to follow along with the information presented.

Features and requirements

Let's start with a review of the application's features and requirements. The eShopOnContainers application represents an online store that sells various physical products like t-shirts and coffee mugs. If you've bought anything online before, the experience of using the store should be relatively familiar. Here are some of the basic features the store implements:

- List catalog items
- Filter items by type
- Filter items by brand
- Add items to the shopping basket
- Edit or remove items from the basket
- Checkout
- Register an account
- Sign in
- Sign out
- Review orders

The application also has the following non-functional requirements:

- It needs to be highly available and it must scale automatically to meet increased traffic (and scale back down once traffic subsides).
- It should provide easy-to-use monitoring of its health and diagnostic logs to help troubleshoot any issues it encounters.
- It should support an agile development process, including support for continuous integration and deployment (CI/CD).
- In addition to the two web front ends (traditional and Single Page Application), the application must also support mobile client apps running different kinds of operating systems.
- It should support cross-platform hosting and cross-platform development.

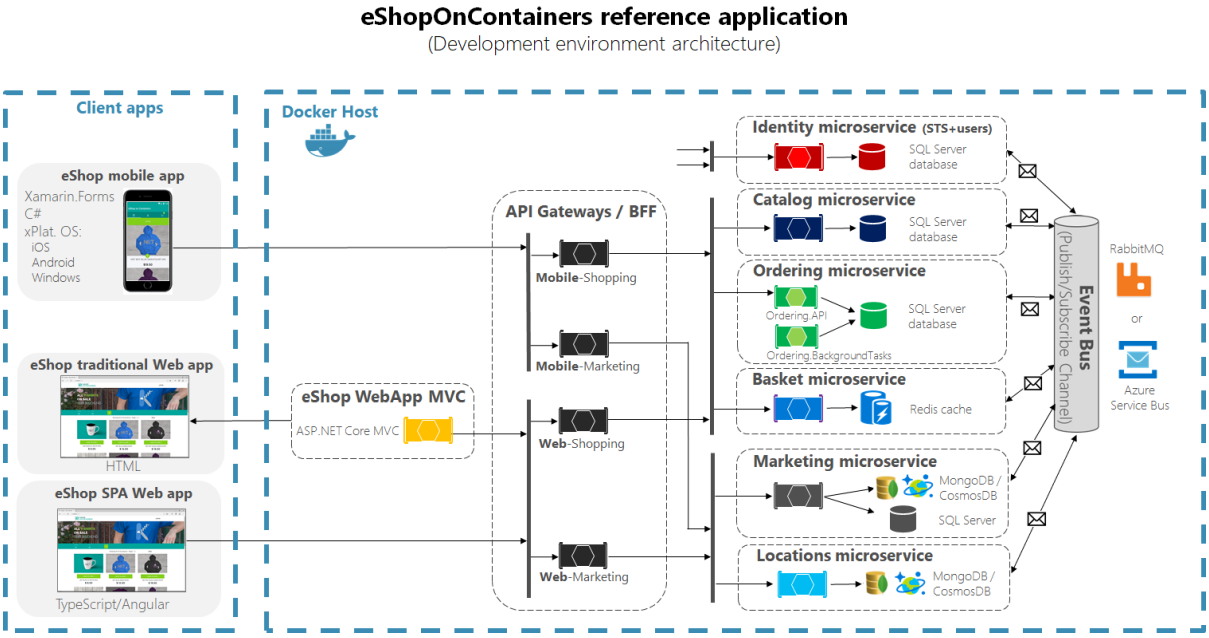


Figure 2-2. eShopOnContainers reference application development architecture.

The eShopOnContainers application is accessible from web or mobile clients that access the application over HTTPS targeting either the ASP.NET Core MVC server application or an appropriate API Gateway. API Gateways offer several advantages, such as decoupling back-end services from individual front-end clients and providing better security. The application also makes use of a related pattern known as Backends-for-Frontends (BFF), which recommends creating separate API gateways for each front-end client. The reference architecture demonstrates breaking up the API gateways based on whether the request is coming from a web or mobile client.

The application's functionality is broken up into a number of distinct microservices. There are services responsible for authentication and identity, listing items from the product catalog, managing users' shopping baskets, and placing orders. Each of these separate services has its own persistent storage. Note that there's no single master data store with which all services interact. Instead, coordination and communication between the services is done on an as-needed basis and through the use of a message bus.

Each of the different microservices is designed differently, based on their individual requirements. This means their technology stack may differ, although they're all built using .NET Core and designed for the cloud. Simpler services provide basic Create-Read-Update-Delete (CRUD) access to the underlying data stores, while more advanced services use Domain-Driven Design approaches and patterns to manage business complexity.

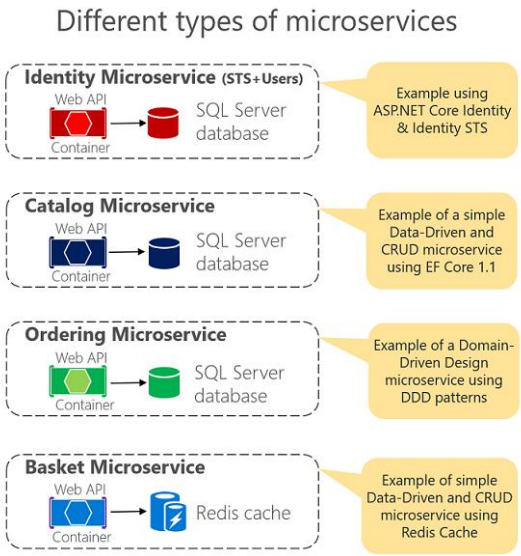


Figure 2-3. Different kinds of microservices.

Overview of the code

Because it leverages microservices, the eShopOnContainers app includes quite a few separate projects and solutions in its GitHub repository. In addition to separate solutions and executable files, the various services are designed to run inside their own containers, both during local development and at runtime in production. Figure 2-4 shows the full Visual Studio solution, in which the various different projects are organized.

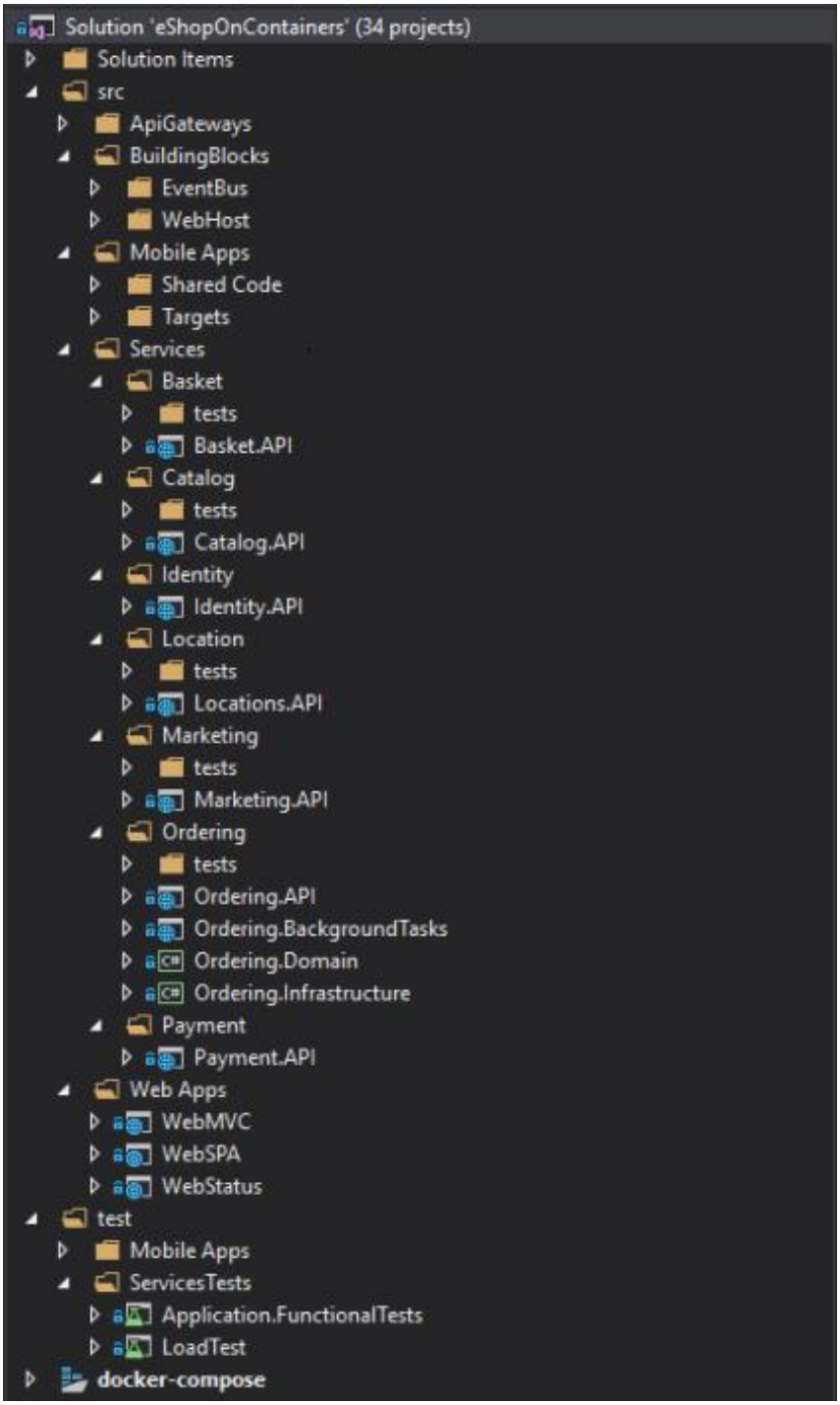


Figure 2-4. Projects in Visual Studio solution.

The code is organized to support the different microservices, and within each microservice, the code is broken up into domain logic, infrastructure concerns, and user interface or service endpoint. In many cases, each service's dependencies can be fulfilled by Azure services in production, as well as alternative options for local development. Let's examine how the application's requirements map to Azure services.

Understanding microservices

This book focuses on cloud-native applications built using Azure technology. To learn more about microservices best practices and how to architect microservice-based applications, read the companion book, .NET Microservices: Architecture for Containerized .NET Applications. The book is available online, in PDF, or eReader formats.

Mapping eShopOnContainers to Azure Services

Although not required, Azure is well-suited to supporting the eShopOnContainers because the project was built to be a cloud-native application. The application is built with .NET Core, so it can run on Linux or Windows containers depending on the Docker host. The application is made up of multiple autonomous microservices, each with its own data. The different microservices showcase different approaches, ranging from simple CRUD operations to more complex DDD and CQRS patterns. Microservices communicate with clients over HTTP and with one another via message-based communication. The application supports multiple platforms for clients as well, since it adopts HTTP as a standard communication protocol and includes ASP.NET Core and Xamarin mobile apps that run on Android, iOS, and Windows platforms.

The application's architecture is shown in Figure 2-5. On the left are the client apps, broken up into mobile, traditional Web, and Web Single Page Application (SPA) flavors. On the right are the server-side components that make up the system, each of which can be hosted in Docker containers and Kubernetes clusters. The traditional web app is powered by the ASP.NET Core MVC application shown in yellow. This app and the mobile and web SPA applications communicate with the individual microservices through one or more API gateways. The API gateways follow the "backends for front ends" (BFF) pattern, meaning that each gateway is designed to support a given front-end client. The individual microservices are listed to the right of the API gateways and include both business logic and some kind of persistence store. The different services make use of SQL Server databases, Redis cache instances, and MongoDB/CosmosDB stores. On the far right is the system's Event Bus, which is used for communication between the microservices.

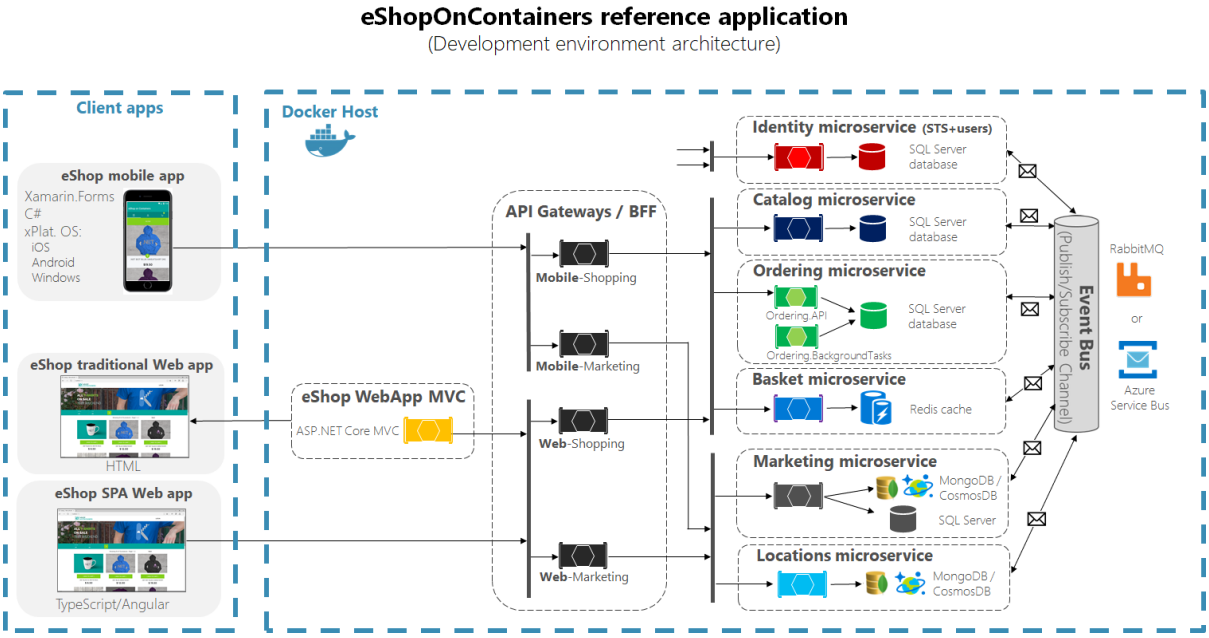


Figure 2-5. The eShopOnContainers Architecture.

The server-side components of this architecture all map easily to Azure services.

Container orchestration and clustering

The application's container-hosted services, from ASP.NET Core MVC apps to individual Catalog and Ordering microservices, can be hosted and managed in Azure Kubernetes Service (AKS). The application can run locally on Docker and Kubernetes, and the same containers can then be deployed to staging and production environments hosted in AKS. This process can be automated as we'll see in the next section.

AKS provides management services for individual clusters of containers. The application will deploy separate AKS clusters for each microservice shown in the architecture diagram above. This approach allows each individual service to each independently according to its resource demands. Each microservice can also be deployed independently, and ideally such deployments should incur zero system downtime.

API Gateway

The eShopOnContainers application has multiple front-end clients and multiple different back-end services. There's no one-to-one correspondence between the client applications and the microservices that support them. In such a scenario, there may be a great deal of complexity when writing client software to interface with the various back-end services in a secure manner. Each client would need to address this complexity on its own, resulting in duplication and many places in which to make updates as services change or new policies are implemented.

Azure API Management (APIM) helps organizations publish APIs in a consistent, manageable fashion. APIM consists of three components: the API Gateway, and administration portal (the Azure portal), and a developer portal.

The API Gateway accepts API calls and routes them to the appropriate back-end API. It can also provide additional services like verification of API keys or JWT tokens and API transformation on the fly without code modifications (for instance, to accommodate clients expecting an older interface).

The Azure portal is where you define the API schema and package different APIs into products. You also configure user access, view reports, and configure policies for quotas or transformations.

The developer portal serves as the main resource for developers. It provides developers with API documentation, an interactive test console, and reports on their own usage. Developers also use the portal to create and manage their own accounts, including subscription and API key support.

Using APIM, applications can expose several different groups of services, each providing a back end for a particular front-end client. APIM is recommended for complex scenarios. For simpler needs, the lightweight API Gateway Ocelot can be used. The eShopOnContainers app uses Ocelot because of its simplicity and because it can be deployed into the same application environment as the application itself. Learn more about eShopOnContainers, APIM, and Ocelot.

Another option if your application is using AKS is to deploy the Azure Gateway Ingress Controller as a pod within your AKS cluster. This allows your cluster to integrate with an Azure Application Gateway, allowing the gateway to load-balance traffic to the AKS pods. Learn more about the Azure Gateway Ingress Controller for AKS.

Data

The various back-end services used by eShopOnContainers have different storage requirements. Several microservices use SQL Server databases. The Basket microservice leverages a Redis cache for its persistence. The Locations microservice expects a MongoDB API for its data. Azure supports each of these data formats.

For SQL Server database support, Azure has products for everything from single databases up to highly scalable SQL Database elastic pools. Individual microservices can be configured to communicate with their own individual SQL Server databases quickly and easily. These databases can be scaled as needed to support each separate microservice according to its needs.

The eShopOnContainers application stores the user's current shopping basket between requests. This is managed by the Basket microservice that stores the data in a Redis cache. In development, this cache can be deployed in a container, while in production it can utilize Azure Cache for Redis. Azure Cache for Redis is a fully managed service offering high performance and reliability without the need to deploy and manage Redis instances or containers on your own.

The Locations microservice uses a MongoDB NoSQL database for its persistence. During development, the database can be deployed in its own container, while in production the service can leverage Azure Cosmos DB's API for MongoDB. One of the benefits of Azure Cosmos DB is its ability to leverage multiple different communication protocols, including a SQL API and common NoSQL APIs including MongoDB, Cassandra, Gremlin, and Azure Table Storage. Azure Cosmos DB offers a fully managed and globally distributed database as a service that can scale to meet the needs of the services that use it.

Distributed data in cloud-native applications is covered in more detail in chapter 5.

Event Bus

The application uses events to communicate changes between different services. This functionality can be implemented with a variety of implementations, and locally the eShopOnContainers application uses RabbitMQ. When hosted in Azure, the application would leverage Azure Service Bus for its messaging. Azure Service Bus is a fully managed integration message broker that allows applications and services to communicate with one another in a decoupled, reliable, asynchronous manner. Azure Service Bus supports individual queues as well as separate *topics* to support publisher-subscriber scenarios. The eShopOnContainers application would leverage topics with Azure Service Bus to support distributing messages from one microservice to any other microservice that needed to react to a given message.

Resiliency

Once deployed to production, the eShopOnContainers application would be able to take advantage of several Azure services available to improve its resiliency. The application publishes health checks, which can be integrated with Application Insights to provide reporting and alerts based on the app's availability. Azure resources also provide diagnostic logs that can be used to identify and correct bugs and performance issues. Resource logs provide detailed information on when and how different Azure resources are used by the application. You'll learn more about cloud-native resiliency features in chapter 6.

References

- The eShopOnContainers Architecture
- Orchestrating microservices and multi-container applications for high scalability and availability
- Azure API Management
- Azure SQL Database Overview
- Azure Cache for Redis
- Azure Cosmos DB's API for MongoDB
- Azure Service Bus
- Azure Monitor overview

Deploying eShopOnContainers to Azure

The logic supporting the eShopOnContainers application can be supported by Azure using a variety of services. The recommended approach is to leverage Kubernetes using Azure Kubernetes Service (AKS). This can be combined with Helm deployment to ensure easily repeated infrastructure configuration. Optionally, developers can leverage Azure Dev Spaces for Kubernetes as part of their development process. Another option is to host the functionality of the app using Azure Serverless features like Azure Functions and Azure Logic Apps.

Azure Kubernetes Service

If you'd like to host the eShopOnContainers application in your own AKS cluster, the first step is to create your cluster. You can do this using the Azure portal, which will walk you through the required steps, or you can use the Azure CLI, taking care to ensure you enable Role-Based Access Control (RBAC) and application routing when you do so. The eShopOnContainers' documentation describes the steps involved in creating your own AKS cluster. Once the cluster is created, you must enable access to the Kubernetes dashboard, at which point you should be able to browse to the Kubernetes dashboard to manage the cluster.

Once the cluster has been created and configured, you can deploy the application to it using Helm and Tiller.

Deploying to Azure Kubernetes Service using Helm

Basic deployments to AKS may use custom CLI scripts or simple deployment files, but more complex applications should use a dependency management tool like Helm. Helm is maintained by the Cloud-native Computing Foundation and helps you define, install, and upgrade Kubernetes applications. Helm is composed of a command-line client, helm, which uses helm charts, and an in-cluster component, Tiller. Helm Charts use standard YAML-formatted files to describe a related set of Kubernetes resources and are typically versioned alongside the application they describe. Helm Charts range from simple to complex depending on the requirements of the installation they describe.

You'll find the eShopOnContainers helm charts in the /k8s/helm folder. Figure 2-6 shows how the different components of the application are organized into a folder structure used by helm to define and managed deployments.

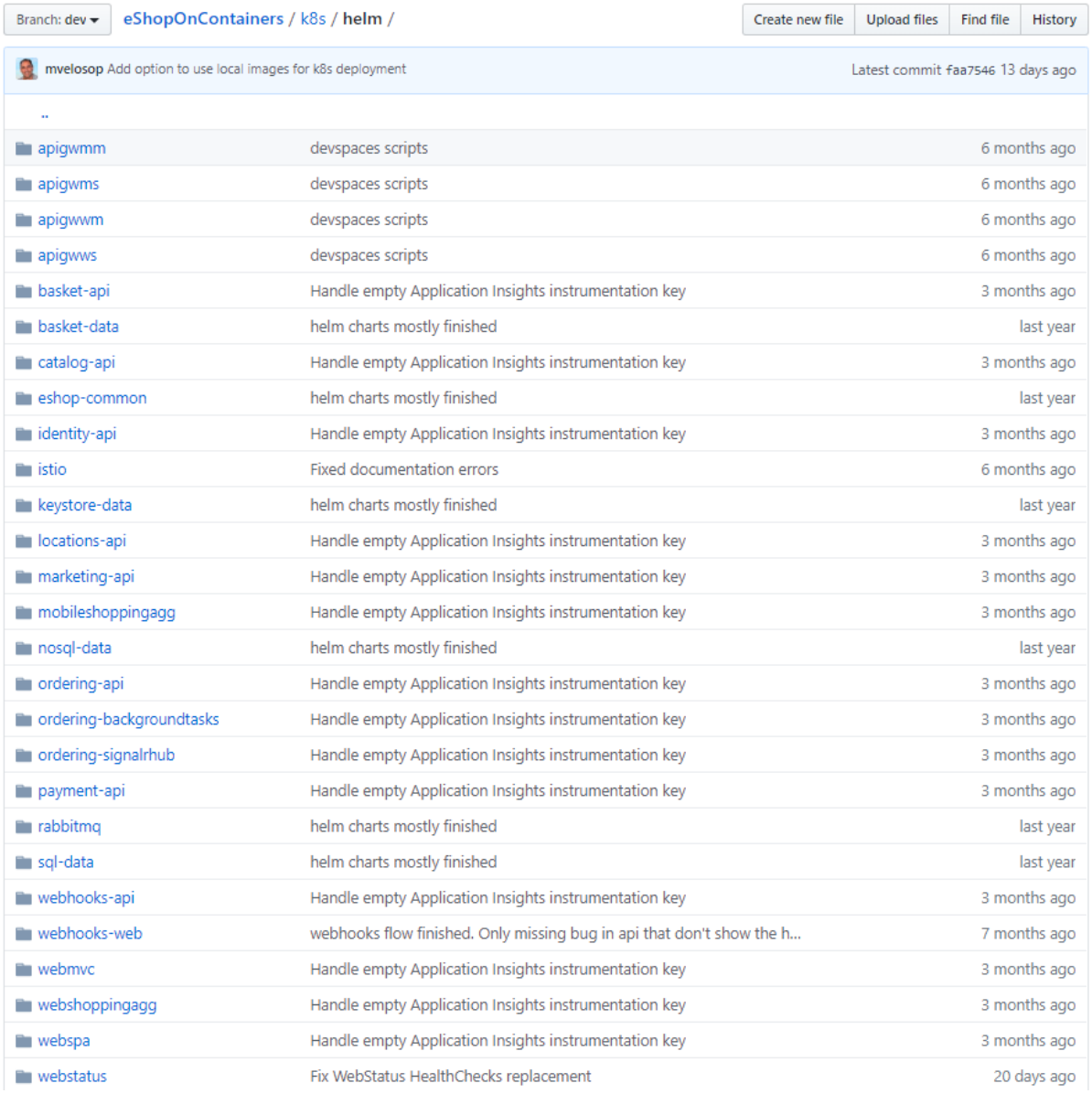


Figure 2-6. The eShopOnContainers helm folder.

Each individual component is installed using a `helm install` command. These commands are easily scripted, and eShopOnContainers provides a "deploy all" script that loops through the different components and installs them using their respective helm charts. The result is a repeatable process, versioned with the application in source control, that anyone on the team can deploy to an AKS cluster with a one-line script command. Especially when combined with Azure Dev Spaces, this makes it easy for developers to diagnose and test their individual changes to their microservice-based cloud-native apps.

Azure Dev Spaces

Azure Dev Spaces helps individual developers host their own unique version of AKS clusters in Azure during development. This minimizes local machine requirements and allows team members to quickly

see how their changes will behave in a real AKS environment. Azure Dev Spaces offers a CLI for developers to use to manage their dev spaces and to deploy to a specific child dev space as needed. Each child dev space is referenced using a unique URL subdomain, allowing side-by-side deployments of modified clusters so that individual developers can avoid conflicting with each other's work in progress. In Figure 2-7 you can see how developer Susie has deployed her own version of the Bikes microservice into her dev space. She's then able to test her changes using a custom URL starting with the name of her space (susie.s.dev.myapp.eus.azds.io).

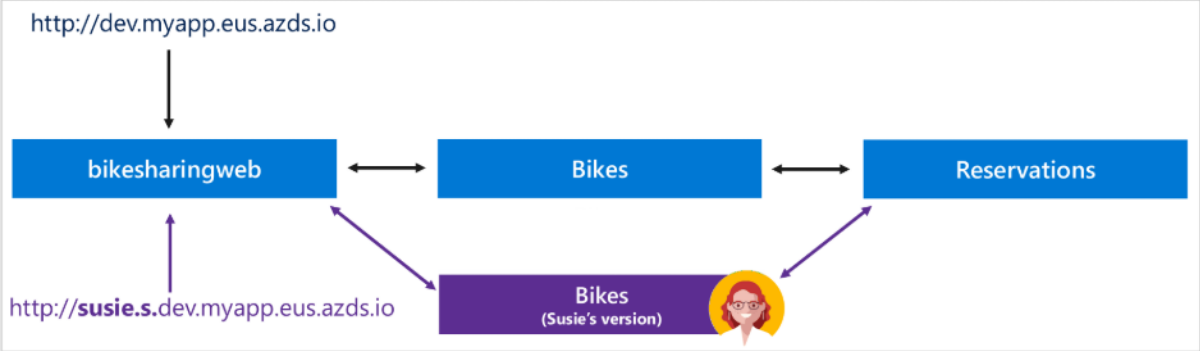


Figure 2-7. Developer Susie deploys her own version of the Bikes microservice and tests it.

At the same time, developer John is customizing the Reservations microservice and needs to test his changes. He's able to deploy his changes to his own dev space without conflicting with Susie's changes as shown in Figure 2-8. He can test his changes using his own URL which is prefixed with the name of his space (john.s.dev.myapp.eus.azds.io).

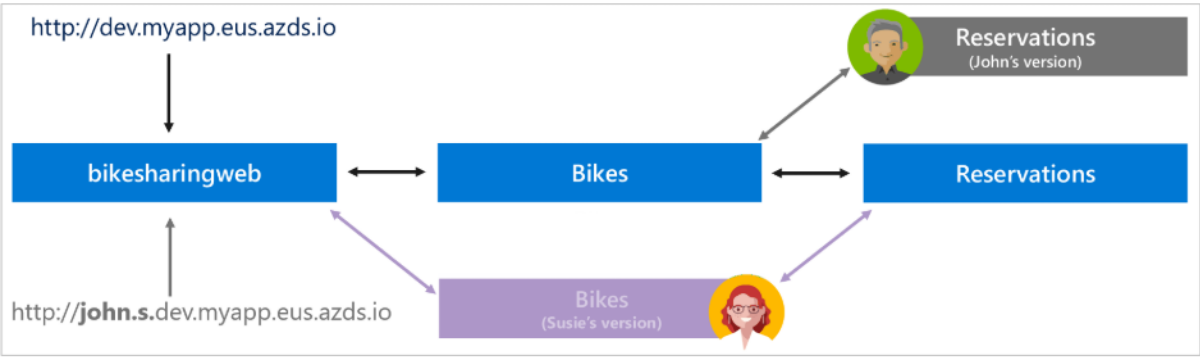


Figure 2-8. Developer John deploys his own version of the Reservations microservice and tests it without conflicting with other developers.

Using Azure Dev Spaces, teams can work directly with AKS while independently changing, deploying, and testing their changes. This approach reduces the need for separate dedicated hosted environments since every developer effectively has their own AKS environment. Developers can work with Azure Dev Spaces using its CLI or launch their application to Azure Dev Spaces directly from Visual Studio. Learn more about how Azure Dev Spaces works and is configured.

Azure Functions and Logic Apps (Serverless)

The eShopOnContainers sample includes support for tracking online marketing campaigns. An Azure Function is used to pull marketing campaign details for a given campaign ID. Rather than creating a complete ASP.NET Core application for this purpose, a single Azure Function endpoint is simpler and sufficient. Azure Functions have a much simpler build and deployment model than full ASP.NET Core applications, especially when configured to run in Kubernetes. Deploying the function is scripted using Azure Resource Manager (ARM) templates and the Azure CLI. This campaign details microservice isn't customer-facing and doesn't have the same requirements as the online store, making it a good candidate for Azure Functions. The function requires some configuration to work properly, such as database connection string data and image base URI settings. You configure Azure Functions in the Azure Portal.

References

- eShopOnContainers: Create Kubernetes cluster in AKS
- eShopOnContainers: Azure Dev Spaces
- Azure Dev Spaces

Centralized configuration

Cloud-native applications involve many more running services than traditional single-instance monolithic apps. Managing configuration settings for dozens of interdependent services can be challenging, which is why centralized configuration stores are often implemented for distributed applications.

As discussed in Chapter 1, the Twelve-Factor App recommendations require strict separation between code and configuration. This means storing configuration settings as constants or literal values in code is a violation. This recommendation exists because the same code should be used across multiple environments, including development, testing, staging, and production. However, configuration values are likely to vary between each of these environments. So, configuration values should be stored in the environment itself, or the environment should store the credentials to a centralized configuration store.

The eShopOnContainers application includes local application settings files with each microservice. These files are checked into source control but don't include production secrets such as connection strings or API keys. In production, individual settings may be overwritten with per-service environment variables. This is a common practice for hosted applications, but doesn't provide a central configuration store. To support centralized management of configuration settings, each microservice includes a setting to toggle between its use of local settings or Azure Key Vault settings.

Azure Key Vault

Azure Key Vault provides secure storage of tokens, passwords, certificates, API keys, and other sensitive secrets. Access to Key Vault requires proper caller authentication and authorization, which in the case of the eShopOnContainers microservices means the use of a ClientId/ClientSecret

combination. Don't check in these credentials into source control, but instead set in the application's environment. Direct access to Key Vault from AKS can be achieved using Key Vault FlexVolume.

With centralized configuration, settings that apply to the entire application, such as the centralized logging endpoint, can be set once and used by every part of the distributed application. Although microservices should be independent of one another, there will typically still be some shared dependencies whose configuration details can benefit from a centralized configuration store.

CHAPTER 3

Scaling cloud-native applications

One of the most-often touted advantages of moving to a cloud hosting environment is scalability. Scalability, or the ability for an application to accept additional user load without unduly degrading performance for each user, is most often achieved by breaking up applications into small pieces that can each be given whatever resources they require. In this chapter, we introduce the technologies that enable cloud-native applications to scale to meet user demand. These technologies include:

- Containers
- Orchestrators
- Serverless computing

Leveraging containers and orchestrators

Containers and orchestrators are designed to solve problems common to monolithic deployment approaches.

Challenges with monolithic deployments

Traditionally, most applications have been deployed as a single unit. Such applications are referred to as a monolith. This general approach of deploying applications as single units even if they're composed of multiple modules or assemblies is known as monolithic architecture, as shown in Figure 3-1.

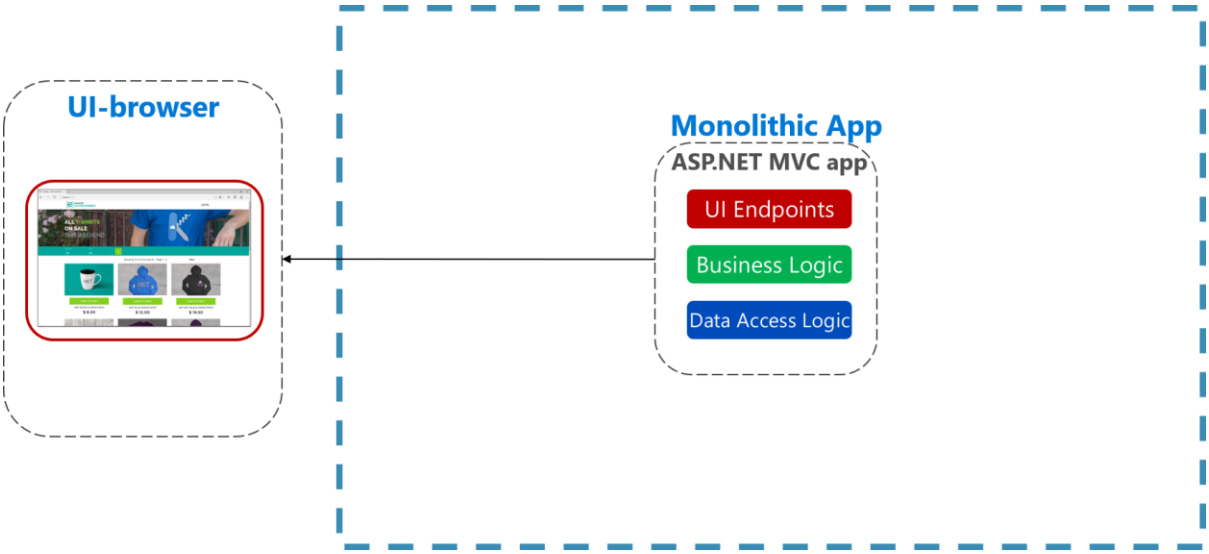


Figure 3-1. Monolithic architecture.

Although they have the benefit of simplicity, monolithic architectures face a number of challenges:

Deployments

Deploying to monolithic applications typically requires restarting the entire application, even if only one small module is being replaced. Depending on the number of machines hosting the application, this can result in downtime during deployments.

Hosting

Monolithic applications are hosted entirely on a single machine instance. This may require higher-capability hardware than any module in a distributed application would need. Also, if any part of the app becomes a bottleneck, the entire application must be deployed to additional machine nodes in order to scale out.

Environment

Monolithic applications are typically deployed into an existing hosting environment (operating system, installed frameworks, etc.). This environment may not match the environment in which the application was developed or tested. Inconsistencies in the application's environment are a common source of problems for monolithic deployments.

Coupling

Monolithic applications are likely to have a great deal of coupling between different parts of the application, and between the application and its environment. This can make it difficult to factor out a particular service or concern later, in order to increase its scalability or swap in an alternative implementation. This coupling also leads to much larger potential impacts for changes to the system, requiring extensive testing in larger applications.

Technology choice

Monolithic applications are built and deployed as a unit. This offers simplicity and uniformity but can be a barrier to innovation. Although a new feature or module in the system might be better-suited to a more modern platform or framework, it's likely to be built using the application's current approach for the sake of consistency as well as ease of development and deployment.

What are the benefits of containers and orchestrators?

Docker is the most popular container management and imaging platform and allows you to quickly work with containers on Linux and Windows. Containers provide separate but reproducible application environments that run the same way on any system. This makes them perfect for developing and hosting applications and app components in cloud-native applications. Containers are isolated from one another, so two containers on the same host hardware can have different versions of software and even operating system installed, without the dependencies causing conflicts.

What's more, containers are defined by simple files that can be checked into source control. Unlike full servers, even virtual machines, which frequently require manual work to apply updates or install additional services, container infrastructure can easily be version-controlled. Thus, apps built to run in containers can be developed, tested, and deployed using automated tools as part of a build pipeline.

Containers are immutable. Once you have the definition of a container, you can recreate that container and it will run exactly the same way. This immutability lends itself to component-based design. If some parts of an application don't change as often as others, why redeploy the entire app when you can just deploy the parts that change most frequently? Different features and cross-cutting concerns of an app can be broken up into separate units. Figure 3-2 shows how a monolithic app can take advantage of containers and microservices by delegating certain features or functionality. The remaining functionality in the app itself has also been containerized.

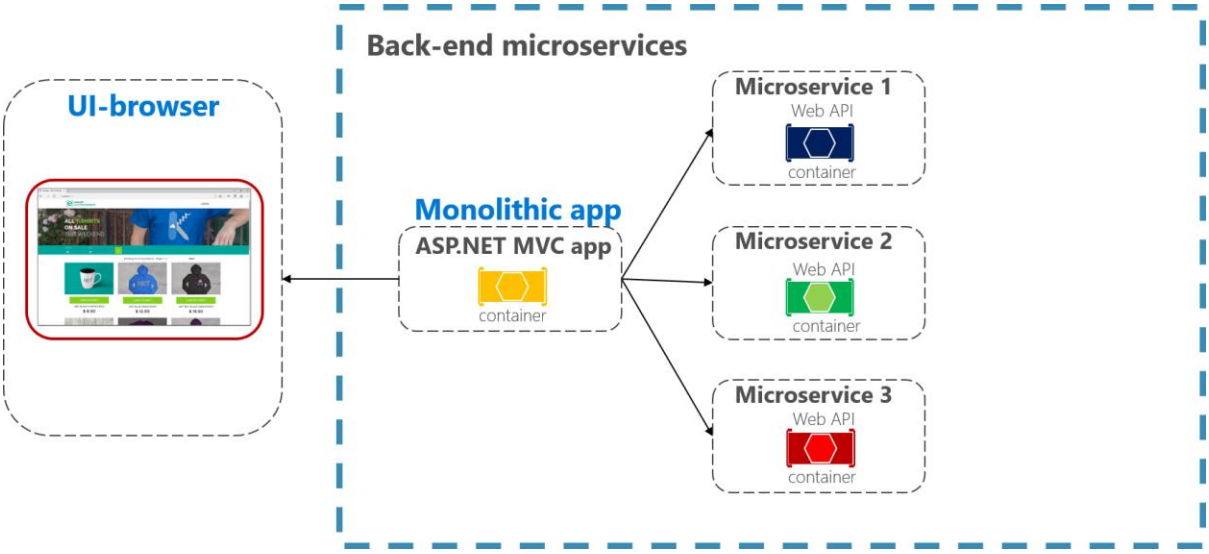


Figure 3-2. Breaking up a monolithic app to use microservices in the back end.

Cloud-native apps built using separate containers benefit from the ability to deploy as much or as little of an application as needed. Individual services can be hosted on nodes with resources

appropriate to each service. The environment each service runs in is immutable, can be shared between dev, test, and production, and can easily be versioned. Coupling between different areas of the application occurs explicitly as calls or messages between services, not compile-time dependencies within the monolith. And any given part of the overall app can choose the technology that makes the most sense for that feature or capability without requiring changes to the rest of the app.

What are the scaling benefits?

Services built on containers can leverage scaling benefits provided by orchestration tools like Kubernetes. By design containers only know about themselves. Once you start to have multiple containers that need to work together, it can be worthwhile to organize them at a higher level. Organizing large numbers of containers and their shared dependencies, such as network configuration, is where orchestration tools come in to save the day! Kubernetes is a container orchestration platform designed to automate deployment, scaling, and management of containerized applications. It creates an abstraction layer on top of groups of containers and organizes them into *pods*. Pods run on worker machines referred to as *nodes*. The whole organized group is referred to as a *cluster*. Figure 3-3 shows the different components of a Kubernetes cluster.

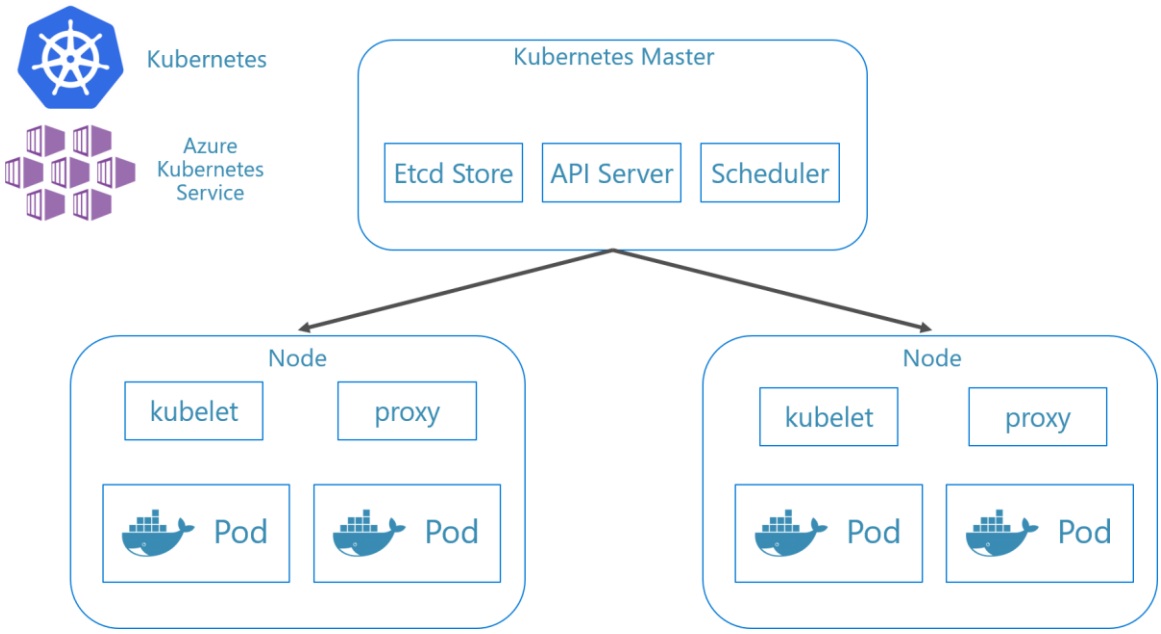


Figure 3-3. Kubernetes cluster components.

Kubernetes has built-in support for scaling clusters to meet demand. Combined with containerized micro-services, this provides cloud-native applications with the ability to quickly and efficiently respond to spikes in demand with additional resources when and where they're needed.

Declarative versus imperative

Kubernetes supports both declarative and imperative object configuration. The imperative approach involves running various commands that tell Kubernetes what to do each step of the way. *Run* this image. *Delete* this pod. *Expose* this port. With the declarative approach, you use a configuration file

that describes *what you want* instead of *what to do* and Kubernetes figures out what to do to achieve the desired end state. If you've already configured your cluster using imperative commands, you can export a declarative manifest by using `kubectl get svc SERVICENAME -o yaml > service.yaml`. This will produce a manifest file like this one:

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: "2019-09-13T13:58:47Z"
  labels:
    component: apiserver
    provider: kubernetes
  name: kubernetes
  namespace: default
  resourceVersion: "153"
  selfLink: /api/v1/namespaces/default/services/kubernetes
  uid: 9b1fac62-d62e-11e9-8968-00155d38010d
spec:
  clusterIP: 10.96.0.1
  ports:
  - name: https
    port: 443
    protocol: TCP
    targetPort: 6443
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

When using declarative configuration, you can preview the changes that will be made before committing them by using `kubectl diff -f FOLDERNAME` against the folder where your configuration files are located. Once you're sure you want to apply the changes, run `kubectl apply -f FOLDERNAME`. Add `-R` to recursively process a folder hierarchy.

In addition to services, you can use declarative configuration for other Kubernetes features, such as *deployments*. Declarative deployments are used by deployment controllers to update cluster resources. Deployments are used to roll out new changes, scale up to support more load, or roll back to a previous revision. If a cluster is unstable, declarative deployments provide a mechanism for automatically bringing the cluster back to a desired state.

Using declarative configuration allows infrastructure to be represented as code that can be checked in and versioned alongside the application code. This provides improved change control and better support for continuous deployment using a build and deploy pipeline tied to source control changes.

What scenarios are ideal for containers and orchestrators?

The following scenarios are ideal for using containers and orchestrators.

Applications requiring high uptime and scalability

Individual applications that have high uptime and scalability requirements are ideal candidates for cloud-native architectures using microservices, containers, and orchestrators. These applications can be developed in containers using versioned environments, can be extensively tested before going to

production, and can be deployed to production with zero downtime. The use of Kubernetes clusters ensures such apps can also scale on demand and recover automatically from node failures.

Large numbers of applications

Organizations that deploy and must subsequently maintain large numbers of applications benefit from containers and orchestrators. The up front effort of setting up containerized environments and Kubernetes clusters is primarily a fixed cost. Deploying, maintaining, and updating individual applications has a cost that varies with the number of applications that must be maintained. Beyond a certain fairly small number of applications, the complexity of maintaining custom applications manually exceeds the cost of implementing a solution using containers and orchestrators.

When should you avoid using containers and orchestrators?

If you're unwilling or unable to build your application following Twelve-Factor App principles, you'll probably be better off avoiding containers and orchestrators. In these cases, it may be best to move forward with a VM-based hosting platform, or possibly some hybrid system in which you can spin off certain pieces of functionality into separate containers or even serverless functions.

Development resources

This section shows a short list of development resources that may help you get started using containers and orchestrators for your next application. If you're looking for guidance on how to design your cloud-native microservices architecture app, read this book's companion, .NET Microservices: Architecture for Containerized .NET Applications.

Local Kubernetes Development

Kubernetes deployments provide great value in production environments, but you can also run them locally. Although much of the time it's good to be able to work on individual apps or microservices independently, sometimes it's good to be able to run the whole system locally just as it will run when deployed to production. There are several ways to achieve this, two of which are Minikube and Docker Desktop. Visual Studio also provides tooling for Docker development.

Minikube

What is Minikube? The Minikube project says "Minikube implements a local Kubernetes cluster on macOS, Linux, and Windows." Its primary goals are "to be the best tool for local Kubernetes application development and to support all Kubernetes features that fit." Installing Minikube is separate from Docker, but Minikube supports different hypervisors than Docker Desktop supports. The following Kubernetes features are currently supported by Minikube:

- DNS
- NodePorts
- ConfigMaps and secrets
- Dashboards
- Container runtimes: Docker, rkt, CRI-O, and containerd

- Enabling Container Network Interface (CNI)
- Ingress

After installing Minikube, you can quickly start using it by running the `minikube start` command, which downloads an image and start the local Kubernetes cluster. Once the cluster is started, you interact with it using the standard Kubernetes `kubectl` commands.

Docker Desktop

You can also work with Kubernetes directly from Docker Desktop on Windows. This is your only option if you're using Windows Containers, and is a great choice for non-Windows containers as well. The standard Docker Desktop configuration app is used to configure Kubernetes running from Docker Desktop.

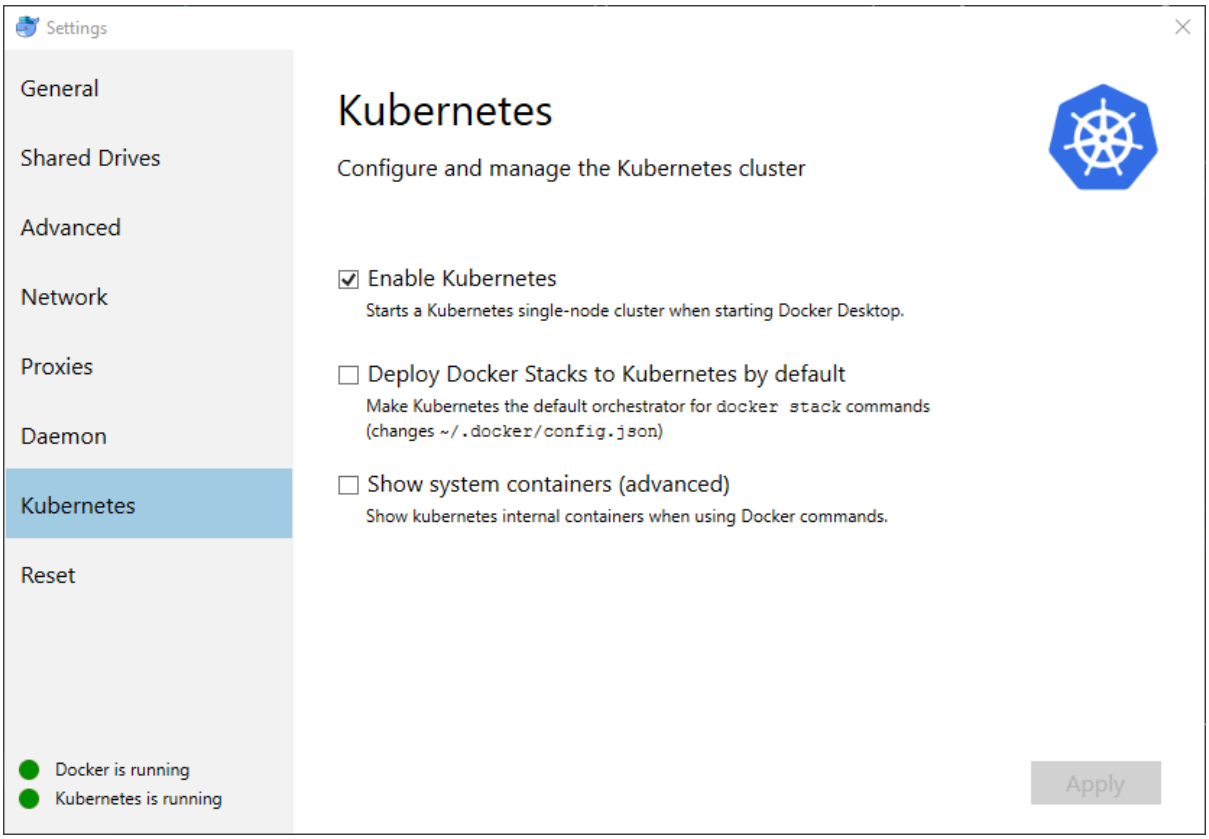


Figure 3-4. Configuring Kubernetes in Docker Desktop.

Docker Desktop is already the most popular tool for configuring and running containerized apps locally. When you work with Docker Desktop, you can develop locally against the exact same set of Docker container images that you'll deploy to production. Docker Desktop is designed to "build, test, and ship" containerized apps locally. Once the images have been shipped to an image registry like Azure Container Registry or Docker Hub, then services like Azure Kubernetes Service (AKS) manage the application in production.

Visual Studio Docker Tooling

Visual Studio supports Docker development for web applications. When you create a new ASP.NET Core application, you're given the option to configure it with Docker support as part of the project creation process, as shown in Figure 3-5.

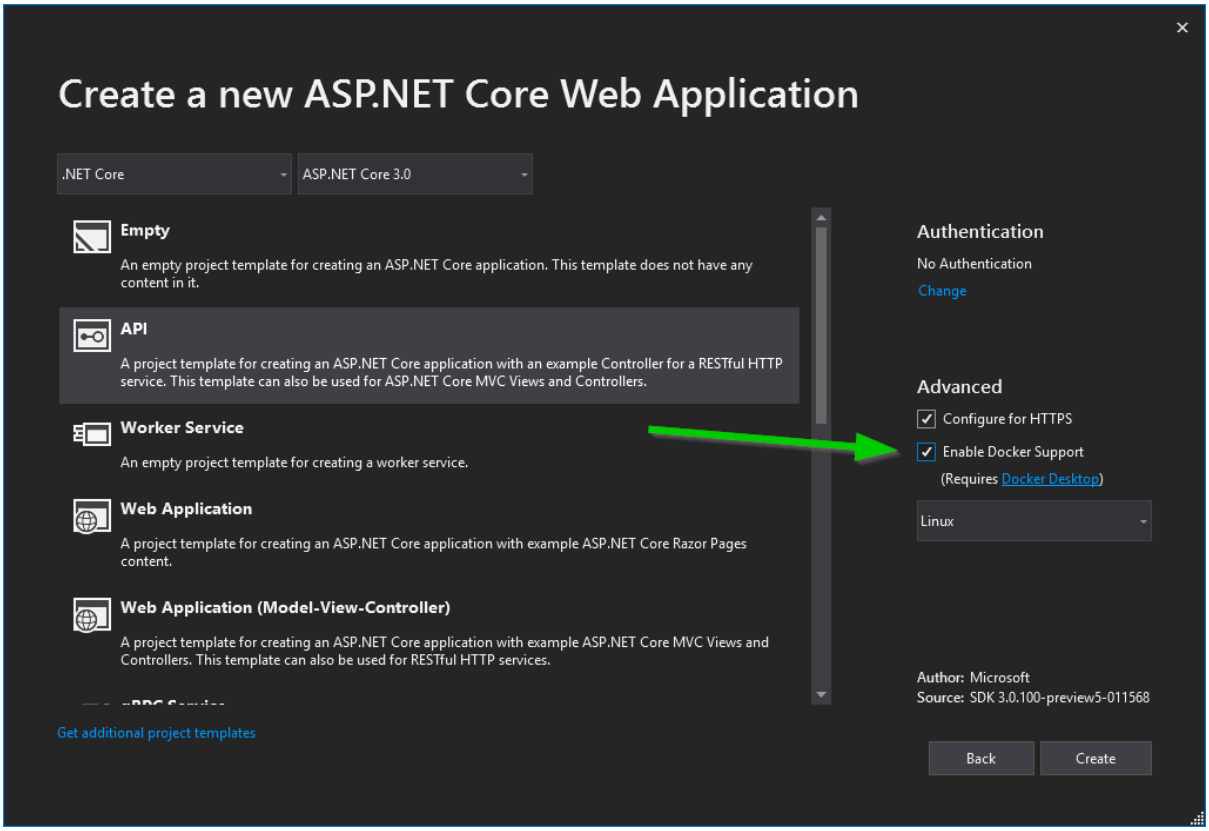


Figure 3-5. Visual Studio Enable Docker Support

When this option is selected, the project is created with a `Dockerfile` in its root, which can be used to build and host the app in a Docker container. An example Dockerfile is shown in Figure 3-6.

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.0-stretch-slim AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/core/sdk:3.0-stretch AS build
WORKDIR /src
COPY ["WebApplication3/WebApplication3.csproj", "WebApplication3/"]
RUN dotnet restore "WebApplication3/WebApplication3.csproj"
COPY . .
WORKDIR "/src/WebApplication3"
RUN dotnet build "WebApplication3.csproj" -c Release -o /app

FROM build AS publish
RUN dotnet publish "WebApplication3.csproj" -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "WebApplication3.dll"]
```

Figure 3-6. Visual Studio generated Dockerfile

The default behavior when the app runs is configured to use Docker as well. Figure 3-7 shows the different run options available from a new ASP.NET Core project created with Docker support added.

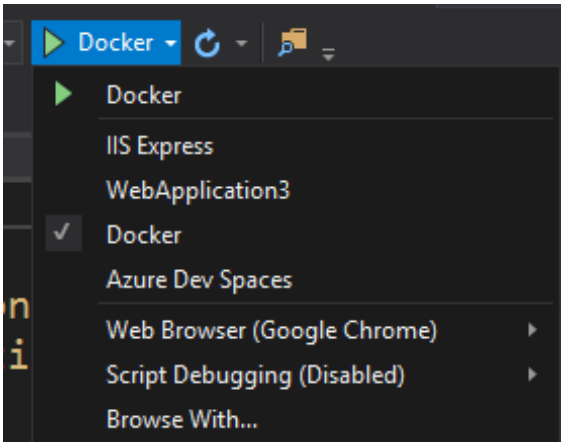


Figure 3-7. Visual Studio Docker Run Options

In addition to local development, Azure Dev Spaces provides a convenient way for multiple developers to work with their own Kubernetes configurations within Azure. As you can see in Figure 3-10, you can also run the application in Azure Dev Spaces.

If you don't add Docker support to your ASP.NET Core application when you create it, you can always add it later. From the Visual Studio Solution Explorer, right click on the project and select **Add** > **Docker Support**, as shown in Figure 3-8.

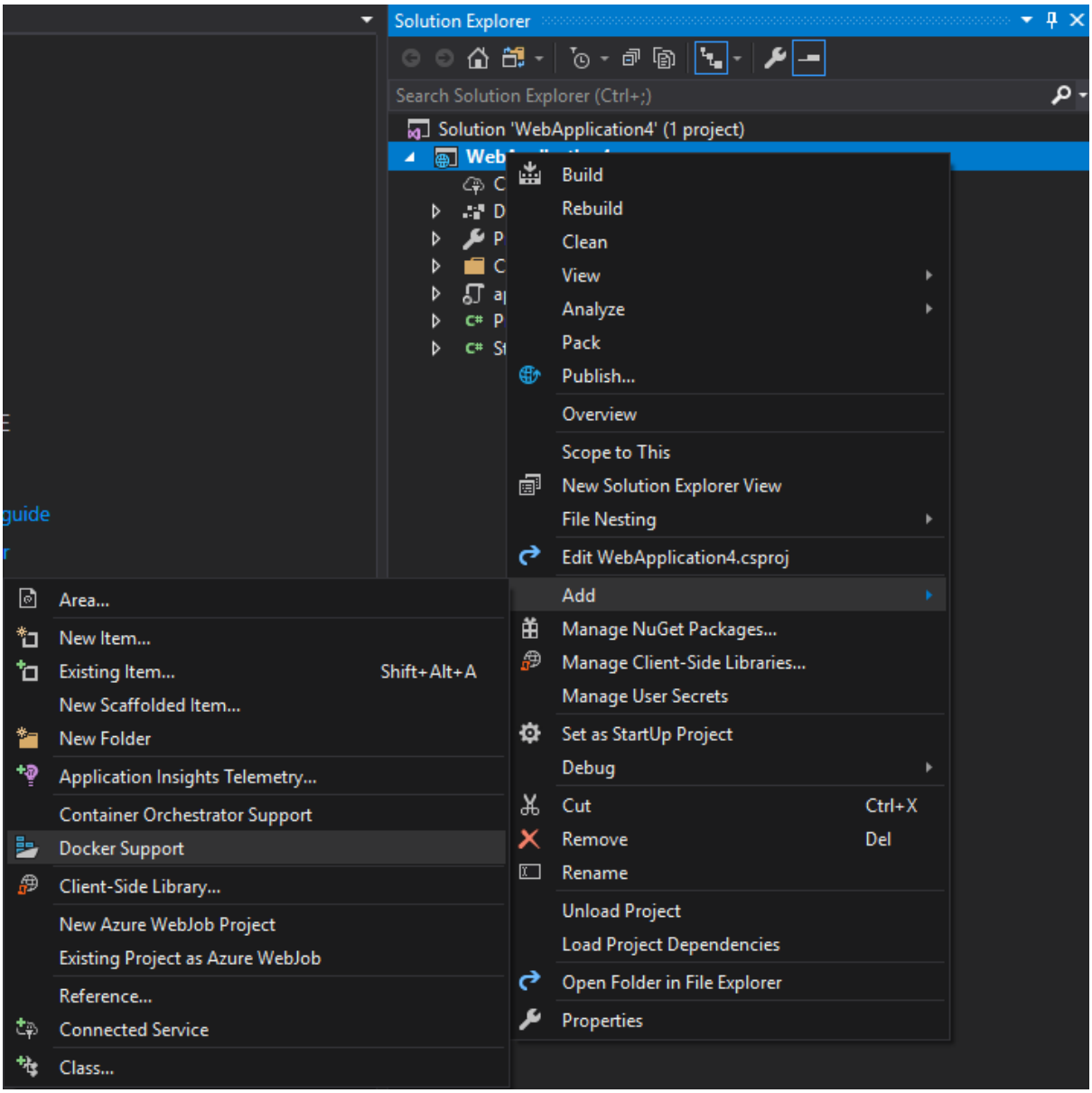


Figure 3-8. Visual Studio Add Docker Support

In addition to Docker support, you can also add Container Orchestration Support, also shown in Figure 3-11. By default, the orchestrator uses Kubernetes and Helm. Once you've chosen the orchestrator, a `azds.yaml` file is added to the project root and a `charts` folder is added containing the Helm charts used to configure and deploy the application to Kubernetes. Figure 3-9 shows the resulting files in a new project.

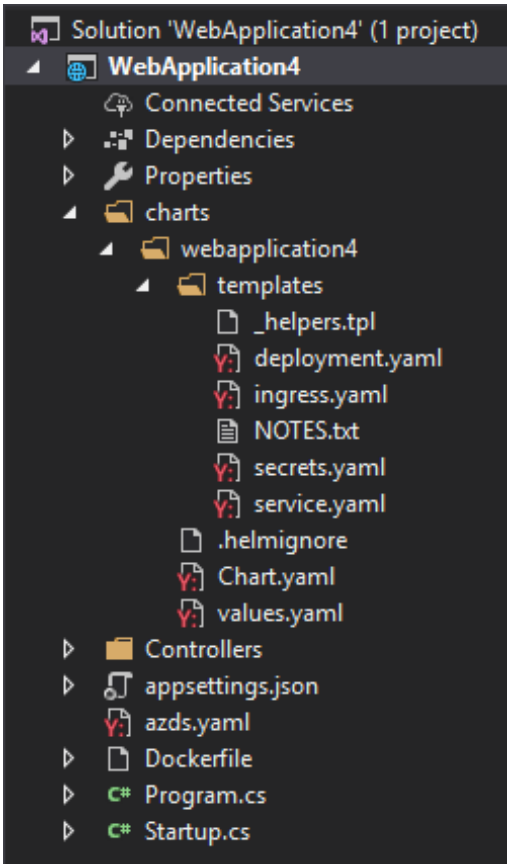


Figure 3-9. Visual Studio Add Orchestrator Support

References

- What is Kubernetes?
- Installing Kubernetes with Minikube
- MiniKube vs Docker Desktop
- Visual Studio Tools for Docker

Leveraging serverless functions

In the spectrum of managing full machines and operating systems to leveraging cloud capabilities, serverless lives at the extreme end where the only thing you're responsible for is your code, and you only pay for your code when it runs. Azure Functions provides a way to build serverless capabilities into your applications.

What is serverless?

Serverless computing doesn't mean there isn't a server involved in running your application - the code still runs on a server somewhere. The distinction is that the application development team no longer needs to concern themselves with managing server infrastructure. Serverless computing solutions like

Azure Functions help teams increase their productivity and allow organizations to optimize their resources and focus on delivering solutions.

Serverless computing uses event-triggered stateless containers to host your application or part of your application. Serverless platforms can scale up and down to meet demand as-needed. Platforms like Azure Functions have easy direct access to other Azure services like queues, events, and storage.

What challenges are solved by serverless?

Serverless is the ultimate abstraction away from running your own hardware. Developers can focus exclusively on writing code to solve business problems, without concern for any of the following tasks that might have been necessary when hosting your own servers:

- Purchasing machines and software licenses
- Housing, securing, configuring, and maintaining the machines and their networking, power, and A/C requirements
- Patching and upgrading operating systems and software
- Configuring web servers or machine services to host application software
- Configuring application software within its platform

Many companies employ dozens of staff members and allocate large budgets to support these hardware infrastructure concerns. Simply moving to the cloud eliminates some of these concerns; shifting applications all the way to serverless will eliminate the rest.

What scenarios are appropriate for serverless?

Serverless uses individual short-running functions that are called in response to some trigger. This makes them ideal for processing background tasks.

For example, an application might need to send an email as part of processing a request. Instead of sending the email as part of handling the web request, the details of the email could be placed onto a queue and an Azure Function could be used to pick up the message and send the email. Many different parts of the application, or even many applications, could leverage this same Azure Function, providing improved performance and scalability for the applications and using queue-based load leveling to avoid bottlenecks related to sending the emails.

Although a Publisher/Subscriber pattern between applications and Azure Functions is the most common pattern, other patterns are possible. Azure Functions can be triggered by other events, such as changes to Azure Blob Storage. An application that supported image uploads could have an Azure Function responsible for creating thumbnail images, or resizing uploaded images to consistent dimensions, or optimizing image size. All of this functionality could be triggered directly by inserts to Azure Blob Storage, keeping the complexity and the workload out of the application itself.

Many applications have long-running processes as part of their workflows. Often these tasks are done as part of the user's interaction with the application, forcing the user to wait and negatively impacting their experience. Serverless computing provides a great way to perform slower tasks outside of the user interaction loop, and these tasks can easily scale with demand without requiring the entire application to scale.

When should you avoid serverless?

Serverless computing is best-used for tasks that don't block the user interface. This means they're not ideal for hosting web applications or web APIs directly. The main reason for this is that serverless solutions are provisioned and scaled on demand. When a new instance of a function is needed, referred to as a *cold start*, it takes time to provision. This time is typically a few seconds, but can be longer depending on a variety of factors. A single instance can often be maintained indefinitely (for instance, by periodically making a request to it), but the cold start issue remains if the number of instances ever needs to scale up.

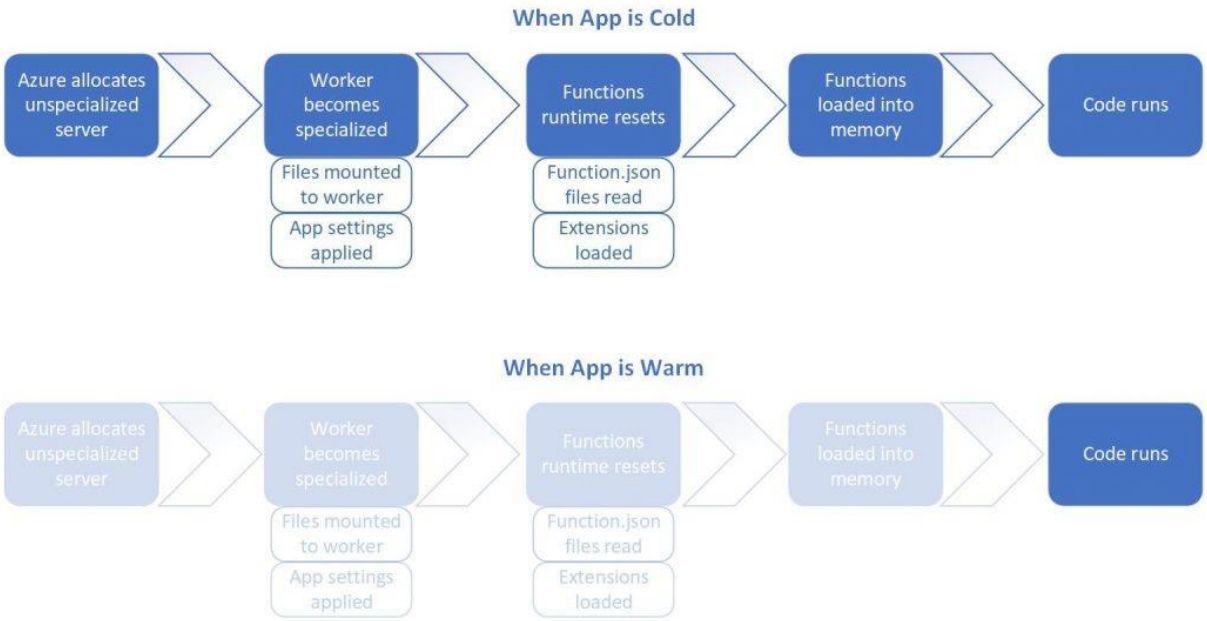


Figure 3-10. Cold start versus warm start.

If you need to avoid cold starts entirely, you can choose to switch from a consumption plan to a dedicated plan. You can also configure one or more pre-warmed instances with the premium plan so when you need to add another instance, it's already up and ready to go. These options can mitigate one of the key concerns associated with serverless computing.

You should also typically avoid serverless for long-running tasks. They're best for small pieces of work that can be completed quickly. Most serverless platforms require individual functions to complete within a few minutes. Azure Functions defaults to a 5-minute time-out duration (can be configured up to 10 minutes). The Azure Functions premium plan can mitigate this issue as well, defaulting time outs to 30 minutes and allowing an unbounded higher limit to be configured.

Finally, leveraging serverless for certain tasks within your application adds complexity. It's often best to architect your application in a modular, loosely coupled manner first, and then identify if there are benefits serverless would offer that make the additional complexity worthwhile. Many smaller applications will run perfectly well in a single monolithic deployment, without the need for the distributed application architecture serverless computing requires.

References

- Understanding serverless cold start
- Pre-warmed Azure Functions instances
- Create a function on Linux using a custom image

Combining containers and serverless approaches

Frequently, microservices-based applications rely heavily on containers, orchestration, and message-passing for communication between nodes. This messaging is an ideal task for Azure Functions, but with the rest of the application configured and deployed using Kubernetes and related tools, it would be nice to be able to leverage Azure Functions within this same toolset. Fortunately, you can wrap Azure Functions in Docker containers and deploy them using the same processes and tools as the rest of your Kubernetes-based app.

When does it make sense to use containers with serverless?

By default, your serverless functions have no knowledge of the platform on which they'll run. However, in some cases you may have specific requirements that make it important for you to customize the container image in which your code will run. You may want to use a custom image because your function relies on a specific language version or has dependencies or configuration requirements that aren't supported by the default image. In these cases, it may make sense to customize your own container and deploy your function in a custom Docker container.

When should you avoid using containers with Azure Functions?

If you're expecting to benefit from the consumption plan billing for your function, you won't be able to do so if you're using your own container. What's more, if you go beyond just using a Docker container and deploy your functions to your own Kubernetes cluster, you'll no longer benefit from the built-in scaling provided by Azure Functions. You'll need to use Kubernetes' scaling features, described below.

How to combine serverless and Docker containers

To wrap an Azure Function in a Docker container, install the Azure Functions Core Tools and then run the following command:

```
func init ProjectName --docker
```

Choose which worker runtime you want from the following options:

- `dotnet` (C#)
- `node` (JavaScript)
- `python`

When the project is created, it will include a Dockerfile. Now, you can create and test your function locally. Build and run it using the `docker build` and `docker run` commands. For detailed steps to get

started building Azure Functions with Docker support, see the Create a function on Linux using a custom image tutorial.

How to combine serverless and Kubernetes with KEDA

Azure functions scale automatically to meet demand based on the rate of events that are targeting a given function. Additionally, you can leverage Kubernetes to host your functions and use Kubernetes-based Event Driven Autoscaling, or KEDA. When no events are occurring, KEDA can scale down to 0 instances, and then in response to events it can scale up the number of containers to meet the demand using its horizontal pod autoscaler. Learn more about scaling Azure functions with KEDA.

References

- Run Azure Functions in a Docker Container
- Create a function on Linux using a custom image
- Azure Functions with Kubernetes Event Driven Autoscaling

Deploying containers in Azure

Containers provide many benefits, one of which is portability. You can easily take the same container you've developed and tested locally and deploy it to Azure where it can run your app in staging and production environments. Azure provides a number of options for container-based app hosting and likewise supports several different means of deployment. The most common and most flexible approach is to deploy your containers to Azure Container Registry (ACR), where they're accessible by whatever services you wish to use to host them. Azure Web App for Containers, Azure Kubernetes Services (AKS), and Azure Container Instance (ACI) all can access container images that have been pushed to ACR.

Azure Container Registry

Azure Container Registry (ACR) lets you build, store, and manage images for all of your container deployments. There are other container registries, both public and private, to which you can deploy containers. The benefit of ACR over other options is that you can keep your images close to your production environment, improving build and deployment times. You can also secure them using the same security procedures you use for the rest of your Azure resources, improving security and reducing asset management effort.

You create a container registry using the Azure Portal or using the Azure CLI or PowerShell tools. Creating a new container registry just requires an Azure subscription, a resource group, and a unique name. Figure 3-11 shows the basic options for creating a registry, which will be hosted at *registryname*.azurecr.io.

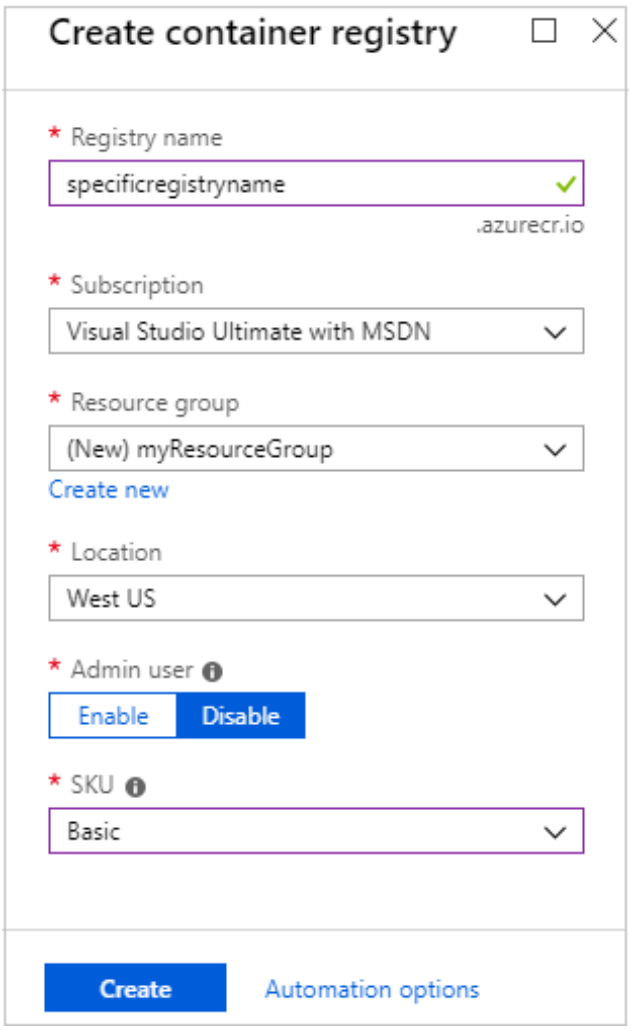


Figure 3-11. Create container registry

Once you've created a registry, you'll need to authenticate with it before you can use it. Typically, you'll log into the registry using the Azure CLI command:

```
az acr login --name *registryname*
```

Once you've created a registry in Azure Container Registry, you can use docker commands to push container images to it. Before you can do so, however, you must first tag your image with the fully qualified name (URL) of your ACR login server. This will have the format *registryname*.azurecr.io.

```
docker tag mycontainer myregistry.azurecr.io/mycontainer:v1
```

After you've tagged the image, you use the `docker push` command to push the image to your ACR instance.

```
docker push myregistry.azurecr.io/mycontainer:v1
```

After you push an image to the registry, it's a good idea to remove the image from your local Docker environment, using this command:

```
docker rmi myregistry.azurecr.io/mycontainer:v1
```

Developers should rarely push directly from their machines to a container registry. Instead, a build pipeline defined in a tool like Azure DevOps should be responsible for this process. Learn more in the Cloud-Native DevOps chapter.

Azure Kubernetes Service

If your container-based application involves multiple containers, you'll most likely want to define and manage the interactions between the containers using an *orchestrator* like Kubernetes. Once you've deployed your container images to ACR, you can easily configure Azure Kubernetes Services to automatically deploy updated images from ACR. With a full CI/CD pipeline in place, you can configure a canary release strategy to minimize the risk involved when rapidly deploying updates. The new version of the app is initially configured in production with no traffic routed to it, and then a small number of users are routed to the newly-deployed version of the app. As the team gains confidence in the new version of the software, more instances of the new version are rolled out and the previous version's instances are retired. AKS easily supports this style of deployment.

As with most resources in Azure, you can create Azure Kubernetes clusters using the portal or using command line tools or infrastructure automation tools like Helm or Terraform. To get started with a new cluster, you need to provide the following information:

- Azure subscription
- Resource group
- Kubernetes cluster name
- Region
- Kubernetes version
- DNS name prefix
- Node size
- Node count

This information is sufficient to get started. As part of the creation process in the Azure Portal, you can also configure options for the following features of your cluster:

- Scale
- Authentication
- Networking
- Monitoring
- Tags

This quickstart walks through deploying an AKS cluster using the Azure portal.

Azure Dev Spaces

Complex Kubernetes clusters can require significant resources to host, which can make it difficult for developers to run the entire application on a single machine (especially a laptop). Azure Dev Spaces offers a solution to this by allowing developers to work with their own versions of Azure Kubernetes clusters hosted in Azure. Azure Dev Spaces is designed to ease development of microservice-based applications using AKS.

To understand the value of Azure Dev Spaces, let me share this quotation from Gabe Monroy, PM Lead of Containers at Microsoft Azure:

"Imagine you are a new employee trying to fix a bug in a complex microservices application consisting of dozens of components, each with their own configuration and backing services. To get started, you must configure your local development environment so that it can mimic production including setting up your IDE, building tool chain, containerized service dependencies, a local Kubernetes environment, mocks for backing services, and more. With all the time involved setting up your development environment, fixing that first bug could take days.

Or you could use Dev Spaces and AKS."

The process for working with Azure Dev Spaces involves the following steps:

1. Create the dev space.
2. Configure the root dev space.
3. Configure a child dev space (for your own version of the system).
4. Connect to the dev space.

All of these steps can be performed using the Azure CLI and new `azds` command line tools. For example, to create a new Azure Dev Space for a given Kubernetes cluster, you would use a command like this one:

```
az aks use-dev-spaces -g my-aks-resource-group -n MyAKSCluster
```

Next, you can use the `azds prep` command to generate the necessary Docker and Helm chart assets for running the application. Then you run your code in AKS using `azds up`. The first time you run this command, the Helm chart will be installed, and the container(s) will be built and deployed according to your instructions. This may take a few minutes the first time it's run. However, after you make changes, you can connect to your own child dev space using `azds space select` and then deploy and debug your updates in your isolated child dev space. Once you have your dev space up and running, you can send updates to it by re-issuing the `azds up` command or you can use built-in tooling in Visual Studio or Visual Studio Code. With VS Code, you use the command palette to connect to your dev space. Figure 3-12 shows how to launch your web application using Azure Dev Spaces in Visual Studio.

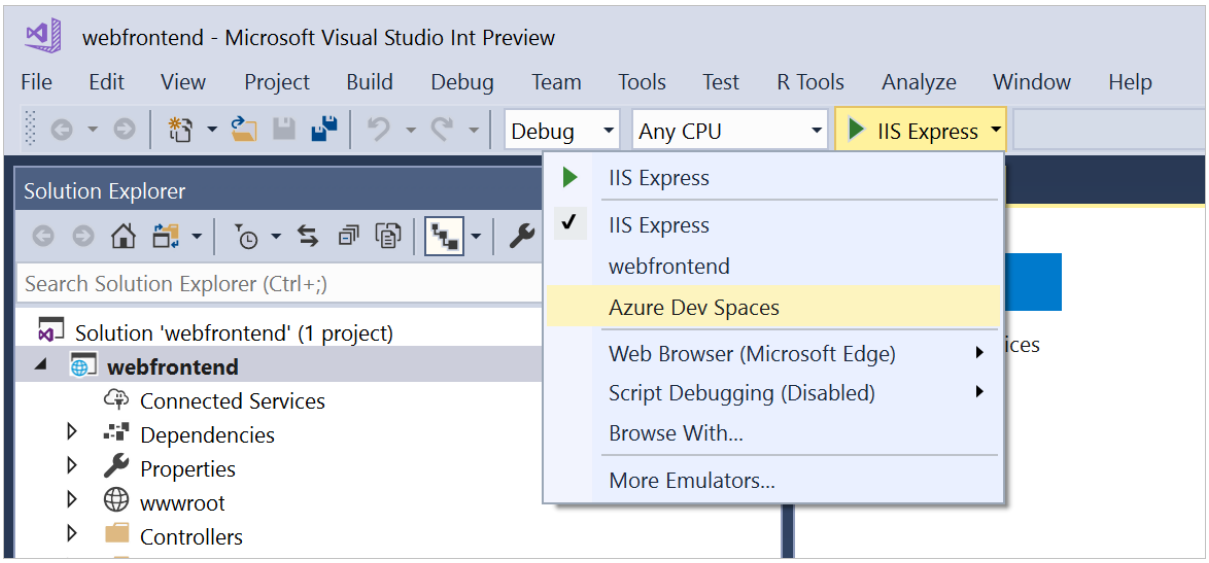


Figure 3-12. Connect to Azure Dev Spaces in Visual Studio

References

- Canary Release
- Azure Dev Spaces with VS Code
- Azure Dev Spaces with Visual Studio

Scaling containers and serverless applications

There are two typical ways to scale an application: scaling up and scaling out. The former refers to adding capabilities to one host, while the latter refers to adding to the total number of hosts. A common analogy to use to think about this is how to get yourself and some friends across town. If it's just one friend, you could hop into your two-seat race car. But if it's three or four, you might need to take one of your SUVs or a minivan, scaling up to increase capacity. When your total number jumps up to a dozen or more, though, you probably need to take multiple vehicles (unless someone drives a bus), which demonstrates the concept of scaling out by adding more instances (in this case, more vehicles). Let's see how this applies to our applications.

The simple solution: scaling up

The process of upgrading existing servers to give them more resources (CPU, memory, disk I/O speed, network I/O speed) is known as *scaling up*. In cloud-native applications, scaling up doesn't typically refer to purchasing and installing actual hardware on physical machines so much as choosing a more capable plan from a list of options available. Cloud-native apps typically scale up by modifying the virtual machine (VM) size used to host the individual nodes in their Kubernetes node pool. Kubernetes concepts like nodes, clusters, and pods are described further in the next section. Azure supports a wide variety of VM sizes running both Windows and Linux. To vertically scale your application, create a new node pool with a larger node VM size and then migrate workloads to the new pool. This requires multiple node pools for your AKS cluster, a feature that is currently in preview. Serverless apps scale

up by choosing a premium plan and premium instance sizes or by choosing a different dedicated app service plan.

Scaling out cloud-native apps

Cloud-native apps support scaling out by adding additional nodes or pods to service requests. This can be accomplished manually by adjusting configuration settings for the app (for example, scaling a node pool), or through *autoscaling*. Autoscaling adjusts the resources used by an app in order to respond to demand, similar to how a thermostat responds to temperature by calling for additional heating or cooling. When using autoscaling, manual scaling is disabled.

AKS clusters can scale in one of two ways:

- The cluster autoscaler watches for pods that can't be scheduled on nodes because of resource constraints. It adds additional nodes as required.
- The **horizontal pod autoscaler** uses the Metrics Server in a Kubernetes cluster to monitor the resource demands of pods. If a service needs more resources, the autoscaler increases the number of pods.

Figure 3-13 shows the relationship between these two scaling services.

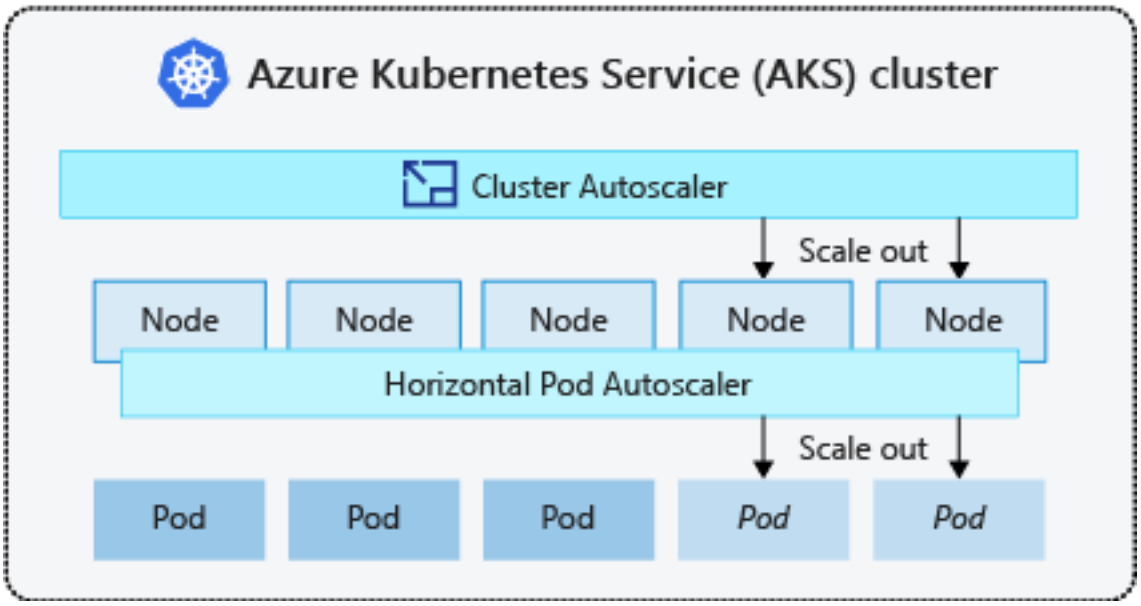


Figure 3-13. Scaling out an App Service plan.

These services can also reduce the number of pods or nodes as needed. These two services can work together and are often deployed together in a cluster. When combined, the horizontal pod autoscaler is focused on running the number of pods required to meet application demand. The cluster autoscaler is focused on running the number of nodes required to support the scheduled pods.

Scaling Azure Functions

Azure Functions automatically supports scaling out. The default consumption plan adds (and removes) resources dynamically based on the number of triggering events coming in. You're only charged for

compute resources being used when your functions are running based on the number of executions, execution time, and memory used. Using the premium plan, you get these same features but you can also control the instance sizes that are used, have instances already warmed up (to avoid cold start delays), and configure dedicated VMs on which to run your functions. While the default configuration should provide an economical and scalable solution for most apps, the premium option allows developers flexibility for custom Azure Functions requirements.

References

- AKS Multiple Node Pools
- AKS Cluster Autoscaler
- Tutorial: Scale applications in AKS
- Azure Functions scale and hosting

Other container deployment options

In addition to deploying to AKS, you can also deploy containers to Azure App Service for Containers and Azure Container Instances.

When does it make sense to deploy to App Service for Containers?

Simple production applications that don't require orchestration are well-suited to Azure App Service for Containers.

How to deploy to App Service for Containers

To deploy to Azure App Service for Containers, you need to have configured an Azure Container Registry (ACR) and credentials for accessing it. Push the container you intend to host to the registry so it's available to pull into your Azure App Service. Once created, you can configure the app for Continuous Deployment, which will automatically deploy updates to the app whenever you update its corresponding image in ACR.

When does it make sense to deploy to Azure Container Instances?

Azure Container Instances are best used for testing scenarios. They provide a fast, simple way to deploy an application to a cloud-hosted container instance. Use them to test or demo applications when you don't require scaling and orchestration features offered by Azure Kubernetes Service.

How to deploy an app to Azure Container Instances

To deploy to Azure Container Instances (ACI), you need to have configured an Azure Container Registry (ACR) and credentials for accessing it. You must also have previously pushed your container image to the registry, so it's available to pull into ACI. You can work with ACI using the Azure CLI or through the portal. Azure Container Registries make it easy to deploy individual container instances to ACI directly from within the registry, as shown in Figure 3-14.

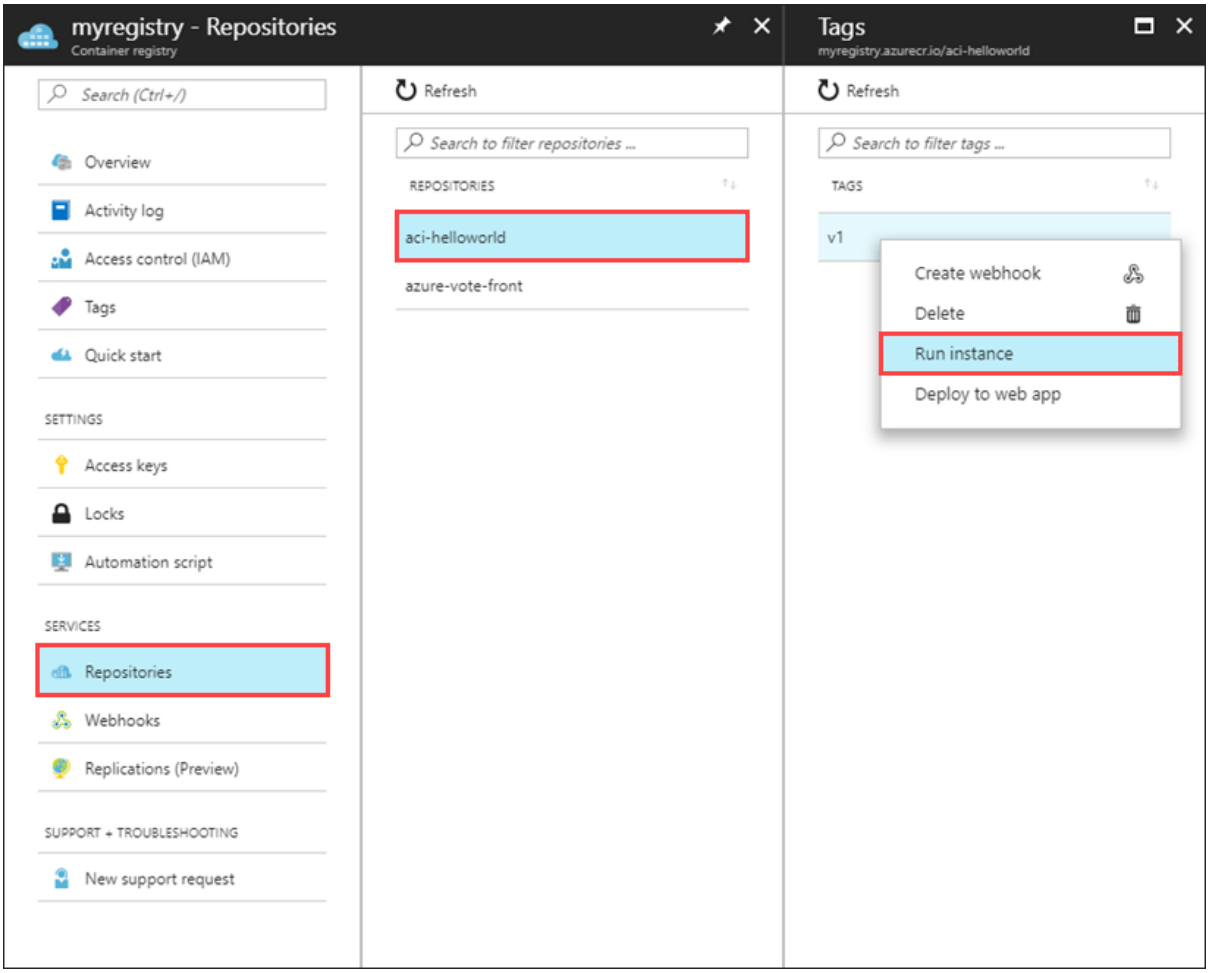


Figure 3-14. Azure Container Registry Run Instance

Creating a container instance from the registry just requires you to specify the usual Azure settings (name, subscription, resource group, and location), how much memory to allocate to the container, and which port it should listen on. This quickstart shows how to deploy a container instance to ACI using the Azure portal.

Once the deployment completes, find the newly deployed container's IP address and communicate with it over the port you specified.

Azure Container Instances offers the fastest, simplest way to run a container in Azure. There's no need to configure an app service or an orchestrator or to deal with virtual machines. However, because of its simplicity, ACI should primarily be used for testing purposes. If your application requires automatic scalability, multiple containers configured to work together, or any additional complex features, there are other better-suited Azure services available to host your app.

References

- Azure Container Instances Docs
- Deploy Container Instance from ACR

CHAPTER 4

Cloud-native communication patterns

When constructing a cloud-native system, communication becomes a significant design decision. How does a front-end client application communicate with a back-end microservice? How do back-end microservices communicate with each other? What are the principles, patterns, and best practices to consider when implementing communication in cloud-native applications?

Communication considerations

In a monolithic application, communication is straightforward. The code modules execute together in the same executable space (process) on a server. This approach can have performance advantages as everything runs together in shared memory, but results in tightly coupled code that becomes difficult to maintain, evolve, and scale.

Cloud-native systems implement a microservice-based architecture with many small, independent microservices. Each microservice executes in a separate process and typically runs inside a container that is deployed to a *cluster*.

A cluster groups a pool of virtual machines together to form a highly available environment. They're managed with an orchestration tool, which is responsible for deploying and managing the containerized microservices. Figure 4-1 shows a Kubernetes cluster deployed into the Azure cloud with the fully managed Azure Kubernetes Services.

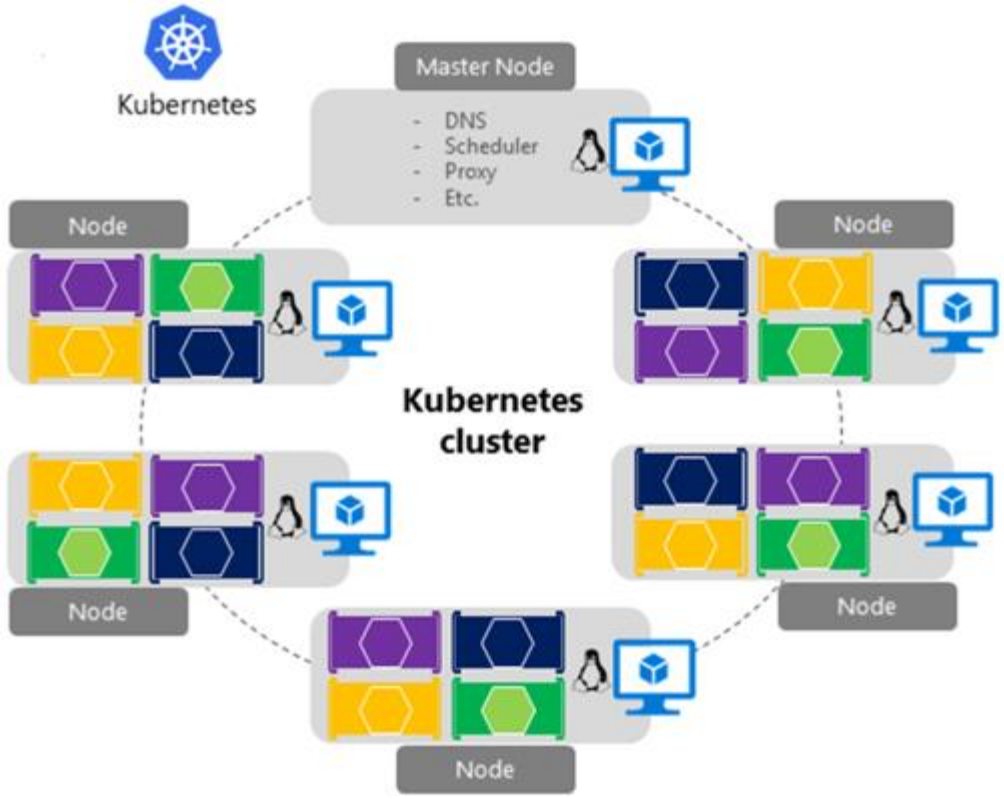


Figure 4-1. A Kubernetes cluster in Azure

Across the cluster, microservices communicate with each other through APIs and messaging technologies.

While they provide many benefits, microservices are no free lunch. Local in-process method calls between components are now replaced with network calls. Each microservice must communicate over a network protocol, which adds complexity to your system:

- Network congestion, latency, and transient faults are a constant concern.
- Resiliency (that is, retrying failed requests) is essential.
- Some calls must be idempotent as to keep consistent state.
- Each microservice must authenticate and authorize calls.
- Each message must be serialized and then deserialized - which can be expensive.
- Message encryption/decryption becomes important.

The book .NET Microservices: Architecture for Containerized .NET Applications, available for free from Microsoft, provides an in-depth coverage of communication patterns for microservice applications. In this chapter, we provide a high-level overview of these patterns along with implementation options available in the Azure cloud.

In this chapter, we'll first address communication between front-end applications and back-end microservices. We'll then look at back-end microservices communicate with each other. We'll explore

the up and gRPC communication technology. Finally, we'll look new innovative communication patterns using service mesh technology. We'll also see how the Azure cloud provides different kinds of *backing services* to support cloud-native communication.

Front-end client communication

In a cloud-native system, front-end clients (mobile, web, and desktop applications) require a communication channel to interact with independent back-end microservices.

What are the options?

To keep things simple, a front-end client could *directly communicate* with the back-end microservices, shown in Figure 4-2.

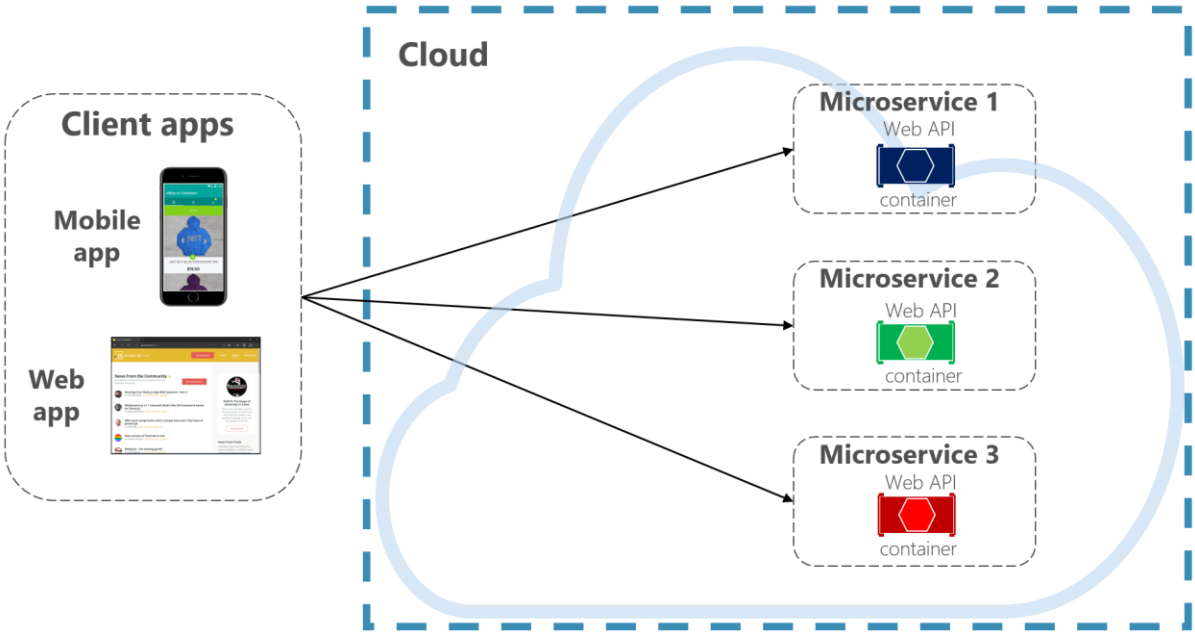


Figure 4-2. Direct client to service communication

With this approach, each microservice has a public endpoint that is accessible by front-end clients. In a production environment, you'd place a load balancer in front of the microservices, routing traffic proportionately.

While simple to implement, direct client communication would be acceptable only for simple microservice applications. This pattern tightly couples front-end clients to core back-end services, opening the door for a number of problems, including:

- Client susceptibility to back-end service refactoring.
- A wider attack surface as core back-end services are directly exposed.
- Duplication of cross-cutting concerns across each microservice.
- Overly complex client code - clients must keep track of multiple endpoints and handle failures in a resilient way.

Instead, a widely accepted cloud design pattern is to implement an API Gateway Service between the front-end applications and back-end services. The pattern is shown in Figure 4-3.

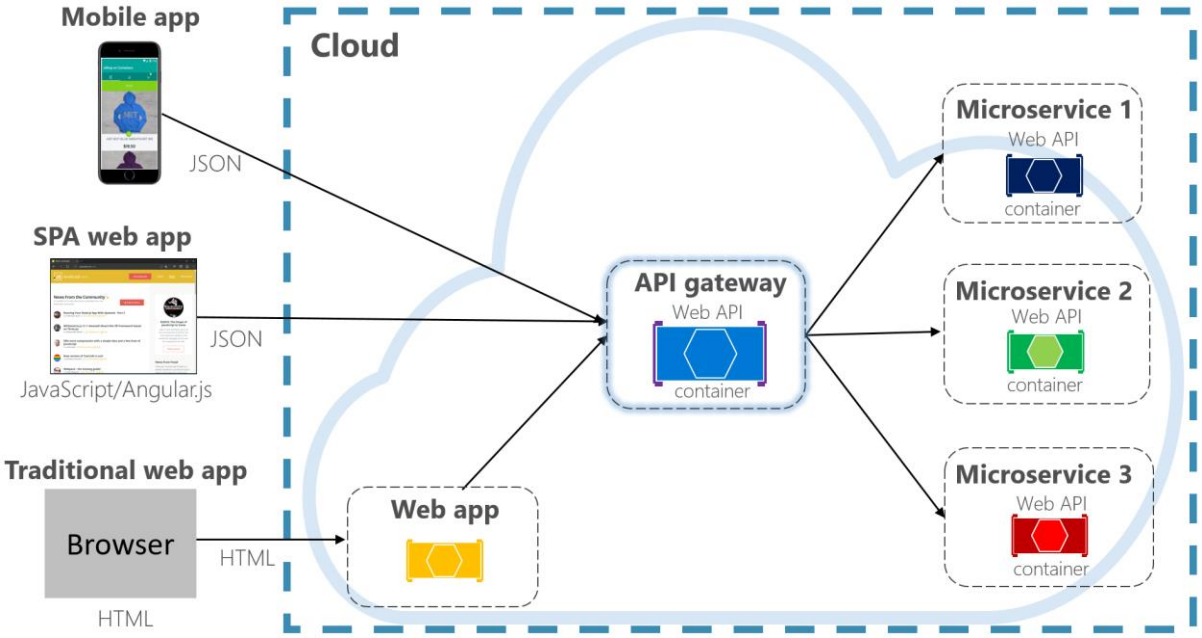


Figure 4-3. API gateway pattern

In the previous figure, note how the API Gateway service abstracts the back-end core microservices. Implemented as a web API, it acts as a *reverse proxy*, routing incoming traffic to the internal microservices.

The gateway insulates the client from internal service partitioning and refactoring. If you change a back-end service, you accommodate for it in the gateway without breaking the client. It's also your first line of defense for cross-cutting concerns, such as identity, caching, resiliency, metering, and throttling. Many of these cross-cutting concerns can be off-loaded from the back-end core services to the gateway, simplifying the back-end services.

Care must be taken to keep the API Gateway simple and fast. Typically, business logic is kept out of the gateway. A complex gateway risks becoming a bottleneck and eventually a monolith itself. Larger systems often expose multiple API Gateways segmented by client type (mobile, web, desktop) or back-end functionality. The Backend for Frontends pattern provides direction for implementing multiple gateways. The pattern is shown in Figure 4-4.

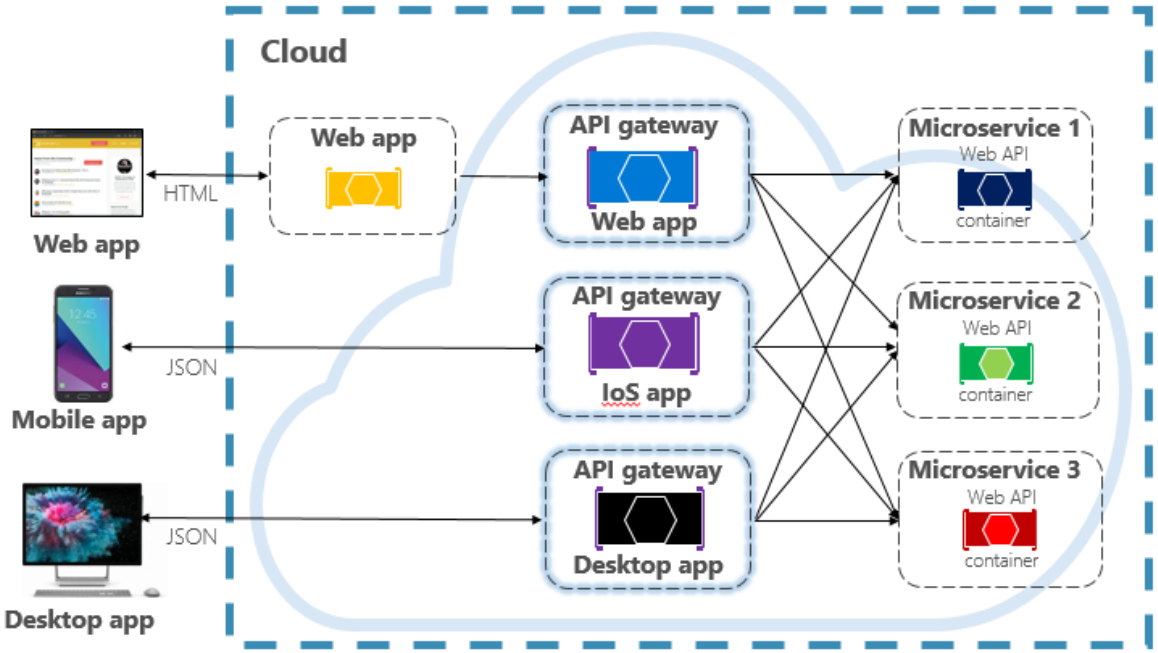


Figure 4-4. Backend for frontend pattern

Note in the previous figure how incoming traffic is sent to a specific API gateway - based upon client type: web, mobile, or desktop app. This approach makes sense as the capabilities of each device differ significantly across form factor, performance, and display limitations. Typically mobile applications expose less functionality than a browser or desktop applications. Each gateway can be optimized to match the capabilities and functionality of the corresponding device.

To start, you could build your own API Gateway service. A quick search of GitHub will provide many examples. However, there are several frameworks and commercial gateway products available.

Ocelot Gateway

For simple .NET cloud-native applications, you might consider the Ocelot Gateway. Ocelot is an Open Source API Gateway created for .NET microservices that require a unified point of entry into their system. It's lightweight, fast, scalable.

Like any API Gateway, its primary functionality is to forward incoming HTTP requests to downstream services. Additionally, it supports a wide variety of capabilities that are configurable in a .NET Core middleware pipeline. Its feature set is presented in following table.

Ocelot Features	
Routing	Authentication
Request Aggregation	Authorization
Service Discovery (with Consul and Eureka)	Throttling
Load Balancing	Logging, Tracing

Ocelot Features	
Caching	Headers/Query String Transformation
Correlation Pass-Through	Custom Middleware
Quality of Service	Retry Policies

Each Ocelot gateway specifies the upstream and downstream addresses and configurable features in a JSON configuration file. The client sends an HTTP request to the Ocelot gateway. Once received, Ocelot passes the HttpRequest object through its pipeline manipulating it into the state specified by its configuration. At the end of pipeline, Ocelot creates a new HTTPResponseObject and passes it to the downstream service. For the response, Ocelot reverses the pipeline, sending the response back to client.

Ocelot is available as a NuGet package. It targets the NET Standard 2.0, making it compatible with both .NET Core 2.0+ and .NET Framework 4.6.1+ runtimes. Ocelot integrates with anything that speaks HTTP and runs on the platforms which .NET Core supports: Linux, macOS, and Windows. Ocelot is extensible and supports many modern platforms, including Docker containers, Azure Kubernetes Services, or other public clouds. Ocelot integrates with open-source packages like Consul, GraphQL, and Netflix's Eureka.

Consider Ocelot for simple cloud-native applications that don't require the rich feature-set of a commercial API gateway.

Azure Application Gateway

For simple gateway requirements, you may consider Azure Application Gateway. Available as an Azure PaaS service, it includes basic gateway features such as URL routing, SSL termination, and a Web Application Firewall. The service supports Layer-7 load balancing capabilities. With Layer 7, you can route requests based on the actual content of an HTTP message, not just low-level TCP network packets.

Throughout this book, we evangelize hosting cloud-native systems in Kubernetes. A container orchestrator, Kubernetes automates the deployment, scaling, and operational concerns of containerized workloads. Azure Application Gateway can be configured as an API gateway for Azure Kubernetes Service cluster.

The Application Gateway Ingress Controller enables Azure Application Gateway to work directly with Azure Kubernetes Service. Figure 4.5 shows the architecture.

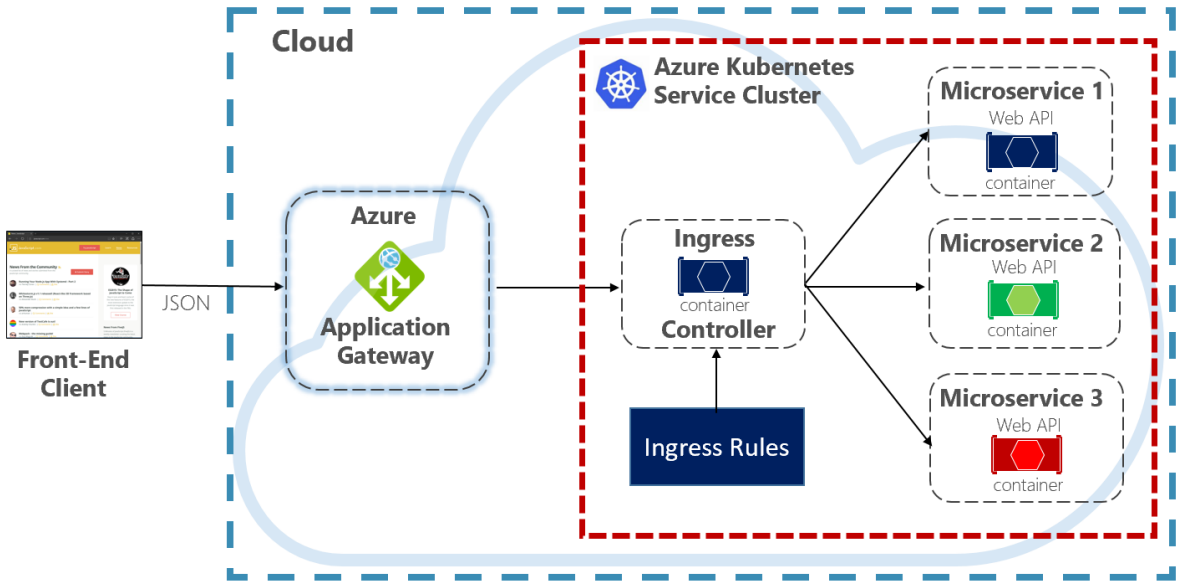


Figure 4-5. Application Gateway Ingress Controller

Kubernetes includes a built-in feature that supports HTTP (Level 7) load balancing, called Ingress. Ingress defines a set of rules for how microservice instances inside AKS can be exposed to the outside world. In the previous image, the ingress controller interprets the ingress rules configured for the cluster and automatically configures the Azure Application Gateway. Based on those rules, the Application Gateway routes traffic to microservices running inside AKS. The ingress controller listens for changes to ingress rules and makes the appropriate changes to the Azure Application Gateway.

Azure API Management

For moderate to large-scale cloud-native systems, you may consider Azure API Management. It's a cloud-based service that not only solves your API Gateway needs, but provides a full-featured developer and administrative experience. API Management is shown in Figure 4-6.

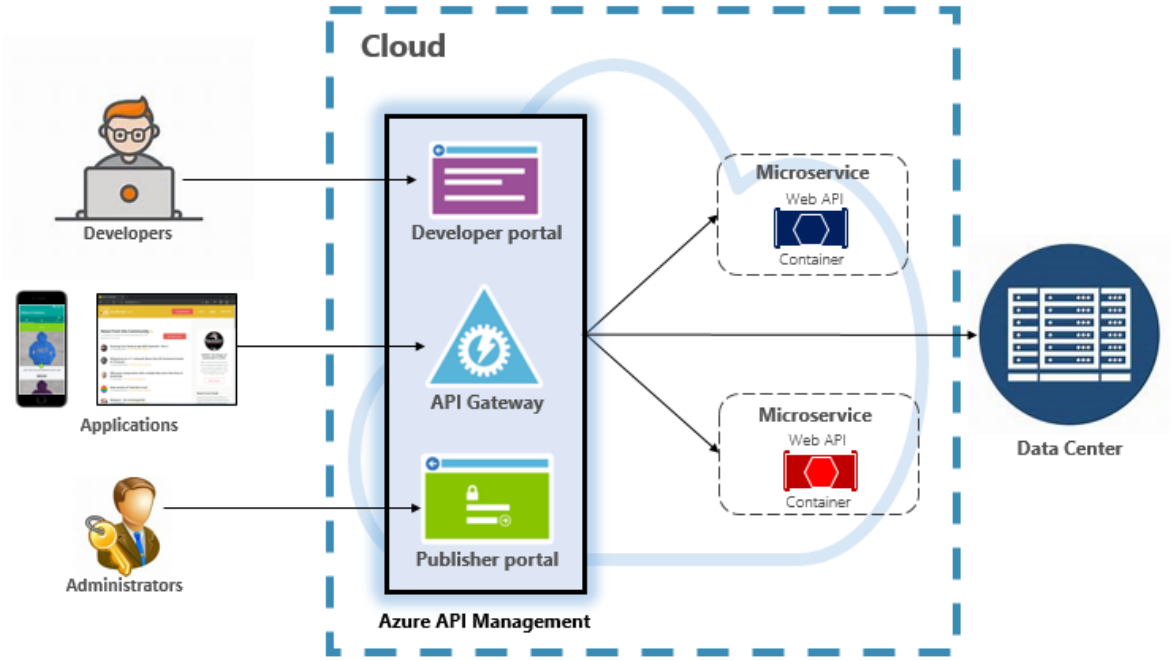


Figure 4-6. Azure API Management

To start, API Management exposes a gateway server that allows controlled access to back-end services based upon configurable rules and policies. These services can be in the Azure cloud, your on-prem data center, or other public clouds. API keys and JWT tokens determine who can do what. All traffic is logged for analytical purposes.

For developers, API Management offers a developer portal that provides access to services, documentation, and sample code for invoking them. Developers can use Swagger/Open API to inspect service endpoints and analyze their usage. The service works across the major development platforms: .NET, Java, Golang, and more.

The publisher portal exposes a management dashboard where administrators expose APIs and manage their behavior. Service access can be granted, service health monitored, and service telemetry gathered. Administrators apply *policies* to each endpoint to affect behavior. Policies are pre-built statements that execute sequentially for each service call. Policies are configured for an inbound call, outbound call, or invoked upon an error. Policies can be applied at different service scopes as to enable deterministic ordering when combining policies. The product ships with a large number of prebuilt policies.

Here are examples of how policies can affect the behavior of your cloud-native services:

- Restrict service access.
- Enforce authentication.
- Throttle calls from a single source, if necessary.
- Enable caching.
- Block calls from specific IP addresses.
- Control the flow of the service.
- Convert requests from SOAP to REST or between different data formats, such as from XML to JSON.

Azure API Management can expose back-end services that are hosted anywhere – in the cloud or your data center. For legacy services that you may expose in your cloud-native systems, it supports both REST and SOAP APIs. Even other Azure services can be exposed through API Management. You could place a managed API on top of an Azure backing service like Azure Service Bus or Azure Logic Apps. Azure API Management doesn't include built-in load-balancing support and should be used in conjunction with a load-balancing service.

Azure API Management is available across four different tiers:

- Developer
- Basic
- Standard
- Premium

The Developer tier is meant for non-production workloads and evaluation. The other tiers offer progressively more power, features, and higher service level agreements (SLAs). The Premium tier provides Azure Virtual Network and multi-region support. All tiers have a fixed price per hour.

Recently, Microsoft announced a API Management serverless tier for Azure API Management. Referred to as the *consumption pricing tier*, the service is a variant of API Management designed around the serverless computing model. Unlike the "pre-allocated" pricing tiers previously shown, the consumption tier provides instant provisioning and pay-per-action pricing.

It enables API Gateway features for the following use cases:

- Microservices implemented using serverless technologies such as Azure Functions and Azure Logic Apps.
- Azure backing service resources such as Service Bus queues and topics, Azure storage, and others.
- Microservices where traffic has occasional large spikes but remains low the majority of the time.

The consumption tier uses the same underlying service API Management components, but employs an entirely different architecture based on dynamically allocated resources. It aligns perfectly with the serverless computing model:

- No infrastructure to manage.
- No idle capacity.
- High-availability.
- Automatic scaling.
- Cost is based on actual usage.

The new consumption tier is a great choice for cloud-native systems that expose serverless resources as APIs.

At the time of writing, the consumption tier is in preview in the Azure cloud.

Real-time communication

Real-time, or push, communication is another option for front-end applications that communicate with back-end cloud-native systems over HTTP. Applications, such as financial-tickers, online education, gaming, and job-progress updates, require instantaneous, real-time responses from the back-end. With normal HTTP communication, there's no way for the client to know when new data is available. The client must continually *poll* or send requests to the server. With *real-time* communication, the server can push new data to the client at any time.

Real-time systems are often characterized by high-frequency data flows and large numbers of concurrent client connections. Manually implementing real-time connectivity can quickly become complex, requiring non-trivial infrastructure to ensure scalability and reliable messaging to connected clients. You could find yourself managing an instance of Azure Redis Cache and a set of load balancers configured with sticky sessions for client affinity.

Azure SignalR Service is a fully managed Azure service that simplifies real-time communication for your cloud-native applications. Technical implementation details like capacity provisioning, scaling, and persistent connections are abstracted away. They're handled for you with a 99.9% service-level agreement. You focus on application features, not infrastructure plumbing.

Once enabled, a cloud-based HTTP service can push content updates directly to connected clients, including browser, mobile and desktop applications. Clients are updated without the need to poll the server. Azure SignalR abstracts the transport technologies that create real-time connectivity, including WebSockets, Server-Side Events, and Long Polling. Developers focus on sending messages to all or specific subsets of connected clients.

Figure 4-7 shows a set of HTTP Clients connecting to a Cloud-native application with Azure SignalR enabled.

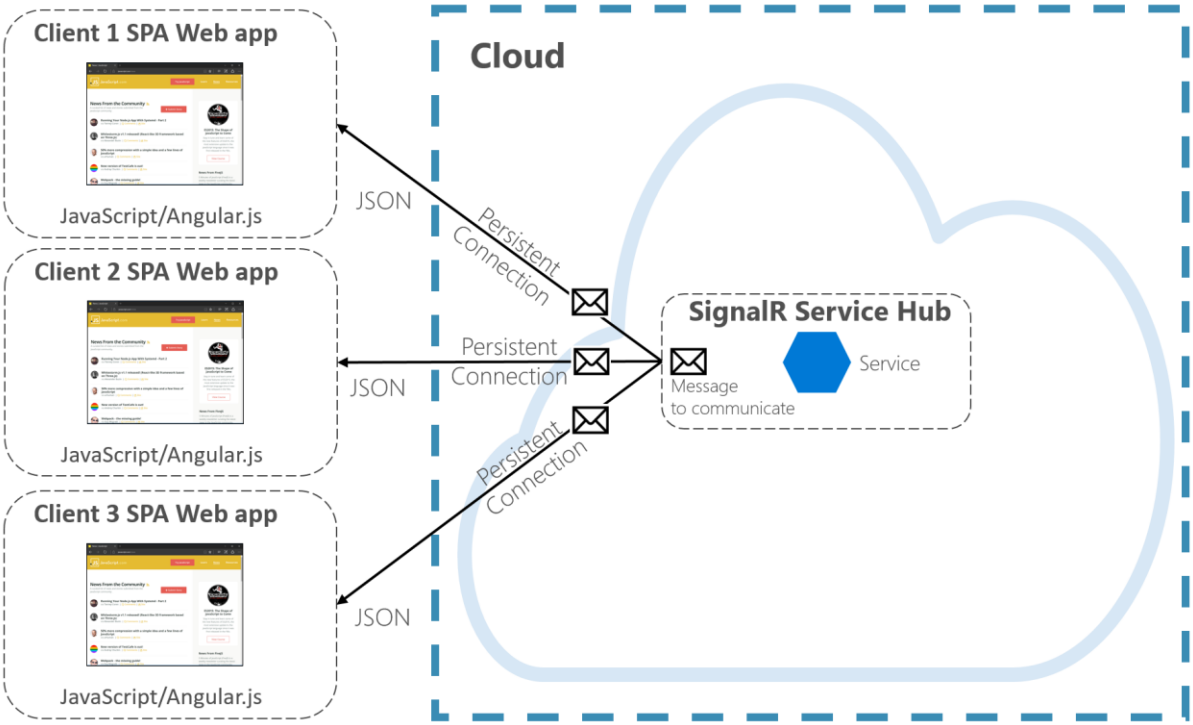


Figure 4-7. Azure SignalR

Another advantage of Azure SignalR Service comes with implementing Serverless cloud-native services. Perhaps your code is executed on demand with Azure Functions triggers. This scenario can be tricky because your code doesn't maintain long connections with clients. Azure SignalR Service can handle this situation since the service already manages connections for you.

Azure SignalR Service closely integrates with other Azure services, such as Azure SQL Database, Service Bus, or Redis Cache, opening up many possibilities for your cloud-native applications.

Service-to-service communication

Moving from the front-end client, we now address back-end microservices communicate with each other.

When constructing a cloud-native application, you'll want to be sensitive to how back-end services communicate with each other. Ideally, the less inter-service communication, the better. However, avoidance isn't always possible as back-end services often rely on one another to complete an operation.

There are several widely accepted approaches to implementing cross-service communication. The *type of communication interaction* will often determine the best approach.

Consider the following interaction types:

- *Query* – when a calling microservice requires a response from a called microservice, such as, "Hey, give me the buyer information for a given customer Id."
- *Command* – when the calling microservice needs another microservice to execute an action but doesn't require a response, such as, "Hey, just ship this order."
- *Event* – when a microservice, called the publisher, raises an event that state has changed or an action has occurred. Other microservices, called subscribers, who are interested, can react to the event appropriately. The publisher and the subscribers aren't aware of each other.

Microservice systems typically use a combination of these interaction types when executing operations that require cross-service interaction. Let's take a close look at each and how you might implement them.

Queries

Many times, one microservice might need to *query* another, requiring an immediate response to complete an operation. A shopping basket microservice may need product information and a price to add an item to its basket. There are a number of approaches for implementing query operations.

Request/Response Messaging

One option for implementing this scenario is for the calling back-end microservice to make direct HTTP requests to the microservices it needs to query, shown in Figure 4-8.

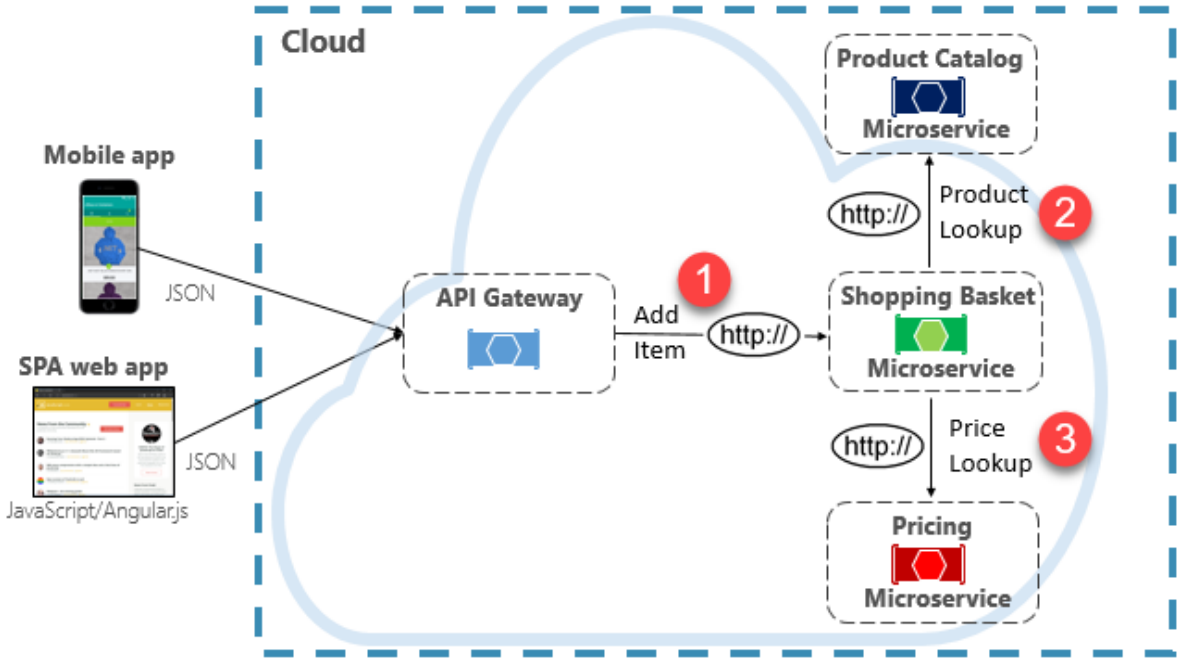


Figure 4-8. Direct HTTP communication

While direct HTTP calls between microservices are relatively simple to implement, care should be taken to minimize this practice. To start, these calls are always *synchronous* and will block the operation until a result is returned or the request times outs. What were once self-contained, independent services, able to evolve independently and deploy frequently, now become coupled to each other. As coupling among microservices increase, their architectural benefits diminish.

Executing an infrequent request that makes a single direct HTTP call to another microservice might be acceptable for some systems. However, high-volume calls that invoke direct HTTP calls to multiple microservices aren't advisable. They can increase latency and negatively impact the performance, scalability, and availability of your system. Even worse, a long series of direct HTTP communication can lead to deep and complex chains of synchronous microservices calls, shown in Figure 4-9:

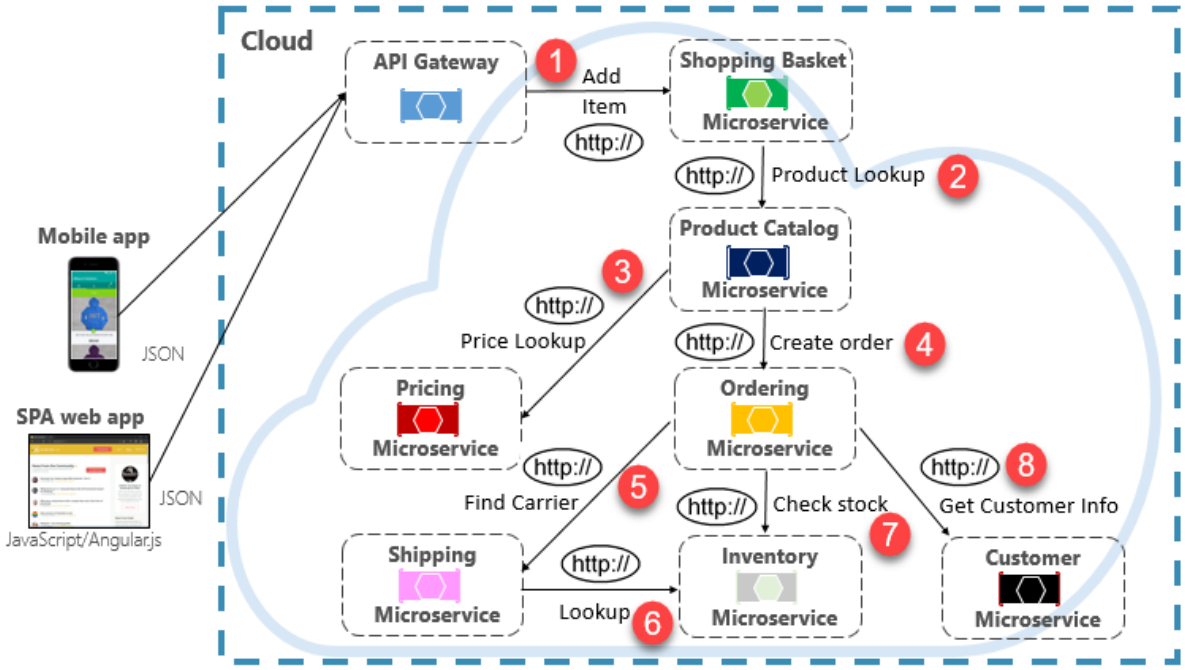


Figure 4-9. Chaining HTTP queries

You can certainly imagine the risk in the design shown in the previous image. What happens if Step #3 fails? Or Step #8 fails? How do you recover? What if Step #6 is slow because the underlying service is busy? How do you continue? Even if all works correctly, think of the latency this call would incur, which is the sum of the latency of each step.

The large degree of coupling in the previous image suggests the services weren't optimally modeled. It would behoove the team to revisit their design.

Materialized View pattern

A popular option for removing microservice coupling is the Materialized View pattern. With this pattern, a microservice stores its own local, denormalized copy of data that's owned by other services. Instead of the Shopping Basket microservice querying the Product Catalog and Pricing microservices, it maintains its own local copy of that data. This pattern eliminates unnecessary coupling and

improves reliability and response time. The entire operation executes inside a single process. We explore this pattern and other data concerns in Chapter 5.

Service Aggregator Pattern

Another option for eliminating microservice-to-microservice coupling is an Aggregator microservice, shown in purple in Figure 4-10.

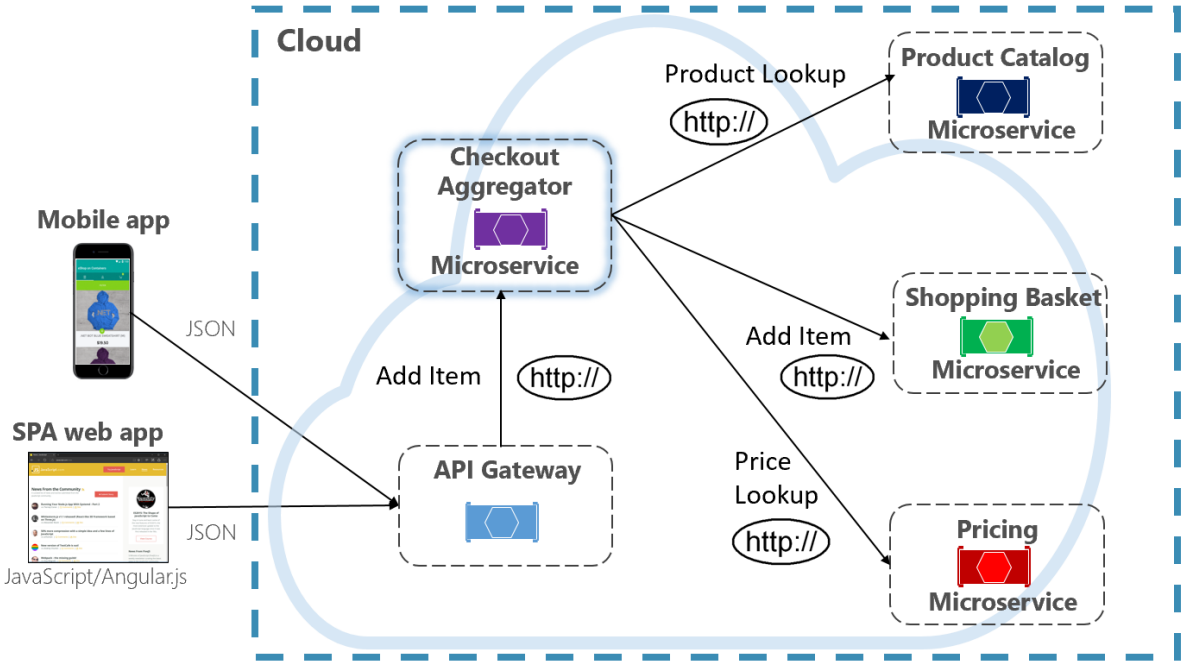


Figure 4-10. Aggregator microservice

The pattern isolates an operation that makes calls to multiple back-end microservices, centralizing its logic into a specialized microservice. The purple checkout aggregator microservice in the previous figure orchestrates the workflow for the Checkout operation. It includes calls to several back-end microservices in a sequenced order. Data from the workflow is aggregated and returned to the caller. While it still implements direct HTTP calls, the aggregator microservice reduces direct dependencies among back-end microservices.

Request/Reply Pattern

Another approach for decoupling synchronous HTTP messages is a Request-Reply Pattern, which uses queuing communication. Communication using a queue is always a one-way channel, with a producer sending the message and consumer receiving it. With this pattern, both a request queue and response queue are implemented, shown in Figure 4-11.

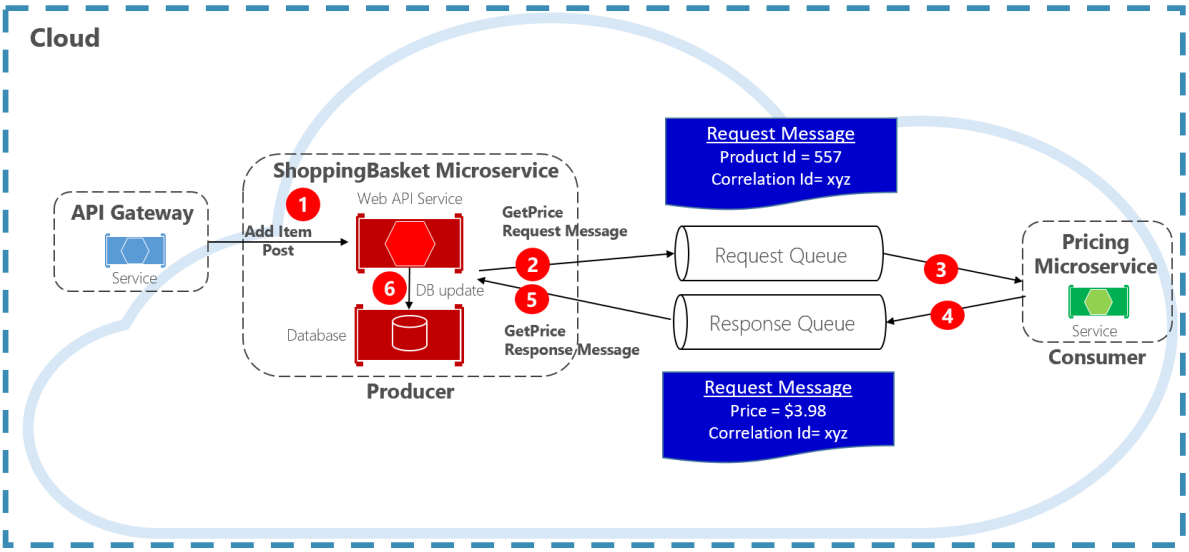


Figure 4-11. Request-reply pattern

Here, the message producer creates a query-based message that contains a unique correlation ID and places it into a request queue. The consuming service dequeues the messages, processes it and places the response into the response queue with the same correlation ID. The producer service dequeues the message, matches it with the correlation ID and continues processing. We cover queues in detail in the next section.

Commands

Another type of communication interaction is a *command*. A microservice may need another microservice to perform an action. The Ordering microservice may need the Shipping microservice to create a shipment for an approved order. In Figure 4-12, one microservice, called a Producer, sends a message to another microservice, the Consumer, commanding it to do something.

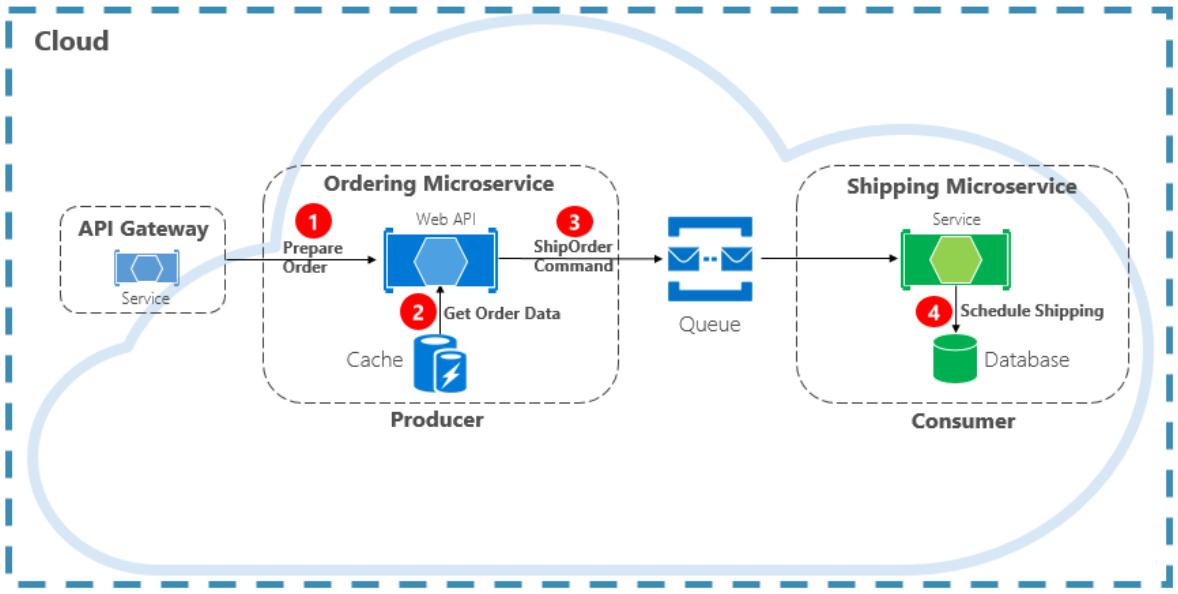


Figure 4-12. Command interaction with a queue

Most often, the Producer doesn't require a response and can *fire-and-forget* the message. If a reply is needed, the Consumer sends a separate message back to Producer on another channel. A command message is best sent asynchronously with a message queue. supported by a lightweight message broker. In the previous diagram, note how a queue separates and decouples both services.

A message queue is an intermediary construct through which a producer and consumer pass a message. Queues implement an asynchronous, point-to-point messaging pattern. The Producer knows where a command needs to be sent and routes appropriately. The queue guarantees that a message is processed by exactly one of the consumer instances that are reading from the channel. In this scenario, either the producer or consumer service can scale out without affecting the other. As well, technologies can be disparate on each side, meaning that we might have a Java microservice calling a Golang microservice.

In chapter 1, we talked about *backing services*. Backing services are ancillary resources upon which cloud-native systems depend. Message queues are backing services. The Azure cloud supports two types of message queues that your cloud-native systems can consume to implement command messaging: Azure Storage Queues and Azure Service Bus Queues.

Azure Storage Queues

Azure storage queues offer a simple queueing infrastructure that is fast, affordable, and backed by Azure storage accounts.

Azure Storage Queues feature a REST-based queuing mechanism with reliable and persistent messaging. They provide a minimal feature set, but are inexpensive and store millions of messages. Their capacity ranges up to 500 TB. A single message can be up to 64 KB in size.

You can access messages from anywhere in the world via authenticated calls using HTTP or HTTPS. Storage queues can scale out to large numbers of concurrent clients to handle traffic spikes.

That said, there are limitations with the service:

- Message order isn't guaranteed.
- A message can only persist for seven days before it's automatically removed.
- Support for state management, duplicate detection, or transactions isn't available.

Figure 4-13 shows the hierarchy of an Azure Storage Queue.

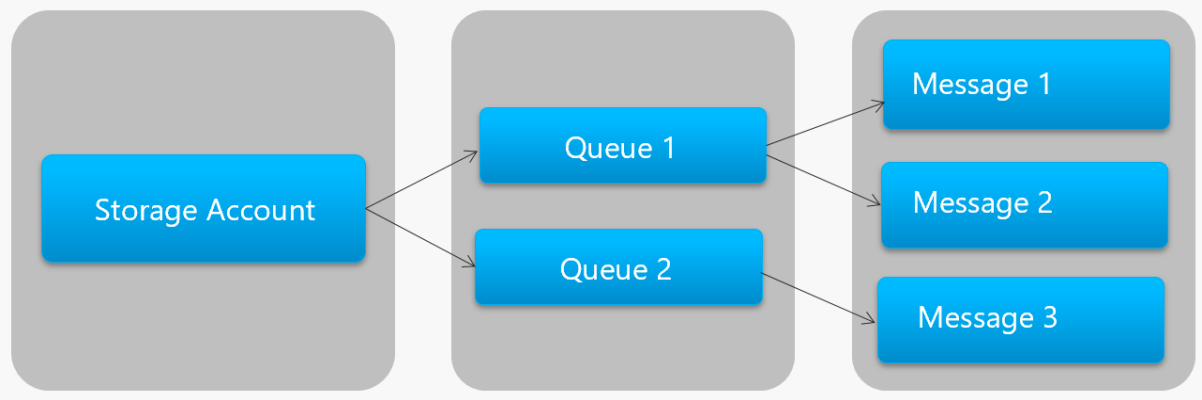


Figure 4-13. Storage queue hierarchy

In the previous figure, note how storage queues store their messages in the underlying Azure Storage account.

For developers, Microsoft provides several client and server-side libraries for Storage queue processing. Most major platforms are supported including .NET, Java, JavaScript, Ruby, Python, and Go. Developers should never communicate directly with these libraries. Doing so will tightly couple your microservice code to the Azure Storage Queue service. It's a better practice to insulate the implementation details of the API. Introduce an intermediation layer, or intermediate API, that exposes generic operations and encapsulates the concrete library. This loose coupling enables you to swap out one queuing service for another without having to make changes to the mainline service code.

Azure Storage queues are an economical option to implement command messaging in your cloud-native applications. Especially when a queue size will exceed 80 GB, or a simple feature set is acceptable. You only pay for the storage of the messages; there are no fixed hourly charges.

Azure Service Bus Queues

For more complex messaging requirements, consider Azure Service Bus queues.

Sitting atop a robust message infrastructure, Azure Service Bus supports a *brokered messaging model*. Messages are reliably stored in a broker (the queue) until received by the consumer. The queue guarantees First-In/First-Out (FIFO) message delivery, respecting the order in which messages were added to the queue.

The size of a message can be much larger, up to 256 KB. Messages are persisted in the queue for an unlimited period of time. Service Bus supports not only HTTP-based calls, but also provides full

support for the AMPQ protocol. AMPQ is an open-standard across vendors that supports a binary protocol and higher degrees of reliability.

Service Bus provides a rich set of features, including transaction support and a duplicate detection feature. The queue guarantees "at most once delivery" per message. It automatically discards a message that has already been sent. If a producer is in doubt, it can resend the same message, and Service Bus guarantees that only one copy will be processed. Duplicate detection frees you from having to build additional infrastructure plumbing.

Two more enterprise features are partitioning and sessions. A conventional Service Bus queue is handled by a single message broker and stored in a single message store. But, Service Bus Partitioning spreads the queue across multiple message brokers and message stores. The overall throughput is no longer limited by the performance of a single message broker or messaging store. A temporary outage of a messaging store doesn't render a partitioned queue unavailable.

Service Bus Sessions provide a way to group-related messages. Imagine a workflow scenario where messages must be processed together and the operation completed at the end. To take advantage, sessions must be explicitly enabled for the queue and each related messaged must contain the same session ID.

However, there are some important caveats: Service Bus queues size is limited to 80 GB, which is much smaller than what's available from store queues. Additionally, Service Bus queues incur a base cost and charge per operation.

Figure 4-14 outlines the high-level architecture of a Service Bus queue.

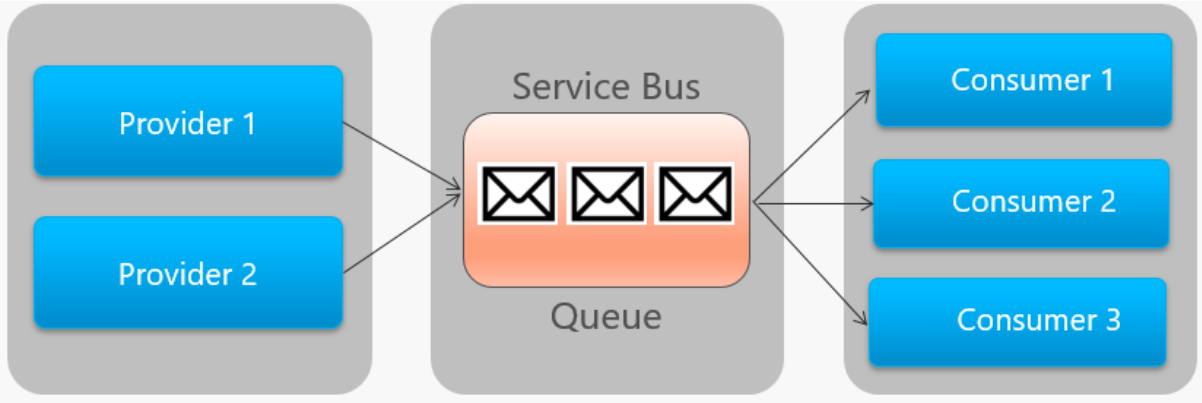


Figure 4-14. Service Bus queue

In the previous figure, note the point-to-point relationship. Two instances of the same provider are enqueuing messages into a single Service Bus queue. Each message is consumed by only one of three consumer instances on the right. Next, we discuss how to implement messaging where different consumers may all be interested the same message.

Events

Message queuing is an effective way to implement communication where a producer can asynchronously send a consumer a message. However, what happens when *many different consumers*

are interested in the same message? A dedicated message queue for each consumer wouldn't scale well and would become difficult to manage.

To address this scenario, we move to the third type of message interaction, the *event*. One microservice announces that an action had occurred. Other microservices, if interested, react to the action, or event.

Eventing is a two-step process. For a given state change, a microservice publishes an event to a message broker, making it available to any other interested microservice. The interested microservice is notified by subscribing to the event in the message broker. You use the Publish/Subscribe pattern to implement event-based communication.

Figure 4-15 shows a shopping basket microservice publishing an event with two other microservices subscribing to it.

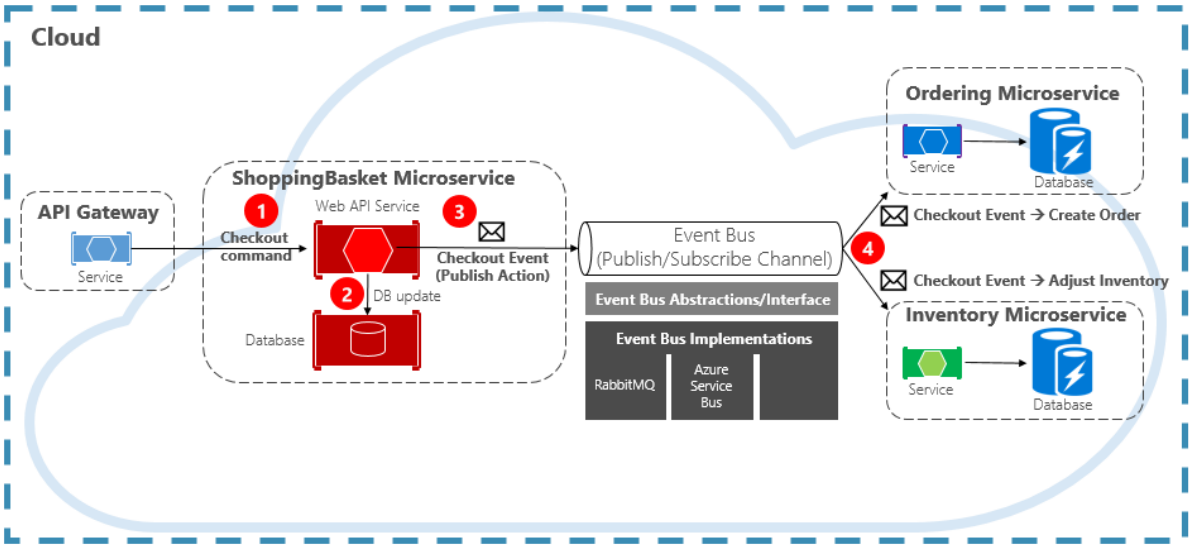


Figure 4-15. Event-Driven messaging

Note the *event bus* component that sits in the middle of the communication channel. It's a custom class that encapsulates the message broker and decouples it from the underlying application. The ordering and inventory microservices independently operate the event with no knowledge of each other, nor the shopping basket microservice. When the registered event is published to the event bus, they act upon it.

With eventing, we move from queuing technology to *topics*. A topic is similar to a queue, but supports a one-to-many messaging pattern. One microservice publishes a message. Multiple subscribing microservices can choose to receive and act upon that message. Figure 4-16 shows a topic architecture.

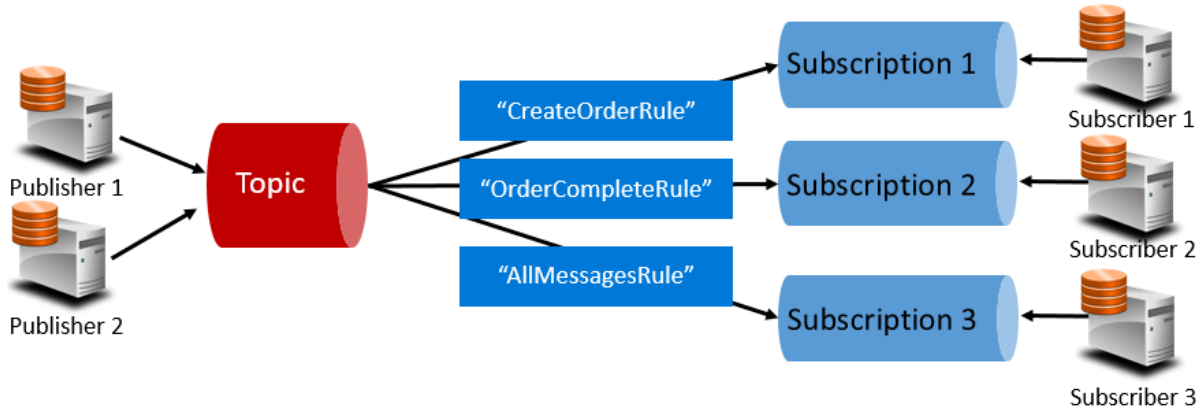


Figure 4-16. Topic architecture

In the previous figure, publishers send messages to the topic. At the end, subscribers receive messages from subscriptions. In the middle, the topic forwards messages to subscriptions based on a set of *rules*, shown in dark blue boxes. Rules act as a filter that forward specific messages to a subscription. Here, a "CreateOrder" event would be sent to Subscription #1 and Subscription #3, but not to Subscription #2. An "OrderCompleted" event would be sent to Subscription #2 and Subscription #3.

The Azure cloud supports two different topic services: Azure Service Bus Topics and Azure EventGrid.

Azure Service Bus Topics

Sitting on top of the same robust brokered message model of Azure Service Bus queues are Azure Service Bus Topics. A topic can receive messages from multiple independent publishers and send messages to up to 2,000 subscribers. Subscriptions can be dynamically added or removed at runtime without stopping the system or recreating the topic.

Many advanced features from Azure Service Bus queues are also available for topics, including Duplicate Detection and Transaction support. By default, Service Bus topics are handled by a single message broker and stored in a single message store. But, Service Bus Partitioning scales a topic by spreading it across many message brokers and message stores.

Scheduled Message Delivery tags a message with a specific time for processing. The message won't appear in the topic before that time. Message Deferral enables you to defer a retrieval of a message to a later time. Both are commonly used in workflow processing scenarios where operations are processed in a particular order. You can postpone processing of received messages until prior work has been completed.

Service Bus topics are a robust and proven technology for enabling publish/subscribe communication in your cloud-native systems.

Azure Event Grid

While Azure Service Bus is a battle-tested messaging broker with a full set of enterprise features, Azure Event Grid is the new kid on the block.

At first glance, Event Grid may look like just another topic-based messaging system. However, it's different in many ways. Focused on event-driven workloads, it enables real-time event processing, deep Azure integration, and an open-platform - all on serverless infrastructure. It's designed for contemporary cloud-native and serverless applications

As a centralized *eventing backplane*, or pipe, Event Grid reacts to events inside Azure resources and from your own services.

Event notifications are published to an Event Grid Topic, which, in turn, routes each event to a subscription. Subscribers map to subscriptions and consume the events. Like Service Bus, Event Grid supports a *filtered subscriber model* where a subscription sets rule for the events it wishes to receive. Event Grid provides fast throughput with a guarantee of 10 million events per second enabling near real-time delivery - far more than what Azure Service Bus can generate.

A sweet spot for Event Grid is its deep integration into the fabric of Azure infrastructure. An Azure resource, such as Cosmos DB, can publish built-in events directly to other interested Azure resources - without the need for custom code. Event Grid can publish events from an Azure Subscription, Resource Group, or Service, giving developers fine-grained control over the lifecycle of cloud resources. However, Event Grid isn't limited to Azure. It's an open platform that can consume custom HTTP events published from applications or third-party services and route events to external subscribers.

When publishing and subscribing to native events from Azure resources, no coding is required. With simple configuration, you can integrate events from one Azure resource to another leveraging built-in plumbing for Topics and Subscriptions. Figure 4-17 shows the anatomy of Event Grid.

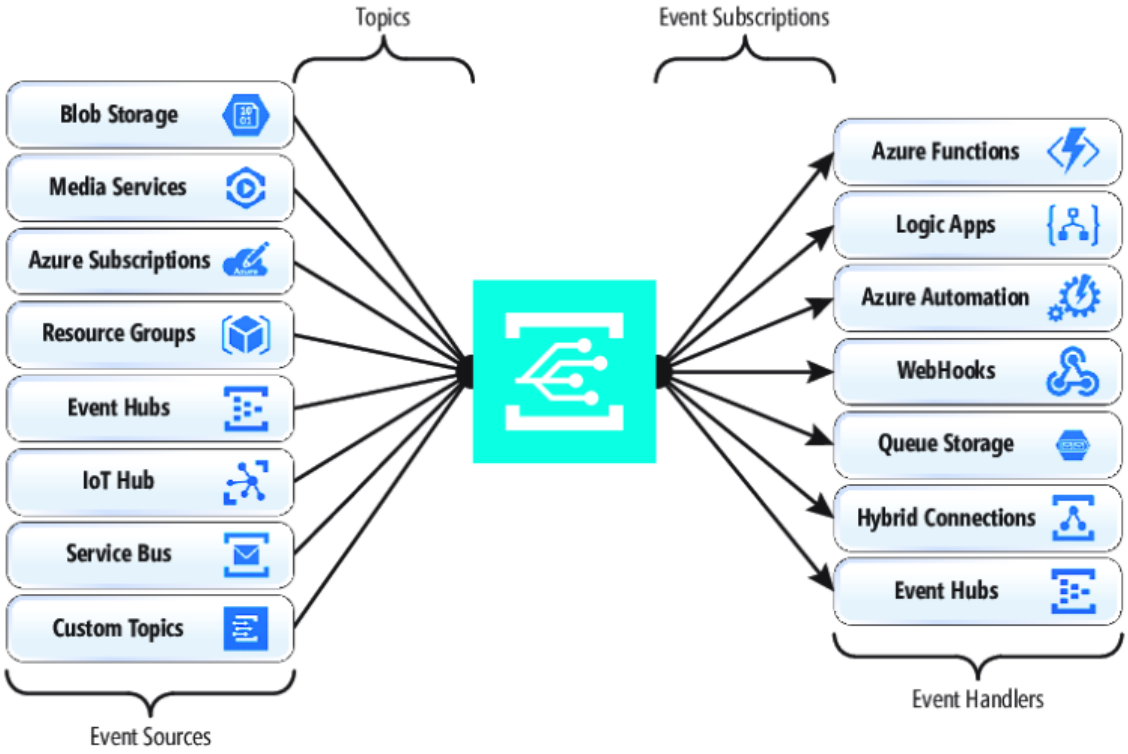


Figure 4-17. Event Grid anatomy

A major difference between EventGrid and Service Bus is the underlying *message exchange pattern*.

Service Bus implements an older style *pull model* in which the downstream subscriber actively polls the topic subscription for new messages. On the upside, this approach gives the subscriber full control of the pace at which it processes messages. It controls when and how many messages to process at any given time. Unread messages remain in the subscription until processed. A significant shortcoming is the latency between the time the event is generated and the polling operation that pulls that message to the subscriber for processing. Also, the overhead of constant polling for the next event consumes resources and money.

EventGrid, however, is different. It implements a *push model* in which events are sent to the EventHandlers as received, giving near real-time event delivery. It also reduces cost as the service is triggered only when it's needed to consume an event – not continually as with polling. That said, an event handler must handle the incoming load and provide throttling mechanisms to protect itself from becoming overwhelmed. Many Azure services that consume these events, such as Azure Functions and Logic Apps provide automatic autoscaling capabilities to handle increased loads.

Event Grid is a fully managed serverless cloud service. It dynamically scales based on your traffic and charges you only for your actual usage, not pre-purchased capacity. The first 100,000 operations per month are free – operations being defined as event ingress (incoming event notifications), subscription delivery attempts, management calls, and filtering by subject. With 99.99% availability, EventGrid guarantees the delivery of an event within a 24-hour period, with built-in retry functionality for unsuccessful delivery. Undelivered messages can be moved to a "dead-letter" queue for resolution. Unlike Azure Service Bus, Event Grid is tuned for fast performance and doesn't support features like ordered messaging, transactions, and sessions.

Streaming messages in the Azure cloud

Azure Service Bus and Event Grid provide great support for applications that expose single, discrete events like a new document been inserted into a Cosmos DB). But, what if your cloud-native system needs to process a *stream of related events*? Event streams are more complex. They're typically time-ordered, interrelated and must be processed as a group.

Azure Event Hub is a data streaming platform and event ingestion service that collects, transforms, and stores events. It's fine-tuned to capture streaming data, such as continuous event notifications emitted from a telemetry context. The service is highly scalable and can store and process millions of events per second. Shown in Figure 4-18, it's often a front door for an event pipeline, decoupling ingest stream from event consumption.

Figure 4-18. Azure Event Hub

Event Hub supports low latency and configurable time retention. Unlike queues and topics, Event Hubs keep event data after it's been read by a consumer. This feature enables other data analytic services, both internal and external, to replay the data for further analysis. Events stored in event hub are only deleted upon expiration of the retention period, which is one day by default, but configurable.

Event Hub supports common event publishing protocols including HTTPS and AMQP. It also supports Kafka 1.0. Existing Kafka applications can communicate with Event Hub using the Kafka protocol providing an alternative to managing large Kafka clusters. Many open-source cloud-native systems embrace Kafka.

Event Hubs implements message streaming through a partitioned consumer model in which each consumer only reads a specific subset, or partition, of the message stream. This pattern enables tremendous horizontal scale for event processing and provides other stream-focused features that are unavailable in queues and topics. A partition is an ordered sequence of events that is held in an event hub. As newer events arrive, they're added to the end of this sequence. Figure 4-19 shows partitioning in an Event Hub.

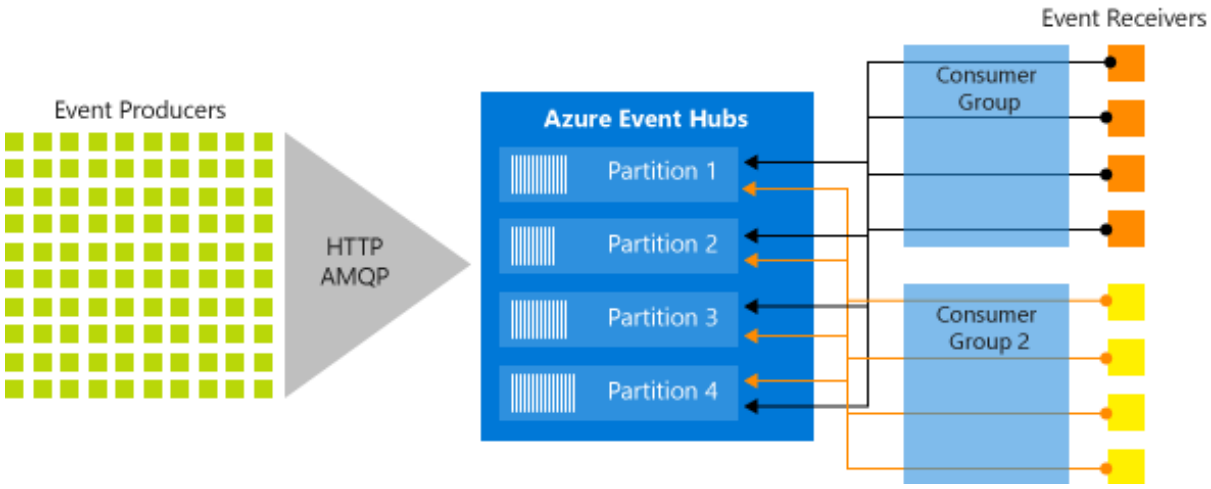


Figure 4-19. Event Hub partitioning

Instead of reading from the same resource, each consumer group reads across a subset, or partition, of the message stream.

For cloud-native applications that must stream large numbers of events, Azure Event Hub can be a robust and affordable solution.

REST and gRPC

So far in this book, we've focused on REST-based communication. REST is an architectural style that promotes interoperability between distributed computer systems. It uses a request/response model where every response from the server is to a request from the client. While widely popular, REST isn't a perfect fit for every problem. A newer communication technology, entitled gRPC, is rapidly gaining popularity and making its way into the cloud-native world.

gRPC

gRPC is an open-source communication that originates from Google. It's built upon the remote procedure call (RPC) model, popular in distributed computing. Following this model, a local client program exposes an in-process method to execute an operation. Behind the scenes, that call is invoked on an out-of-process microservice across a distributed network. The developer codes the operation as a local procedure call. The underlying platform abstracts the point-to-point networking communication, serialization, and execution.

gRPC is a modern RPC framework that is lightweight and highly performant. It uses HTTP/2 for its transport protocol. While compatible with HTTP 1.1, HTTP/2 features many advanced capabilities:

- While HTTP 1.1 sends data as clear text, HTTP/2 is a binary protocol. It parses data faster using less memory, reduces network latency with the related improvements to speed, and manages network resources more efficiently.
- While HTTP 1.1 is limited to processing one round-trip request/response at a time, HTTP /2 supports multiplexing, or multiple parallel requests over the same connection.
- HTTP/2 supports full-duplex, or bidirectional communication, where both client and server and can communicate at the same time. The client can be uploading request data at the same time the server is sending back response data.
- Streaming is built into HTTP/2 meaning that both requests and responses can asynchronously stream large data sets.
- Combining gRPC and HTTP/2, performance dramatically increases. In Windows Communication Foundation (WCF) parlance, gPRC performance meets and exceeds the speed and efficiency of NetTCP bindings. However, unlike NetTCP, gRPC isn't constrained to Microsoft languages such as C# or VB.NET.

gRPC is supported across most popular platforms, including Java, C#, Golang and NodeJS.

Protocol Buffers

gRPC embraces another open-source technology called Protocol Buffers or Protobuf messages to send and receive data. Similar to a WCF Data Contract, Protobuf serializes structured data for systems to read and write. It reduces the overhead that human-readable formats like XML or JSON incur.

Many object serialization techniques reflect across the structure of data objects at run-time. Protobuf requires you to define the structure up front with a platform-agnostic language (Protocol Buffer Language). Each definition is stored in a ".proto" file. Then using Protobuf compiler, "Proton," you generate client and server code for any of the supported platforms. The generated code is optimized for fast serialization/deserialization of data. At runtime, each message is wrapped in the strongly-typed service contract and serialized in a standard Protobuf representation.

gRPC support in .NET

The Microsoft .NET Core framework 3.0 includes tooling and native support for gRPC. Figure 4-20 shows the Visual Studio 2019 template that scaffolds a gRPC skeleton project for a gRPC service. Note how .NET Core supports the Windows, Linux, macOS platforms.

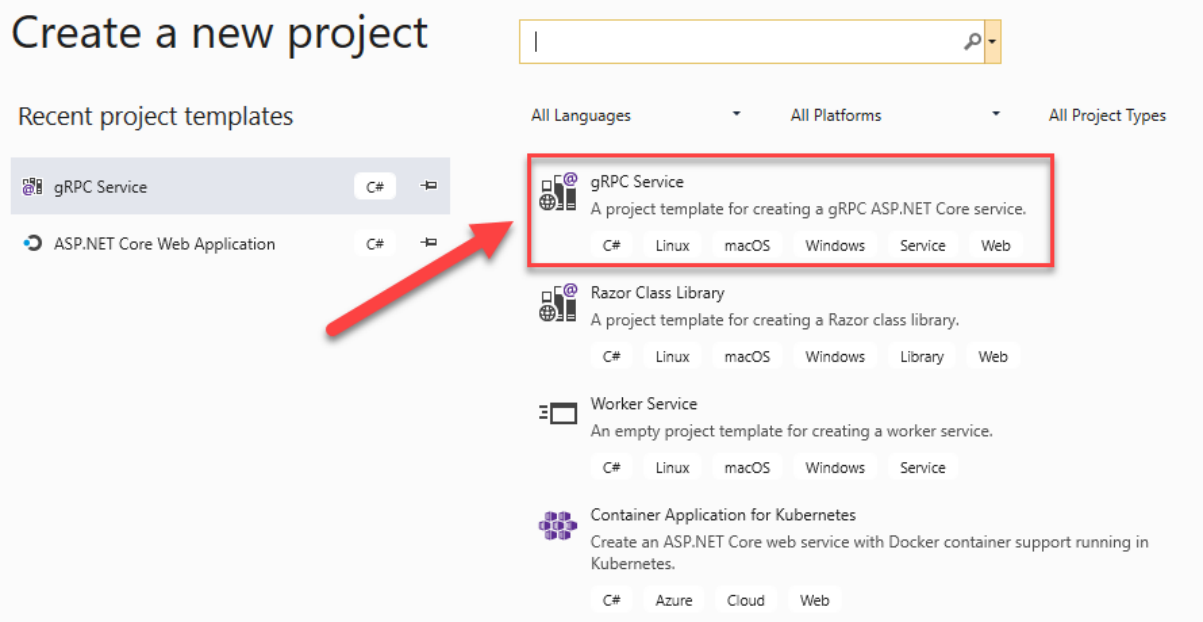


Figure 4-20. gRPC support in Visual Studio 2019

.NET Core 3.0 seamlessly integrates gRPC into its framework, including endpoint routing, built-in IoC support, and logging. The open-source Kestrel web server fully supports HTTP/2 connections.

Figure 4-21 shows structure of a gRPC service in Visual Studio 2019. Note how the folder structure includes folders for the proto files and service code.

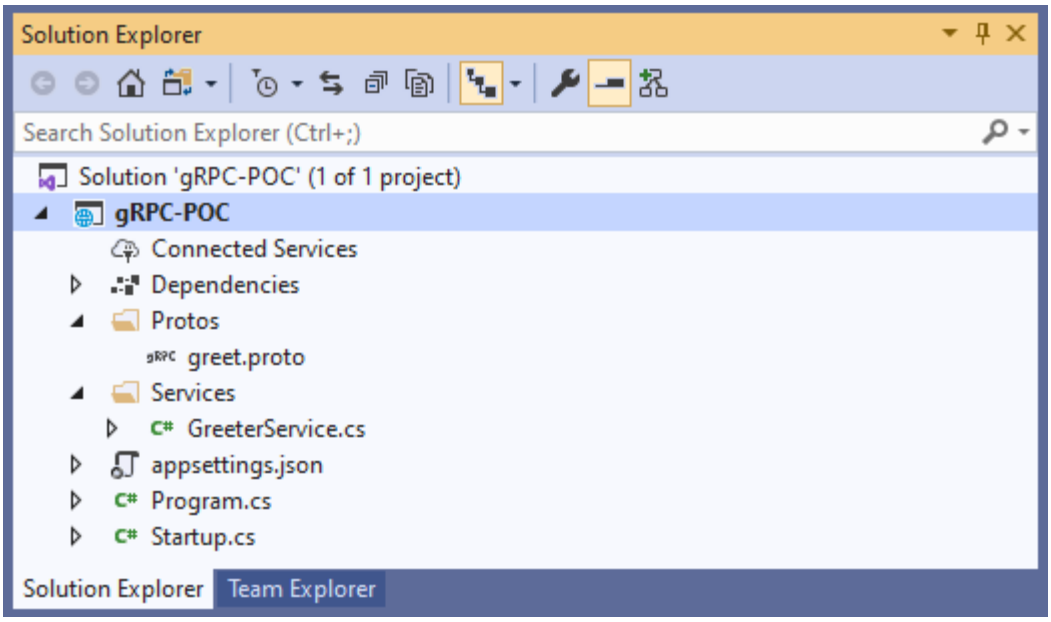


Figure 4-21. gRPC project in Visual Studio 2019

gPRC Usage

gRPC is well suited for the following scenarios:

- Low latency and high throughput communication. gRPC is great for lightweight microservices where efficiency is critical.
- Point-to-point real-time communication. gRPC has excellent support for bi-directional streaming. gRPC services can push messages in real time without polling.
- Polyglot environments – gRPC tooling supports most popular development languages, making it a good choice for multi-language environments.
- Network constrained environments – gRPC messages are serialized with Protobuf, a lightweight message format. A gRPC message is always smaller than an equivalent JSON message.

At of the writing of this book, most browsers have limited support for gRPC. gRPC heavily uses HTTP/2 features and no browser provides the level of control required over web requests to support a gRPC client. gRPC is typically used for internal microservice to microservice communication. Figure 4-22 shows a simple, but common usage pattern.

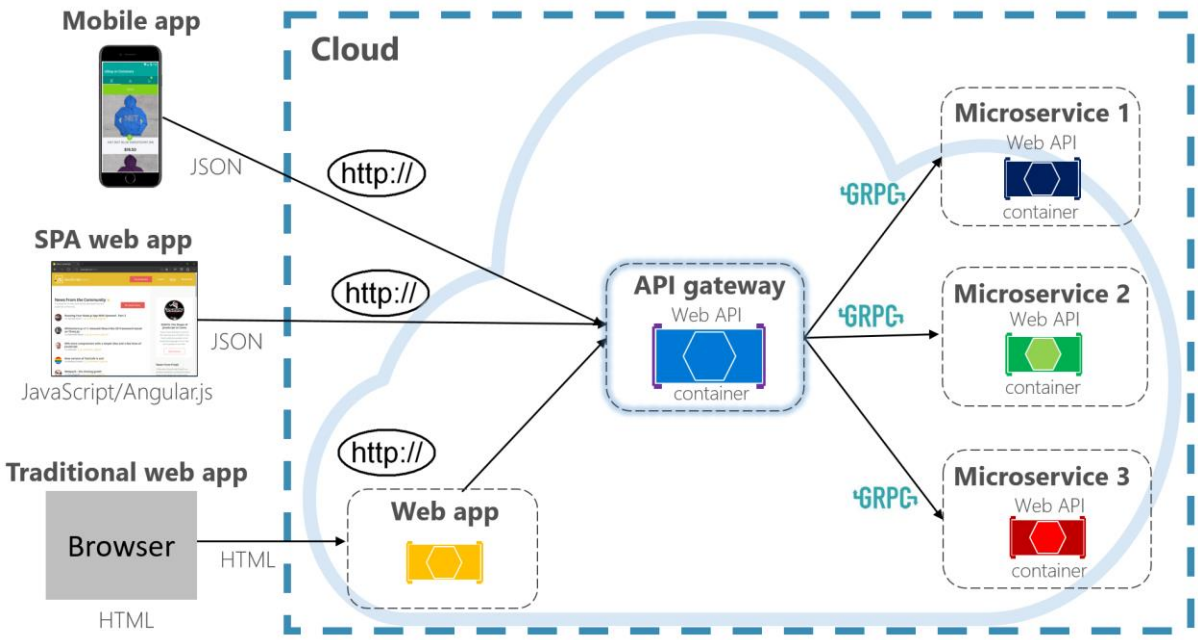


Figure 4-22. gRPC usage patterns

Note in the previous figure how front-end traffic is invoked with HTTP while back-end microservice to microservice uses gRPC.

Looking ahead, gRPC could play a major role in dethroning the dominance of REST for cloud-native systems. The performance benefits and ease of development are too good to pass up. However, make no mistake, REST will still be around for a long time. It still excels for publicly exposed APIs and for backward compatibility reasons.

Service Mesh communication infrastructure

Throughout this chapter, we've explored the challenges of microservice communication. We said that development teams need to be sensitive to how back-end services communicate with each other. Ideally, the less inter-service communication, the better. However, avoidance isn't always possible as back-end services often rely on one another to complete operations.

We explored different approaches for implementing synchronous HTTP communication and asynchronous messaging. In each of the cases, the developer is burdened with implementing communication code. Communication code is complex and time intensive. Incorrect decisions can lead to significant performance issues.

A more modern approach to microservice communication centers around a new and rapidly evolving technology entitled *Service Mesh*. A service mesh is a configurable infrastructure layer with built-in capabilities to handle service-to-service communication, resiliency, and many cross-cutting concerns. It moves the responsibility for these concerns out of the microservices and into service mesh layer. Communication is abstracted away from your microservices.

A key component of a service mesh is a proxy. In a cloud-native application, an instance of a proxy is typically colocated with each microservice. While they execute in separate processes, the two are closely linked and share the same lifecycle. This pattern, known as the Sidecar pattern, and is shown in Figure 4-23.

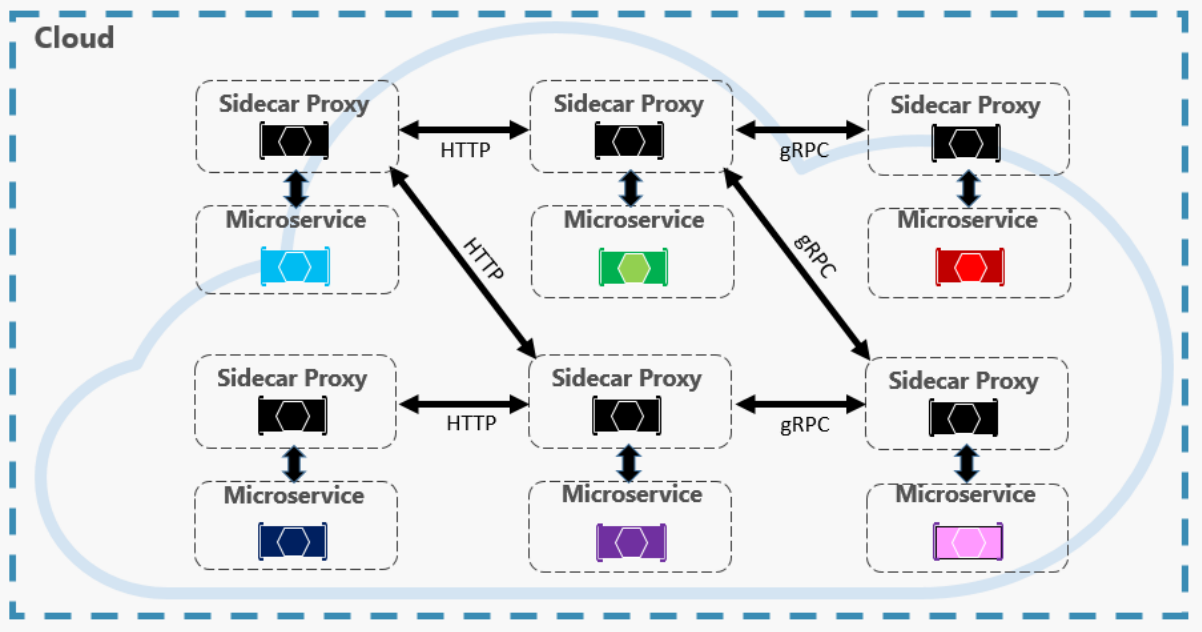


Figure 4-23. Service mesh with a side car

Note in the previous figure how messages are intercepted by a proxy that runs alongside each microservice. Each proxy can be configured with traffic rules specific to the microservice. It understands messages and can route them across your services and the outside world.

Along with managing service-to-service communication, the Service Mesh provides support for service discovery and load balancing.

Once configured, a service mesh is highly functional. The mesh retrieves a corresponding pool of instances from a service discovery endpoint. It sends a request to a specific service instance, recording the latency and response type of the result. It chooses the instance most likely to return a fast response based on different factors, including the observed latency for recent requests.

A service mesh manages traffic, communication, and networking concerns at the application level. It understands messages and requests. A service mesh typically integrates with a container orchestrator. Kubernetes supports an extensible architecture in which a service mesh can be added.

In chapter 6, we deep-dive into Service Mesh technologies including a discussion on its architecture and available open-source implementations.

Summary

In this chapter, we discussed cloud-native communication patterns. We started by examining how front-end clients communicate with back-end microservices. Along the way, we talked about API Gateway platforms and real-time communication. We then looked at how microservices communicate

with other back-end services. We looked at both synchronous HTTP communication and asynchronous messaging across services. We covered gRPC, an upcoming technology in the cloud-native world. Finally, we introduced a new and rapidly evolving technology entitled Service Mesh that can streamline microservice communication.

Special emphasis was on managed Azure services that can help implement communication in cloud-native systems:

- Azure Application Gateway
- Azure API Management
- Azure SignalR Service
- Azure Storage Queues
- Azure Service Bus
- Azure Event Grid
- Azure Event Hub

We next move to distributed data in cloud-native systems and the benefits and challenges that it presents.

References

- .NET Microservices: Architecture for Containerized .NET applications
- Designing Interservice Communication for Microservices
- Azure SignalR Service, a fully managed service to add real-time functionality
- Azure API Gateway Ingress Controller
- About Ingress in Azure Kubernetes Service (AKS)
- Practical gRPC
- gRPC Documentation
- gRPC for WCF Developers [Mark's gRPC book]
- Comparing gRPC Services with HTTP APIs

Distributed data for cloud-native apps

When constructing a cloud-native system that consists of many independent microservices, the way you think about data storage changes.

Traditional monolithic applications favor a centralized data store shown in Figure 5-1.

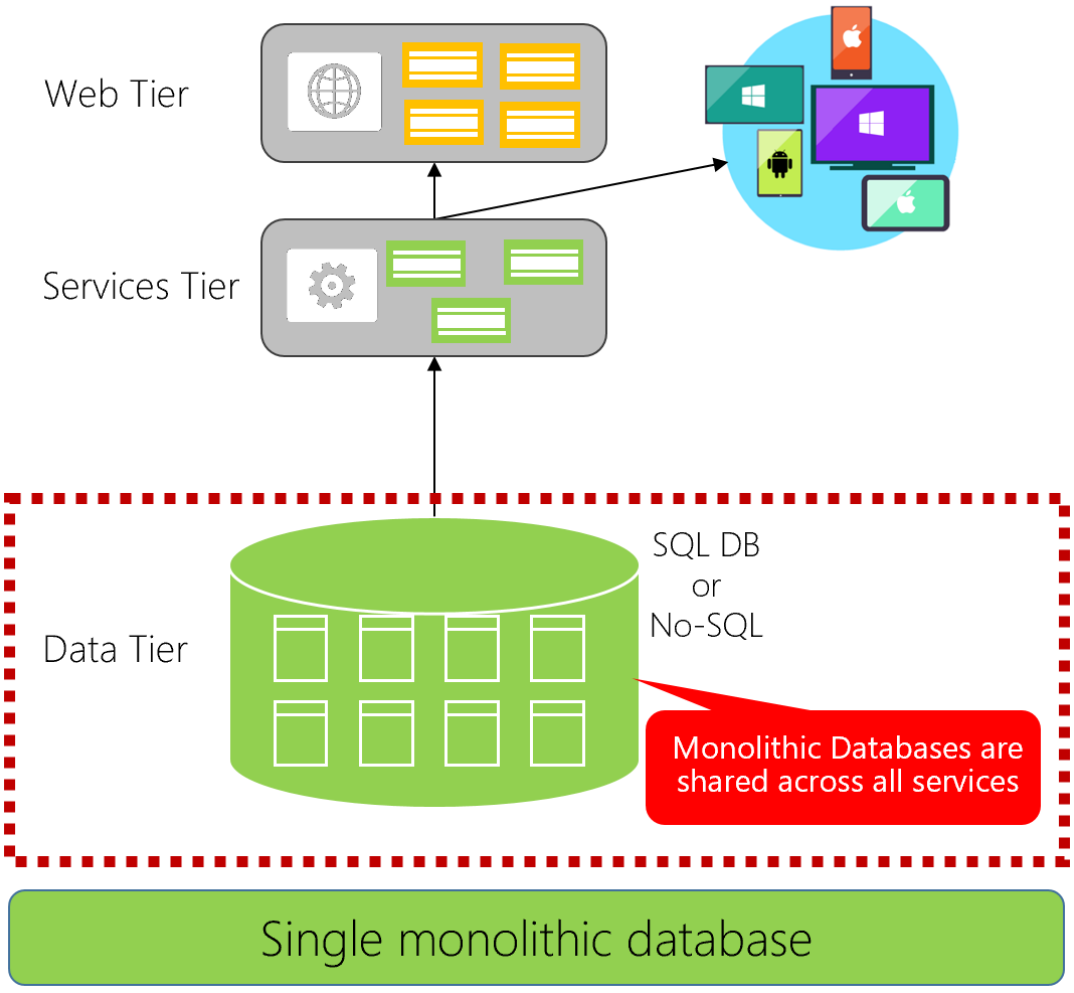


Figure 5-1. Single monolithic database

Note in the previous figure how all of the application components consume a single relational database.

There are many benefits to this approach. It's straightforward to query data spread across multiple tables, and it's straightforward to implement ACID transactions that ensure data consistency. You always end up with *immediate consistency*: either all your data updates or none of it does.

Cloud-native systems favor a data architecture shown in Figure 5-2 in which each microservice owns and encapsulates its own data.

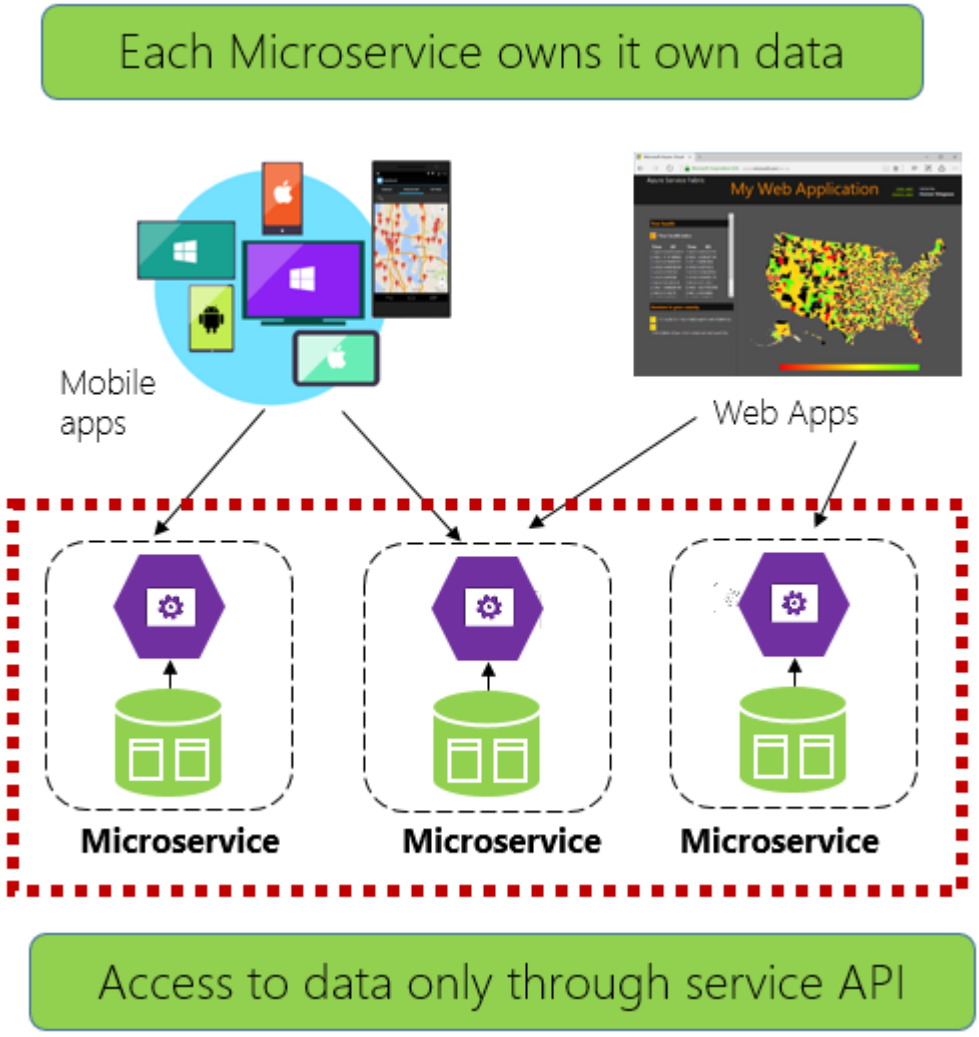


Figure 5-2. Multiple databases across microservices

Note how in the previous figure each microservice owns and encapsulates it data store and only exposes data to the outside world from its public API.

This model enables each microservice to evolve independently without having to coordinate data schema changes with other microservices. Each microservice is free to implement the data store (relational database, document database, key-value store) type that best matches its needs. At runtime, each microservice can scale its data accordingly. This is shown in Figure 5-3:

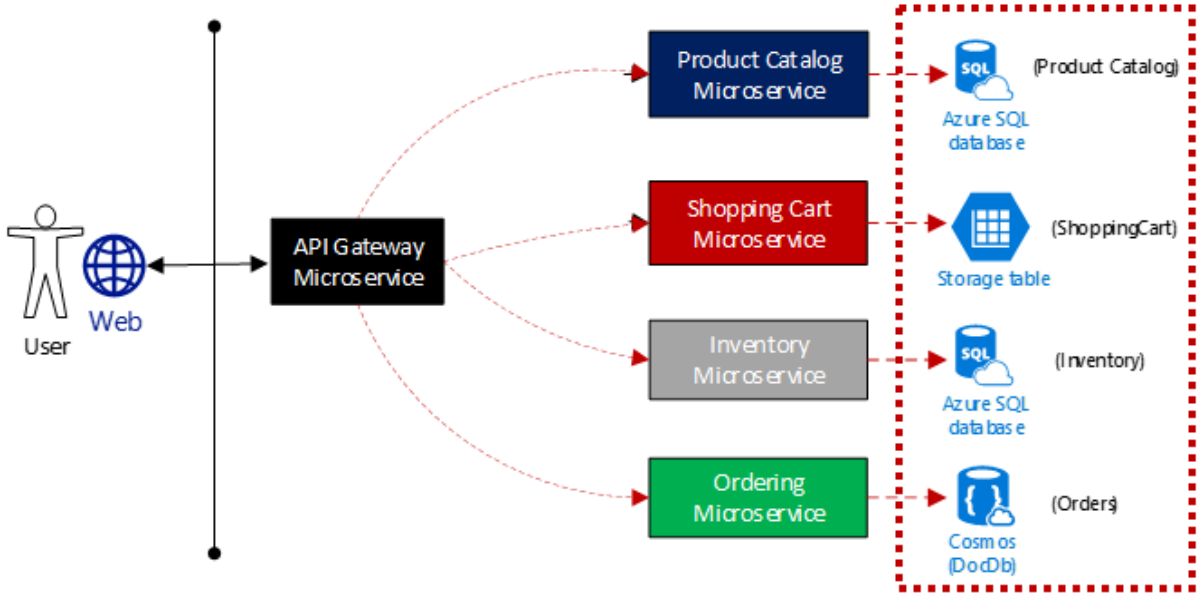


Figure 5-3. Polyglot data persistence

Note how in the previous figure the product catalog and inventory microservices adopt relational databases, the ordering microservice, a NoSql document database, and the shopping cart microservice, which is an external key-value store. While relational databases remain relevant for microservices with complex data, NoSQL databases have gained considerable popularity, providing adaptability, fast lookup, and high availability. Their schemaless nature allows developers to move away from an architecture of typed data classes and ORMs that make change expensive and time-consuming.

Cloud-native data patterns

While decentralized data can lead to improved performance, scalability, and cost savings, it also presents many challenges. Querying for data across microservices is complex. A transaction that spans microservices must be managed programmatically as distributed transactions aren't supported in cloud-native applications. You move from a world of *immediate consistency* to *eventual consistency*.

We discuss these challenges now.

Cross-service queries

How does an application query data that is spread across many independent microservices?

Figure 5-4 shows this scenario.

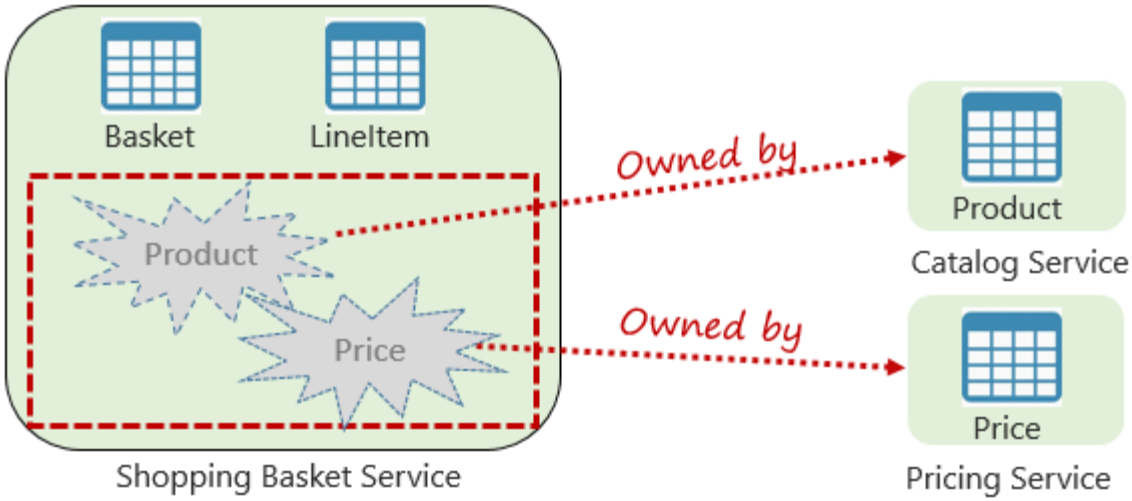


Figure 5-4. Querying across microservices

Note how in the previous figure we see a shopping basket microservice that adds an item to a user's shopping cart. While the shopping basket's data store contains a basket and lineItem table, it doesn't contain product or pricing data as those items are found in the product and price microservices. To add an item, the shopping basket microservice needs product data and pricing data. What are options to obtain the product and pricing data?

Figure 5-5 shows the shopping basket microservice making a direct HTTP call to both the product catalog and pricing microservices.

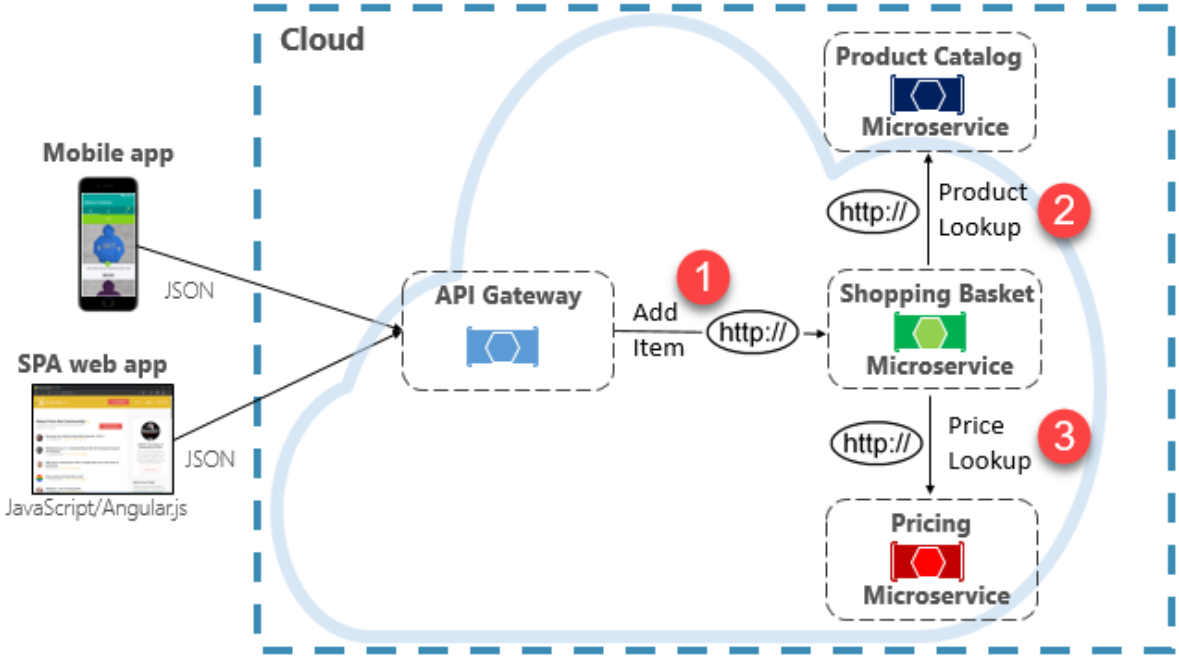


Figure 5-5. Direct HTTP communication

While feasible to implement, in chapter 4 we discussed how direct HTTP calls across microservices couple the system and aren't considered a good practice.

We could implement an aggregator microservice shown in Figure 5-6.

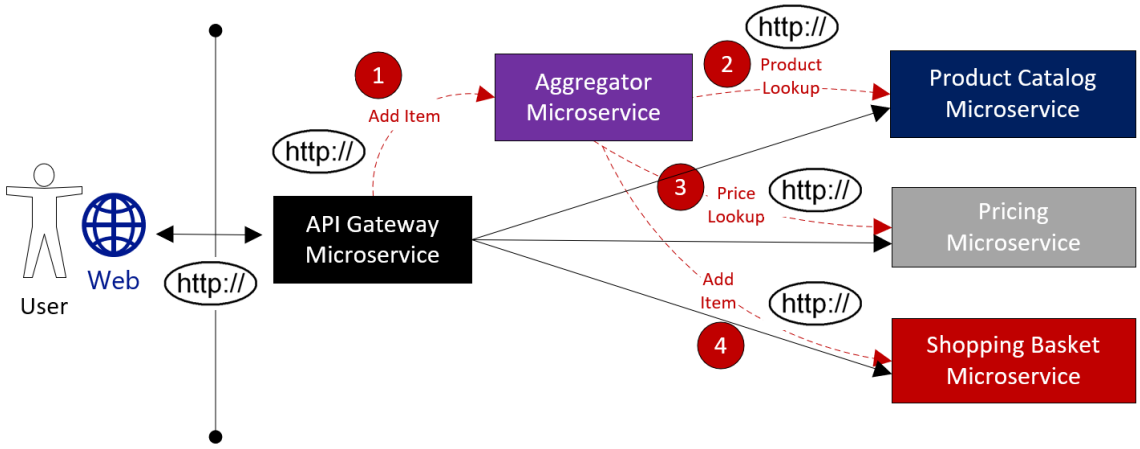


Figure 5-6. Aggregator microservice

While this approach encapsulates the business operation workflow in an individual microservice, it adds complexity and still results in direct HTTP calls.

A common approach for executing cross-service queries uses the Materialized View Pattern, shown in Figure 5-7.

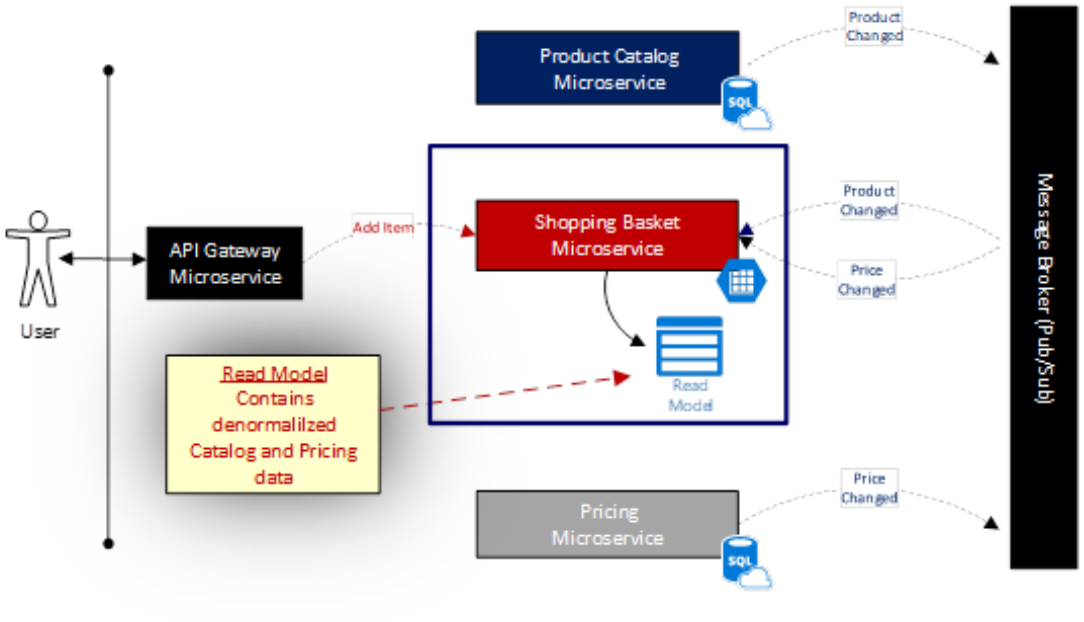


Figure5-7. Materialized View Pattern

With this pattern, you directly place a local table (known as a *read model*) in the shopping basket service that contains a denormalized copy of the data that is needed from the product and pricing microservices. Placing that data inside the shopping basket microservice eliminates the need for

invoking expensive cross-service calls. With the data local to the service, you improve response time and reliability.

The catch with this approach is you now have duplicate data in your system. In cloud-native systems, duplicate data isn't considered an anti-pattern and is commonly implemented in cloud-native systems. However, one and only one system can be the owner of any dataset, and you'll need to implement a synchronization mechanism for the system of record to update all of the associated read models, whenever a change to its underlying data occurs.

Transactional support

While queries across microservices are challenging, implementing a transaction across microservices can be complex. The inherent challenge of maintaining data consistency across data sources that reside in different microservices can't be understated. Figure 5-8 shows the problem.

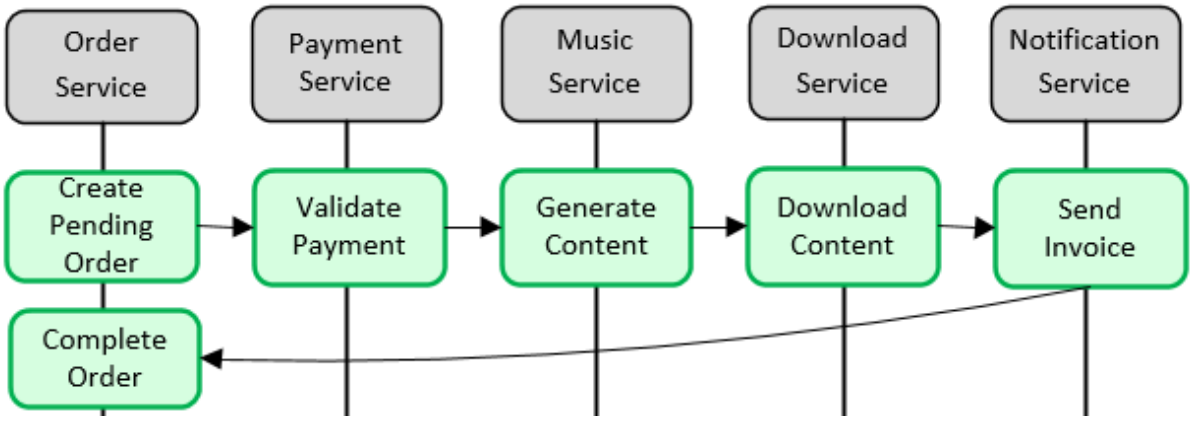


Figure 5-8. Implementing a transaction across microservices

Note how in the previous figure five independent microservices all participate in a distributed *Create Order* transaction. However, the transaction for each of the five individual microservices must succeed, or all must abort and roll-back the operation. While built-in transactional support is available inside each of the microservices, there's no support for a distributed transaction across all five services.

Since transactional support is essential for this operation to keep the data consistent in each of the microservices, you have to programmatically construct a distributed transaction.

A popular pattern for programmatically adding transactional support is the Saga pattern. It's implemented by grouping local transactions together and sequentially invoking each one. If a local transaction fails, the Saga aborts the operation and invokes a set of compensating transactions to undo the changes made by the preceding local transactions. Figure 5-9 shows a failed transaction with the Saga pattern.

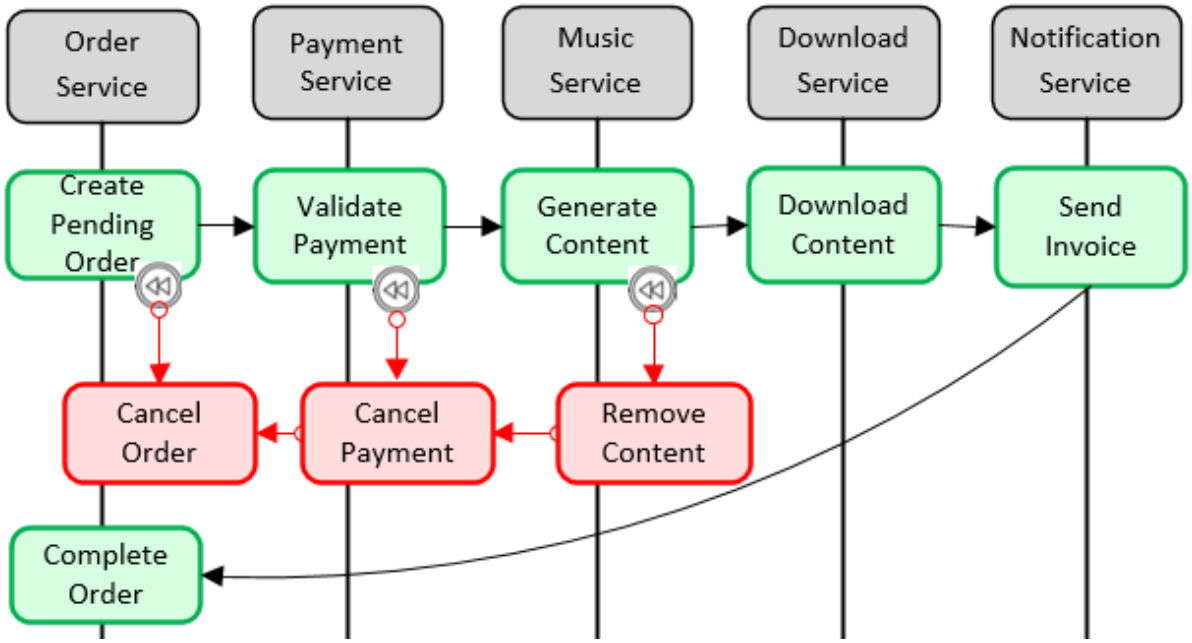


Figure 5-9. Rolling back a transaction

Note how in the previous figure the *GenerateContent* operation has failed in the music microservice. The Saga invokes compensating transactions (in red) to remove the content, cancel the payment, and cancel the order, returning the data for each microservice back to a consistent state.

Saga patterns are typically choreographed as a series of related events or orchestrated as a set of related commands.

CQRS pattern

CQRS, or Command and Query Responsibility Segregation, is an architectural pattern that separate operations that read data from those that write data. This pattern can help maximize performance, scalability, and security.

In normal data access scenarios, you implement a single model (entity and repository object) that perform *both* read and write data operations.

However, a more advanced data access scenario might benefit from separate models and data tables for reads and writes. To improve performance, the read operation, known as a *query*, might query against a highly denormalized representation of the data to avoid expensive repetitive table joins. Whereas the *write* operation, known as a *command*, might update against a fully normalized representation of the data. You would then need to implement a mechanism to keep both representations in sync. Typically, whenever the write table is modified, it raises an event that replicates the data modification to the read table.

Figure 5-10 shows an implementation of the CQRS pattern.

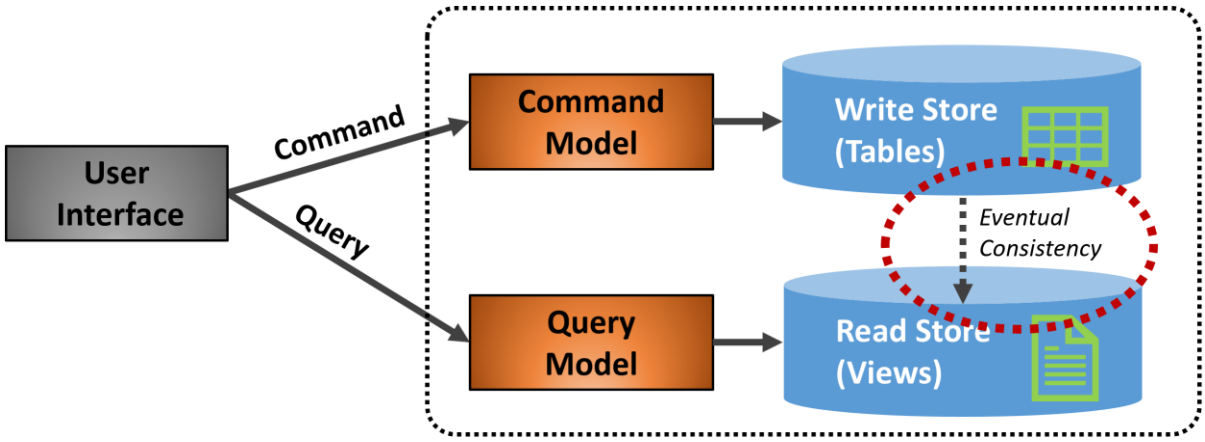


Figure 5-10. CQRS implementation

Note how in the previous figure separate command and query models are implemented. Moreover, each data write operation is saved to the write store and then propagated to the read store. Pay close attention to how the propagation process operates on the principle of eventual consistency, whereas the read model eventually synchronizes with the write model, but there may be some lag in the process.

By implementing separation, you have the ability to scale reads and writes separately. As well, you might impose tighter security on write operations than those concerning reads.

Typically, CQRS patterns are applied to limited sections of your system based upon specific needs.

Relational vs NoSQL

The impact of NoSQL technologies can't be overstated, especially for distributed cloud-native systems. The proliferation of new data technologies in this space has disrupted solutions that once exclusively relied on relational databases.

On the one side, relational databases have been a prevalent technology for decades. They're mature, proven, and widely implemented. Competing database products, expertise and tooling abounds. Relational databases provide a store of related data tables. These tables have a fixed schema, use SQL (Structured Query Language) to manage data and have ACID (also known as Atomicity, Consistency, Isolation, and Durability) guarantees.

No-SQL databases, on the other side, refer to high-performance, non-relational data stores. They excel in their ease-of-use, scalability, resilience, and availability characteristics. Instead of joining tables of normalized data, NoSQL stores self-describing (schemaless) data typically in JSON documents. They don't offer ACID guarantees.

A way to understand the differences between these types of databases can be found in the CAP theorem, a set of principles that can be applied to distributed systems that store state. Figure 5-11 shows the three properties of the CAP theorem.

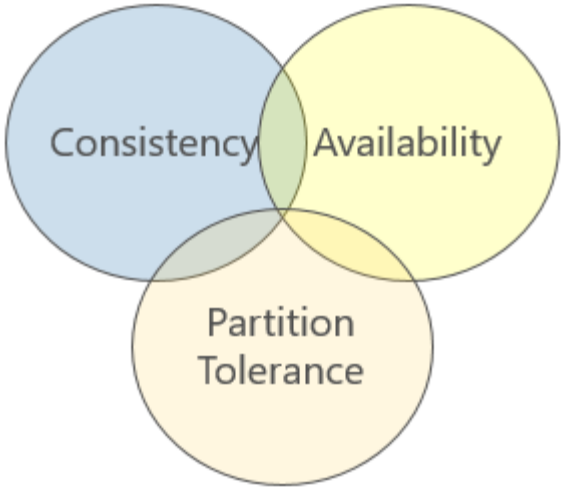


Figure 5-11. The CAP theorem

The theorem states that any distributed data system will offer a trade-off between consistency, availability, and partition tolerance, and that any database can only guarantee two of the three properties:

- *Consistency*: every node in the cluster will respond with the most recent data, even if it requires blocking a request until all replicas are correctly updated.
- *Availability*: every node will return a response in a reasonable amount of time, even if that response isn't the most recent data.
- *Partition Tolerance*: guarantees that the system will continue operating if a node fails or loses connectivity with another.

Relational databases exhibit consistency and availability, but not partition tolerance. Partitioning a relational database, such as sharding, is difficult and can impact performance.

On the other hand, NoSQL databases typically exhibit partition tolerance, known as horizontal scalability, and high availability. As the CAP theorem specifies, you can only have two of the three principles, and you lose the consistency property.

NoSQL databases are distributed and commonly scaled out across commodity servers. Doing so can provide great availability, both within and across geographical regions at a reduced cost. Data can be partitioned and replicated across these machines, or nodes, providing redundancy and fault tolerance. The downside is consistency. A change to data on one NoSQL node can take some time to propagate to other nodes. Typically, a NoSQL database node will provide an immediate response to a query, even if the data that it is presenting is stale and has not been updated yet.

This is known eventual consistency, a characteristic of distributed data systems where ACID transactions aren't supported. It's a brief delay between the update of a data item and time that it takes to propagate that update to each of the replica nodes. If you update a product item in a NoSQL database in the United States, but at same time query that same data item from a replica node in Europe, you might retrieve the earlier product information - until the European node has been updated with product change. The trade-off is that by giving up strong consistency, waiting for all

replica nodes to update before returning a query result, you can support enormous scale and traffic volume, but with the possibility of presenting older data.

NoSQL databases can be categorized by the following four models:

- *Document Store* (MongoDB, CouchDB, Couchbase): data (and corresponding metadata) is stored non-relationally in denormalized JSON-based documents inside the database.
- *Key/Value Store* (Redis, Riak, memcached): data is stored in simple key-value pairs with system operations performed against a unique access key that is mapped to a value of user data.
- *Wide-Column Store* (HBase, Cassandra): related Data is stored in a columnar format as a set of nested-key/value pairs within a single column with data typically retrieved as a single unit without having to join multiple tables together.
- *Graph stores* (neo4j, titan): data is stored as a graphical representation within a node along with edges that specify the relationship between the nodes.

NoSQL databases can be optimized to deal with large-scale data, especially when the data is relatively simple. Consider a NoSQL database when:

- Your workload requires large-scale and high-concurrency.
- You have large numbers of users.
- Your data can be expressed simply without relationships.
- You need to geographically distribute your data.
- You don't need ACID guarantees.
- Will be deployed to commodity hardware.

Then, consider a relational database when:

- Your workloads require medium to large scale.
- Concurrency isn't a major concern.
- ACID guarantees are needed.
- Data is best expressed relationally.
- Your application will be deployed to large, high-end hardware.

Next, we look at data storage in the Azure cloud.

Data storage in Azure

As we've seen throughout this book, the cloud is changing the way applications are designed, deployed, and managed. When moving to the cloud, a critical question is how do you move your data? Fortunately, the Azure cloud offers many options.

You could just provision an Azure virtual machine and install your database of choice. This is known as Infrastructure as a Service (IaaS). This approach simplifies moving an on-premises database to the cloud, as-is, but shifts the burden of managing the virtual machine and the database to you.

Instead, a fully managed Database as a Service (DBaaS) is a better option. You get many built-in features while the hosting, maintenance, and licensing are managed by Microsoft. Azure features different kinds of fully managed data storage options, each with specific benefits. They all support just-in-time capacity and a pay-as-you-go model.

We'll next look at DBaaS options available in Azure. You'll see how Microsoft continues is commitment to keeping Azure an "open platform," offering managed support for many open-source relational and NoSQL databases and making key contributions to the various open-source foundations as an active member.

Azure SQL Database

Azure SQL Database is a feature-rich, general-purpose relational database-as-a-service (DBaaS) based on the Microsoft SQL Server Database Engine. It's fully managed by Microsoft and is a high-performance, reliable, and secure cloud database. The service shares many of the features found in the on-premises version of SQL Server.

You can provision a SQL Database server and database in minutes. When demand for your application grows from a handful of customers to millions, Azure SQL Database scales on-the-fly with minimal downtime. You can dynamically add or remove resources, including CPU power, memory, IO throughput, and storage allocated to your databases.

Figure 5-12 shows the deployment options for Azure SQL Database.

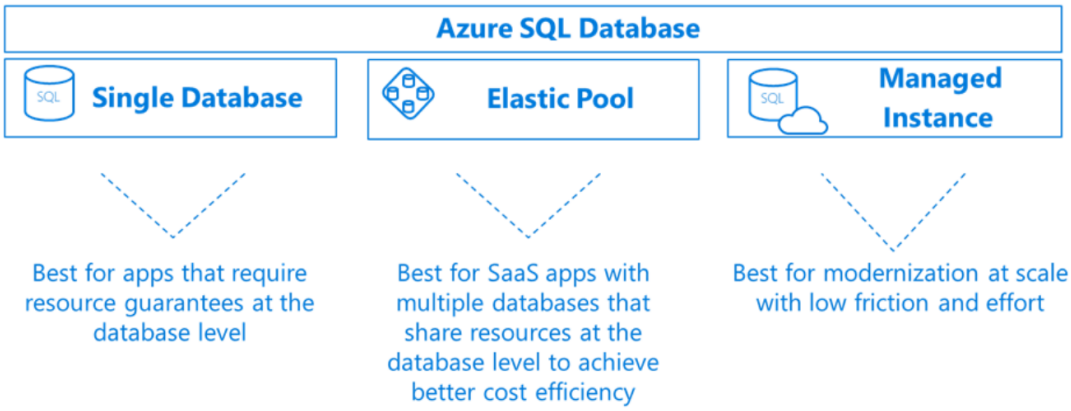


Figure 5-12. Azure SQL deployment options

Note the alternatives in the previous figure when deploying a SQL Database:

- A Single database with its own set of resources managed by a SQL Database server. A single database is similar to a contained database in an on-premises SQL Server deployment.
- An Elastic pool in which a collection of SQL Databases share a single SQL Database server at a set price. Single databases can be moved in and out of an elastic pool as needed to optimize the price performance for a group of databases.
- A Managed Instance in which is a collection of system and user databases provide near-100% compatibility with an on-premises SQL Server. This option supports larger databases, up to 35 TB and is placed in an Azure Virtual Network for better isolation.

Azure SQL Database is a fully managed Platform as a Service (PaaS) Database Engine that handles upgrading, patching, backups, and monitoring without user involvement. It always runs the latest stable version of the SQL Server Database Engine and patched OS and guarantees 99.99% availability. One feature, active geo-replication, lets you create readable secondary databases in the same or a different Azure data center. Upon failure, a failover to a secondary database can be initiated. At that point, the other secondaries automatically link to the new primary. Up to four secondary replicas are supported in either the same or in different regions, and these secondaries can also be used for read-only access queries.

Azure SQL Database includes built-in monitoring and intelligent tuning features that can help you maximize performance and reduce operational costs. For example, the Automatic Tuning feature provides continuous performance tuning based on AI and machine learning. The service learns from your running workloads and can apply tuning recommendations. The longer an Azure SQL Database runs with automatic tuning enabled, the better it performs.

Azure SQL Database serverless (available for preview at time of the writing of this book) is a compute tier for single databases that automatically scales based on workload demand, and bills for the amount of compute used per second. The serverless compute tier also automatically pauses databases during inactive periods so that only storage charges are billed. It automatically resumes when activity returns.

Finally, there's the new Azure SQL Database Hyperscale pricing tier. It's powered by a highly scalable storage architecture and enables your database to grow as needed, eliminating the need to pre-provision storage resources. You can scale compute and storage resources independently, providing the flexibility to optimize performance for each workload. Azure SQL Database Hyperscale is optimized for OLTP processing and high throughput analytic workloads with storage up to 100 TB. With read-intensive workloads, Hyperscale provides rapid scale-out by provisioning additional read replicas as needed for offloading read workloads.

In addition to the traditional Microsoft SQL Server stack, Azure also features managed versions of several popular open-source databases.

Azure Database for MySQL

MySQL is an open-source relational database. It's a component in the LAMP software stack and used by many large organizations, including Facebook, Twitter, and YouTube. The community edition is available for free and the enterprise edition requires a license purchase. Originally created in 1995, the product was purchased by Sun Microsystems in 2008, which was acquired by Oracle in 2010.

Azure Database for MySQL is a fully managed, enterprise-ready relational database service based on the open-source MySQL Server engine. Implementing the MySQL Community edition, it includes the following PaaS capabilities at no additional cost:

- Built-in high availability.
- Predictable performance, using inclusive pay-as-you-go pricing.
- Scale as needed within seconds.
- Secured to protect sensitive data at-rest and in-motion.

- Automatic backups and point-in-time-restore for up to 35 days.
- Enterprise-grade security and compliance.

These built-in PaaS features are important for organizations who have hundreds of "tactical" (non-strategic) databases in their data centers, but don't have the resources to perform patching, backup, security, and performance monitoring.

Additionally, the Azure Data Migration Service can migrate data from multiple database sources to Azure Data platforms with minimal downtime. The service generates assessment reports and provides recommendations to guide you through the changes required to performing a migration, both small or large.

The managed Azure MySQL server is the central administrative point for the service. It's the same MySQL server engine used for on-premises deployments. With it, you can create a single database per server to consume all resources or create multiple databases per server to share resources. Your team can continue to develop applications with the open-source tools and platform of your choice without having to learn new skills or manage virtual machines and infrastructure.

Azure Database for MariaDB

MariaDB Server is another popular open-source database server. It was created as a fork of MySQL by the original developers of MySQL at the time that Oracle purchased Sun Microsystems who owned MySQL. The intent was to ensure that MariaDB remained open-source.

Since MariaDB is a fork of MySQL, the data and table definitions are compatible, and the client protocols, structures, and APIs, are close-knit. MySQL data connectors will work MariaDB without modification.

MariaDB has a strong following and is used by many large enterprises. While Oracle continues to maintain, enhance and support MySQL, MariaDB is managed by the MariaDB Foundation allowing public contributions to the product and documentation.

Azure Database for MariaDB is a fully managed database as a service in the Azure cloud. It's based on the MariaDB community edition server engine. It can handle mission-critical workloads with predictable performance and dynamic scalability. Similar to the other Azure Database platforms, it includes many Platform-as-a-Service capabilities at no additional cost:

- Built-in high availability.
- Predictable performance, using inclusive pay-as-you-go pricing.
- Scaling as needed within seconds.
- Secured protection of sensitive data at rest and in-motion.
- Automatic backups and point-in-time-restore for up to 35 days.
- Enterprise-grade security and compliance.

Azure Database for PostgreSQL

PostgreSQL is another popular, open-source relational database with over 30 years of active development. It's a general purpose and object-relational database management system. Its licensing is considered to be "liberal" and the product is free to use, modify, and distribute in any form. Many large enterprises including Apple, Red Hat, and Fujitsu have built products using PostgreSQL.

Azure Database for PostgreSQL is a fully managed relational database service, based on the open-source Postgres database engine. It can handle mission-critical workloads with predictable performance, security, high availability, and dynamic scalability. It supports several open-source frameworks and languages—including C++, Java, Python, Node, C#, and PHP. It enables migration of PostgreSQL databases through a command-line interface or the Azure Data Migration Service.

The service includes built-in intelligence that studies your unique database patterns and provides customized recommendations and insights to help you maximize the performance of your PostgreSQL database. Advanced Threat Protection monitors your database around the clock and detects potential malicious activities, alerting you upon detection so you can intervene right away.

Azure Database for PostgreSQL is available as two deployment options: Single Server and Hyperscale (Citus), available for preview at time of the writing of this book

- The Single Server deployment option is a central administrative point for multiple databases. It's the same PostgreSQL server engine available for on-premises deployments. With it, you can create a single database per server to consume all resources or create multiple databases to share the resources. The pricing is structured per-server based upon cores and storage.
- The Hyperscale (Citus) option is powered by Citus Data technology. It enables high-performance scaling by horizontally scaling a single database across hundreds of nodes to deliver blazingly fast performance and scale. This option allows the engine to fit more data in memory, parallelize queries across hundreds of nodes, and index data faster. The Hyperscale feature is compatible with the latest innovations, versions, and tools for PostgreSQL, so you can leverage your existing PostgreSQL expertise.

Cosmos DB

Azure Cosmos DB is a fully managed, globally distributed NoSQL database service that's designed to provide low latency, elastic scalability, managed data consistency, and high availability. In short, if your application needs guaranteed fast response time anywhere in the world, if it's required to be always online and needs unlimited and elastic scalability of throughput and storage, Cosmos DB is a great choice. Figure 5-13 shows a high-level overview of Cosmos DB.

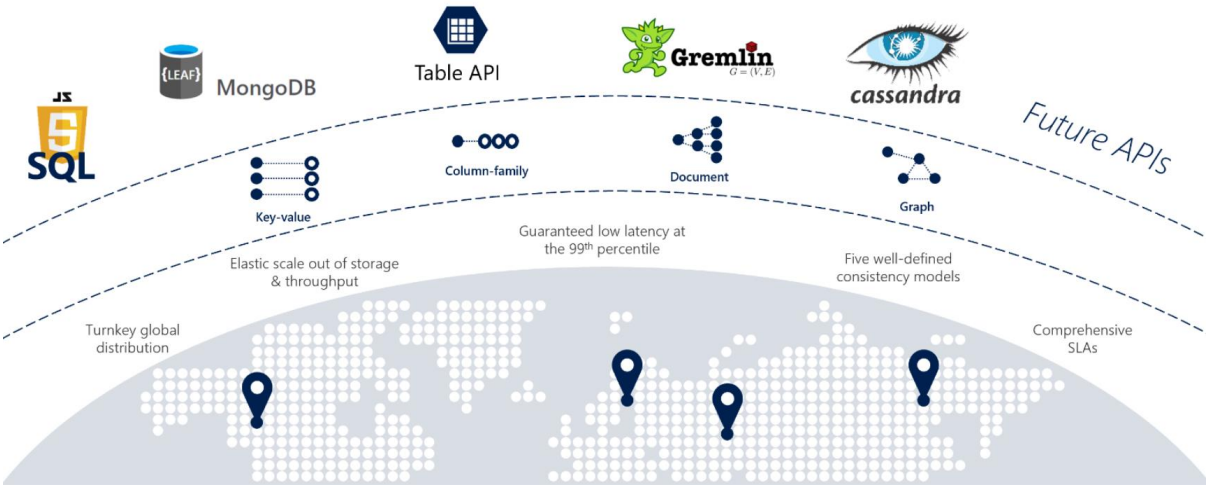


Figure 5-13: Overview of Cosmos DB

Note in the Figure 5-13 how Cosmos DB is a robust and highly versatile database service with many built-in cloud-native capabilities. In this section, we'll take a closer look at them.

Global Support

You can globally distribute Cosmos databases across all Azure regions across the world, placing data close to your users, improving response time, and reducing latency. You can add or remove a database from a region without pausing or redeploying your application. In the background, Cosmos DB transparently replicates the data to all of the configured regions.

Cosmos DB supports active/active clustering at the global level, enabling you to configure any or all your database regions to support both writes and reads.

The Multi-Master protocol feature in Cosmos DB enables the following functionality:

- Unlimited elastic write and read scalability.
- 99.999% read and write availability all around the world.
- Guaranteed reads and writes served in less than 10 milliseconds at the 99th percentile.

Internally, Cosmos DB handles data replication between regions with consistency level guarantees and financially backed service level agreements.

With the Cosmos DB Multi-Homing APIs, your application can automatically become aware of the nearest Azure region and send requests to it. The nearest region is identified by Cosmos DB without any configuration changes. Should a region become unavailable, Cosmos DB supports automatic failover, and the Multi-Homing feature will automatically route your request to the next nearest available region.

Multi-Model Support

Cosmos DB is a *multi-model data platform* enabling you to interact with your data using a number of supported NoSQL models, including documents, key-value pairs, wide-column, and graph

representations. Internally, data is stored in a simple struct format made up of primitive data types, including strings, bools, and numbers. For each request, the database engine translates data into the model representation you have selected. You can choose from a proprietary Cosmos DB SQL-based API or any of the compatibility APIs shown in Figure 5-14.

Cosmo's DB API Providers	
SQL API	Proprietary API built into Cosmos DB supporting JSON documents. Exposes data via a SQL-language interface and extends processing with JavaScript-based stored procedures, triggers and user-defined functions.
Mongo DB API	Works with Mongo DB APIs supporting JSON documents
Gremlin API	Works with the Gremlin API supporting graph-based nodes and edge representations.
Cassandra API	Works with the Cassandra APIs for Wide-Column data representations.
Table API	Enables Cosmos DB support for Azure table Storage with added premium enhancements.
etcd API	Enables you to use Cosmos DB as the backend store for Azure Kubernetes Service clusters.

Figure 5-14: Cosmos DB providers

Note in Figure 5-14 how Cosmos DB supports Table Storage. Both Cosmos DB and Azure Table Storage share the same underlying table model and expose many of the same table operations. However, the Cosmos DB Table API provides many premium enhancements not available in the Azure Storage API. These features are contrasted in Figure 5-15.

	Azure Table Storage	Azure Cosmos DB
Latency	Fast	Single-digit millisecond latency for reads and writes anywhere in the world
Throughput	Limit of 20,000 operations per table	10 Million operations per table
Global Distribution	Single region with optional single secondary read region.	Turnkey distributions to all regions with automatic failover
Indexing	Available on partition and row key only	Automatic indexing of all properties
Pricing	Based on storage	Based on throughput

Figure 5-15: Azure Table API

Applications written for Azure Table storage can migrate to Azure Cosmos DB by using the Table API with no code changes.

In [Brownfield](https://en.wikipedia.org/wiki/Brownfield_(software_development) application scenarios, development teams can migrate existing Mongo, Gremlin, or Cassandra databases into

Cosmos DB with minimal changes to the existing data or application code. For Greenfield scenarios, development teams can choose the data model that best meets their requirements and preferences, including fully supported open-source options for the MongoDB, Cassandra, and Gremlin platforms.

Consistency Models

Earlier in the *Relational versus NoSQL* section, we discussed the subject of *data consistency*, which is a term that refers to the integrity of your data. Distributed databases that rely on replication for high availability, low latency, or both, must make a fundamental tradeoff between read consistency, availability, and latency.

Most distributed databases allow developers to choose between two consistency models: strong consistency and eventual consistency. *Strong consistency* is the gold standard of data programmability. It guarantees that a query result will always return the most current data, even if the system must incur latency waiting for an update to replicate across all database copies. On the other hand, a system configured for *eventual consistency* will return data immediately, even if that data isn't the most current copy. This option enables higher availability, greater scale, and increased performance.

Azure Cosmos DB offers a spectrum of five well-defined consistency models shown in Figure 5-16. These options enable you to make precise choices and granular tradeoffs with respect to availability and performance based on the needs of your application. These models are well-defined, intuitive, and backed by the service level agreements (SLAs).

Cosmos DB Consistency Levels	
Eventual	No ordering of data is guaranteed for reads. Replicas will eventually converge.
Consistent Prefix	Reads are still eventual, but data is returned in the ordering in which it was written.
Session	Guarantees that you can read any data you have written during the current session. It is the default consistency level.
Bounded Staleness	Reads trail writes by an interval which you specify in terms of versions or time.
Strong	Reads are guaranteed to return the most recent committed version of an item. A client never sees an uncommitted or partial write.

Figure 5-16: Cosmos DB Consistency Levels

Partitioning

Azure Cosmos DB uses automatic partitioning to scale the database to meet the performance needs of your application.

You manage data in Cosmos DB data by creating databases, containers, and items, shown in Figure 5-17.

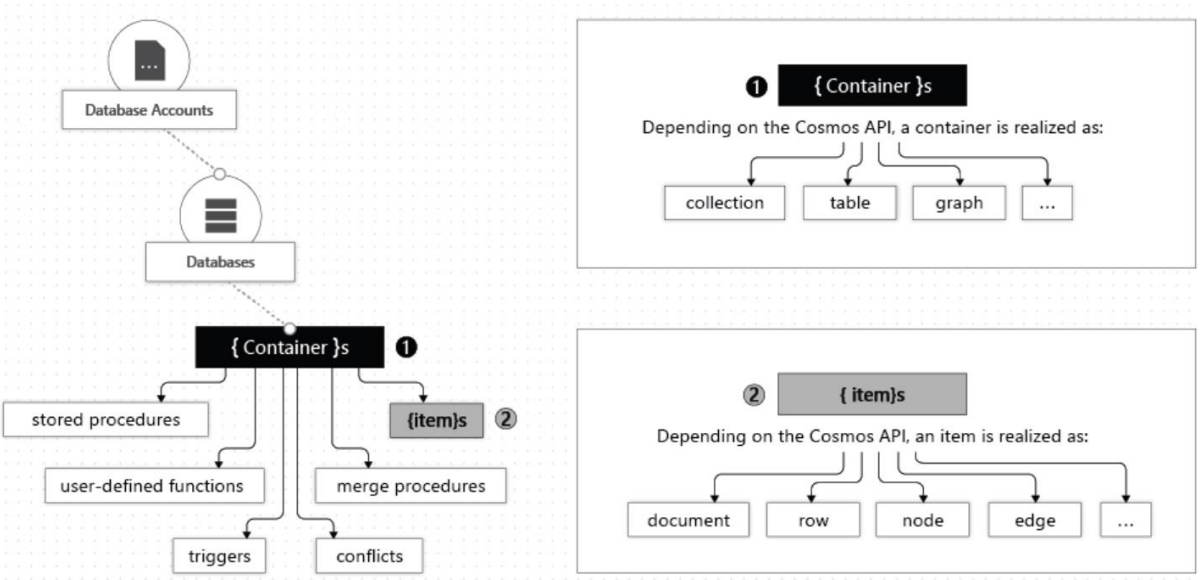


Figure 5-17: Hierarchy of Cosmos DB entities

Note in Figure 5-17 how you start by creating a Cosmos DB database inside of an Azure account. That database becomes the unit of management for a set of containers. A container is a schema-agnostic grouping of items that can be expressed as a collection, table, or graph, based on your selected API provider (discussed in the prior section). Items are the data that you add to the container and are represented as documents, rows, nodes, or edges. By default, all items that you add to a container are automatically indexed without requiring explicit index or schema management.

To partition the container, items are divided into distinct subsets called logical partitions. Logical partitions are created based on the value of a partition key that is associated with each item in a container. Figure 5-18 shows how all items in a logical partition have the same partition key value.

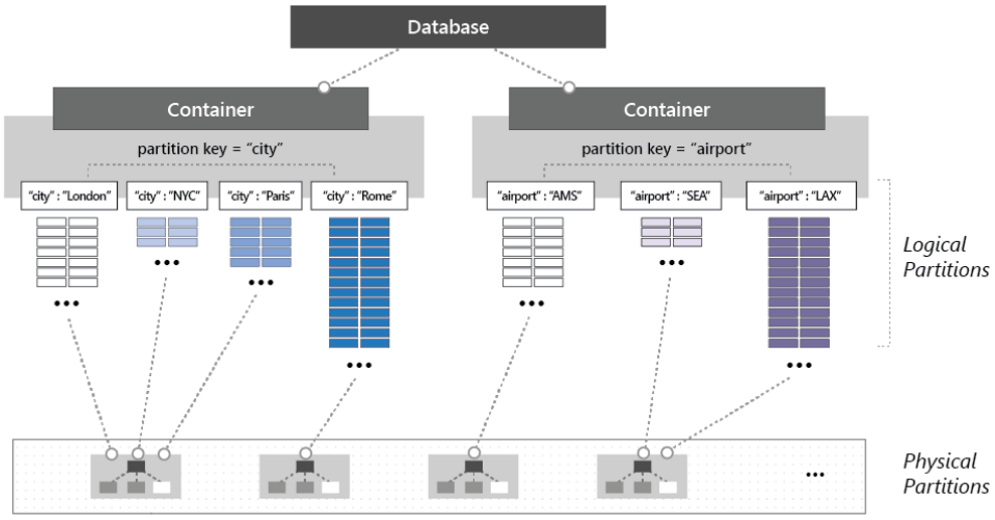


Figure 5-18: Cosmos DB partitioning mechanics

Note in Figure 5-18 how each item includes a partition key of either 'city' or 'airport'. This partition key determines the item's logical partition. Each city code is assigned to a logical partition in the container

on the left-side and those with an airport code to the container on the right. Combining the partition key value with an item's ID value creates the item's index, which uniquely identifies the item.

Internally, Cosmos DB automatically manages the placement of logical partitions on physical partitions to efficiently satisfy the scalability and performance needs of the container. As the throughput and storage requirements of an application increase, Azure Cosmos DB moves logical partitions to redistribute the load across a greater number of servers. These redistribution operations are managed by Cosmos DB and are performed without any interruption or downtime.

Azure Redis Cache

The benefits of caching to improve performance and scalability are well understood.

For a cloud-native application, a common location to add caching is inside the API Gateway. The gateway serves as a front end for all incoming requests. By adding caching, you can increase performance and responsiveness by returning cached data and avoiding round-trips to a local database or downstream service. Figure 5-19 shows a caching architecture for a cloud-native application.

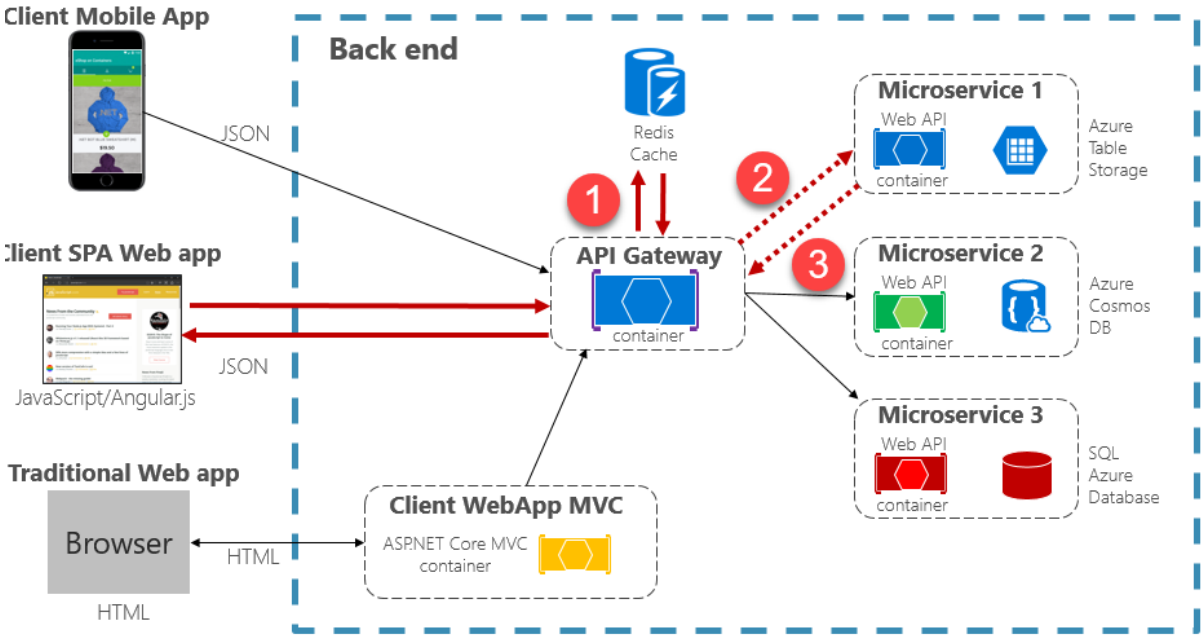


Figure 5-19: Caching in a cloud-native app

A common caching pattern is the cache-aside pattern. For an incoming request, you first query the cache for the response, shown in step #1 in Figure 5-19. If found, the data is returned immediately. If the data doesn't exist in the cache (known as a cache miss), it's retrieved from the local database or downstream service (step #2), written to the cache for future requests (step #3), and returned to the caller. Care must be taken to periodically evict cached data so that the system remains consistent and accurate.

Additionally, note in Figure 5-19 how the cache isn't implemented locally within the boundaries of the service, but instead is consumed as a cloud-based backing service, as discussed in Chapter 1.

Azure Redis Cache is a data caching and messaging broker service. It provides high throughput and low-latency access to data for applications. It's fully managed by Microsoft, hosted within Azure, and accessible to any application within or outside of Azure.

Internally, Azure Cache for Redis is backed by the open-source Redis server and natively supports data structures such as strings, hashes, lists, sets, and sorted sets. If your application uses Redis, it will work as-is with Azure Cache for Redis.

Azure Cache for Redis can also be used as an in-memory data cache, a distributed non-relational database, and a message broker. It's available in three different pricing tiers. The Premium tier features many enterprise-level features such as clustering, data persistence, geo-replication, and Virtual-network security and isolation.

Cloud-native resiliency

Resiliency is the ability of your system to react to failure and still remain functional. It isn't about avoiding failure. But it's about accepting that failure is inevitable in cloud-based systems and building your application to respond to it. The end-goal of resiliency is to return the application to a fully functioning state after a failure.

Unlike traditional monolithic applications, where everything runs together in a single process, cloud-native systems embrace distributed architecture as shown in Figure 6-1:

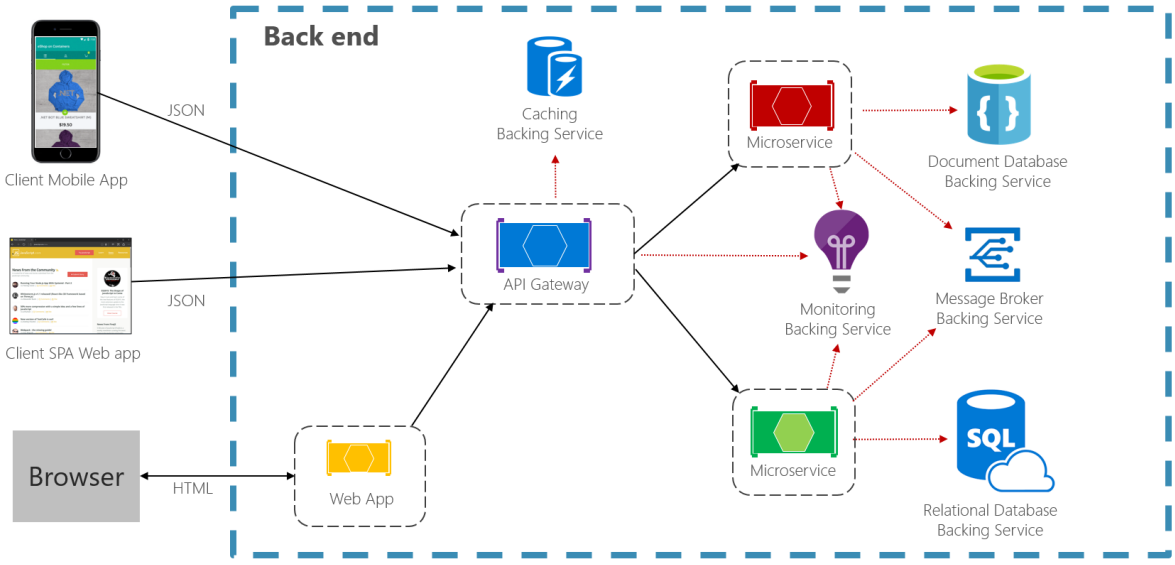


Figure 6-1. Distributed cloud-native environment

In the previous figure, note how each client, microservice, and cloud-based backing service executes as a separate process, running across different servers, all communicating via network-based calls.

So, what could go wrong?

- Unexpected network latency.
- Transient faults (temporary network connectivity errors).
- Blocking by a long-running synchronous operation.
- A host process that has crashed and is being restarted or moved.
- An overloaded microservice that can't respond for a short time.
- An in-flight DevOps operation such as an update or scaling operation.
- An Orchestrator operation such as moving a service from one node to another.
- Hardware failures from commodity hardware.

When deploying distributed services into cloud-based infrastructure, the factors from the previous list become very real and you must architect and develop defensively to deal with them.

In a small-scale distributed system, failure will be less frequent, but as a system scales up and out, you can expect to experience more of these issues to a point where partial failure becomes normal operation.

Therefore, your application and infrastructure must be resilient. In the following sections, we'll explore defensive techniques that you can add to your application and built-in cloud features that you can leverage to help bullet-proof your user's experience.

Application resiliency patterns

The first line of defense is software-enabled application resiliency.

While you could invest considerable time writing your own resiliency framework, such products already exist. For example, Polly is a comprehensive .NET resilience and transient-fault-handling library that allows developers to express resiliency policies in a fluent and thread-safe manner. Polly targets applications built with either the full .NET Framework or .NET Core. Figure 6-2 shows the resiliency policies (that is, functionality) available from the Polly Library. These policies can be applied individually or combined together.

Policy	Feature
Retry	Allows configuring automatic retries
Circuit Breaker	Blocks executions for a period when faults exceed a pre-configured threshold
Timeout	Places a limit on the duration for which a caller can wait for response
Bulkhead	Constrains actions to fixed-size resource pool to prevent failing calls from swamping a resource
Cache	Stores responses automatically
Fallback	Defines alternative value to return upon a failure

Figure 6-2. Polly resiliency framework features

Note how in the previous figure the resiliency policies apply to request messages, whether coming from an external client or another back-end service. The goal is to compensate the request for a service that might be momentarily unavailable. These short interruptions typically manifest themselves with the HTTP status codes shown in Figure 6-3.

HTTP Status Code	Cause
404	Not Found
408	Request timeout
429	Too many requests (you've most likely been throttled)
502	Bad gateway
503	Service unavailable
504	Gateway timeout

Figure 6-3. HTTP status codes to retry

Question: Would you retry an HTTP Status Code of 403 - Forbidden? No. Here, the system is functioning properly, but informing the caller that they aren't authorized to perform the requested operation. Care must be taken to retry only those operations caused by failures.

As recommended in Chapter 1, Microsoft developers constructing cloud-native applications should target .NET Core. Version 2.1 introduced the HTTPClientFactory library for creating HTTP Client instances for interacting with URL-based resources. Superseding the original HTTPClient class, the factory class supports many enhanced features, one of which is tight integration with the Polly resiliency library. With it, you can easily define resiliency policies in the application Startup class to handle partial failures and connectivity issues.

Next, let's expand on retry and circuit breaker patterns.

Retry pattern

In a distributed cloud-native environment, calls to services and cloud resources can fail because of transient (short-lived) failures, which typically correct themselves after a brief period of time. Implementing a retry strategy helps a cloud-native service handle these scenarios.

The Retry pattern enables a service to retry a failed request operation a (configurable) number of times with an exponentially increasing wait time. Figure 6-4 shows a retry in action.

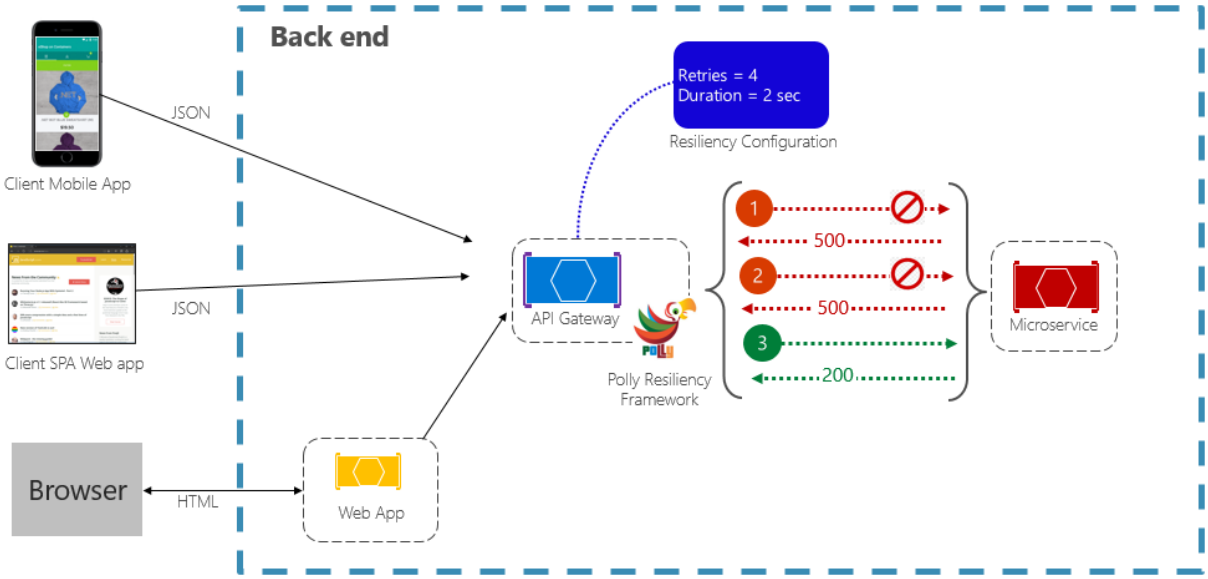


Figure 6-4. Retry pattern in action

In the previous figure, a retry pattern has been implemented for a request operation. It's configured to allow up to four retries before failing with a backoff interval (wait time) starting at two seconds, which exponentially doubles for each subsequent attempt.

- The first invocation fails and returns an HTTP status code of 500. The application waits for two seconds and reties the call.
- The second invocation also fails and returns an HTTP status code of 500. The application now doubles the backoff interval to four seconds and retries the call.
- Finally, the third call succeeds.
- In this scenario, the retry operation would have attempted up to four retries while doubling the backoff duration before failing the call.

It's important to increase the backoff period before retrying the call to allow the service time to self-correct. It's a best practice to implement an exponentially increasing backoff (doubling the period on each retry) to allow adequate correction time.

Circuit breaker pattern

While the retry pattern can help salvage a request entangled in a partial failure, there are situations where failures can be caused by unanticipated events that will require longer periods of time to resolve. These faults can range in severity from a partial loss of connectivity to the complete failure of a service. In these situations, it's pointless for an application to continually retry an operation that is unlikely to succeed.

To make things worse, executing continual retry operations on a non-responsive service can move you into a self-imposed denial of service scenario where you flood your service with continual calls exhausting resources such as memory, threads and database connections, causing failure in unrelated parts of the system that use the same resources.

In these situations, it would be preferable for the operation to fail immediately and only attempt to invoke the service if it's likely to succeed.

The Circuit Breaker pattern can prevent an application from repeatedly trying to execute an operation that's likely to fail. It also monitors the application with a periodic trial call to determine whether the fault has resolved. Figure 6-5 shows the Circuit Breaker pattern in action.

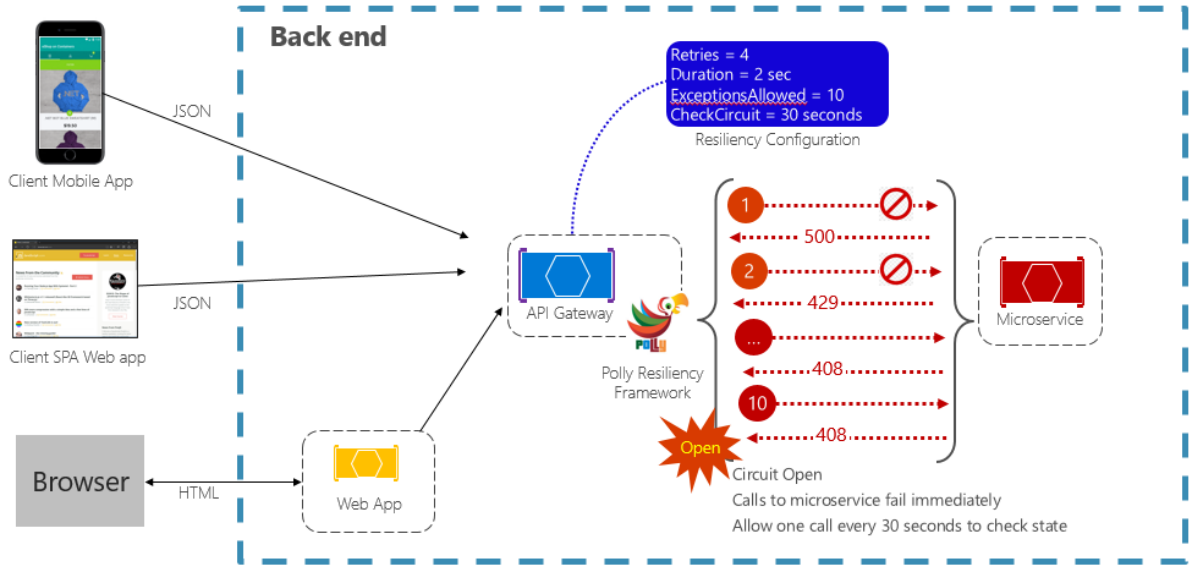


Figure 6-5. Circuit breaker pattern in action

In the previous figure, a Circuit Breaker pattern has been added to the original retry pattern. Note how after 10 failed requests, the circuit breakers opens and no longer allows calls to the service. The CheckCircuit value, set at 30 seconds, specifies how often the library allows one request to proceed to the service. If that call succeeds, the circuit closes and the service is once again available to traffic.

Keep in mind that the intent of the Circuit Breaker pattern is *different* than that of the Retry pattern. The Retry pattern enables an application to retry an operation in the expectation that it will succeed. The Circuit Breaker pattern prevents an application from doing an operation that is likely to fail. Often, an application will *combine* these two patterns by using the Retry pattern to invoke an operation through a circuit breaker. However, the retry logic should be sensitive to any exceptions returned by the circuit breaker and abandon retry attempts if the circuit breaker indicates that a fault isn't transient.

Application resiliency is a must for handling problematic requested operations. But, it's only half of the story. Next, we cover resiliency features available in the Azure cloud.

Azure platform resiliency

Building a reliable application in the cloud is different from traditional on-premises application development. While historically you purchased higher-end hardware to scale up, in a cloud environment you scale out. Instead of trying to prevent failures, the goal is to minimize their effects and keep the system stable.

That said, reliable cloud applications display distinct characteristics:

- They're resilient, recover gracefully from problems, and continue to function.
- They're highly available (HA) and run as designed in a healthy state with no significant downtime.

Understanding how these characteristics work together - and how they affect cost - is essential to building a reliable cloud-native application. We'll next look at ways that you can build resiliency and availability into your cloud-native applications leveraging features from the Azure cloud.

Design with redundancy

Failures vary in scope of impact. A hardware failure, such as a failed disk, can affect a single node in a cluster. A failed network switch could affect an entire server rack. Less common failures, such as loss of power, could disrupt a whole datacenter. Rarely, an entire region becomes unavailable.

Redundancy is one way to provide application resilience. The exact level of redundancy needed depends on your business requirements and will affect both cost and complexity of your system. For example, a multi-region deployment is more expensive and more complex to manage than a single-region deployment. You'll need operational procedures to manage failover and failback. The additional cost and complexity might be justified for some business scenarios and not others.

To architect redundancy, you need to identify the critical paths in your application, and then determine if there's redundancy at each point in the path? If a subsystem should fail, will the application fail over to something else? Finally, you need a clear understanding of those features built into the Azure cloud platform that you can leverage to meet your redundancy requirements. Here are recommendations for architecting redundancy:

- *Deploy multiple instances of services.* If your application depends on a single instance of a service, it creates a single point of failure. Provisioning multiple instances improves both resiliency and scalability. When hosting in Azure Kubernetes Service, you can declaratively configure redundant instances (replica sets) in the Kubernetes manifest file. The replica count value can be managed programmatically, in the portal or through autoscaling features, which will be discussed later on.
- *Leveraging a load balancer.* Load-balancing distributes your application's requests to healthy service instances and automatically removes unhealthy instances from rotation. When deploying to Kubernetes, load balancing can be specified in the Kubernetes manifest file in the Services section.
- *Plan for multiregion deployment.* If your application is deployed to a single region, and the region becomes unavailable, your application will also become unavailable. This may be unacceptable under the terms of your application's service level agreements. Instead, consider deploying your application and its services across multiple regions. For example, an Azure Kubernetes Service (AKS) cluster is deployed to a single region. To protect your system from a regional failure, you might deploy your application to multiple AKS clusters across different regions and use the Paired Regions feature to coordinate platform updates and prioritize recovery efforts.

- *Enable geo-replication*. Geo-replication for services such as Azure SQL Database and Cosmos DB will create secondary replicas of your data across multiple regions. While both services will automatically replicate data within the same region, geo-replication protects you against a regional outage by enabling you to fail over to a secondary region. Another best practice for geo-replication centers around storing container images. To deploy a service in AKS, you need to store and pull the image from a repository. Azure Container Registry integrates with AKS and can securely store container images. To improve performance and availability, consider geo-replicating your images to a registry in each region where you have an AKS cluster. Each AKS cluster then pulls container images from the local container registry in its region as shown in Figure 6-6:

Figure 6-6. Replicated resources across regions

- *Implement a DNS traffic load balancer*. Azure Traffic Manager provides high-availability for critical applications by load-balancing at the DNS level. It can route traffic to different regions based on geography, cluster response time, and even application endpoint health. For example, Azure Traffic Manager can direct customers to the closest AKS cluster and application instance. If you have multiple AKS clusters in different regions, use Traffic Manager to control how traffic flows to the applications that run in each cluster. Figure 6-7 shows this scenario.

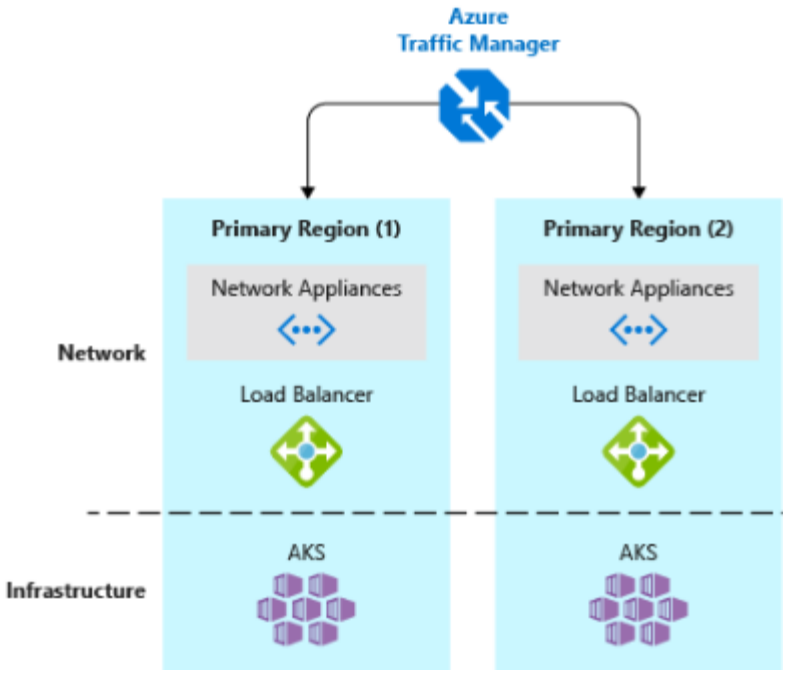


Figure 6-7. AKS and Azure Traffic Manager

Design for scalability

The cloud thrives on scaling. The ability to increase/decrease system resources to address increasing/decreasing system load is a key tenet of the Azure cloud. But, to effectively scale an application, you need an understanding of the scaling features of each Azure service that you include in your application. Here are recommendations for effectively implementing scaling in your system.

- *Design for scaling.* An application must be designed for scaling. To start, services should be stateless so that requests can be routed to any instance. Having stateless services also means that adding or removing an instance doesn't adversely impact current users.
- *Partition workloads*. Decomposing domains into independent, self-contained microservices enable each service to scale independently of others. Typically, services will have different scalability needs and requirements. Partitioning enables you to scale only what needs to be scaled without the unnecessary cost of scaling an entire application.
- *Favor scale-out.* Cloud-based applications favor scaling out resources as opposed to scaling up. Scaling out (also known as horizontal scaling) involves adding more service resources to an existing system to meet and share a desired level of performance. Scaling up (also known as vertical scaling) involves replacing existing resources with more powerful hardware (more disk, memory, and processing cores). Scaling out can be invoked automatically with the autoscaling features available in some Azure cloud resources. Scaling out across multiple resources also adds redundancy to the overall system. Finally scaling up a single resource is typically more expensive than scaling out across many smaller resources. Figure 6-8 shows the two approaches:

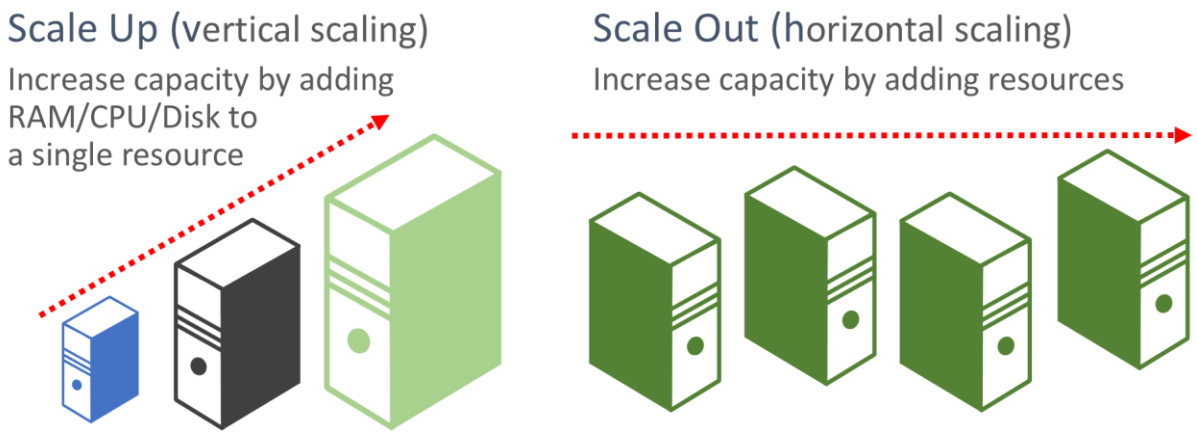


Figure 6-8. Scale up versus scale out

- *Scale proportionally*. When scaling a service, think in terms of *resource sets*. If you were to dramatically scale out a specific service, what impact would that have on back-end data stores, caches and dependent services? Some resources such as Cosmos DB can scale out proportionally, while many others can't. You want to ensure that you don't scale out a resource to a point where it will exhaust other associated resources.
- *Avoid affinity*. A best practice is to ensure a node doesn't require local affinity, often referred to as a *sticky session*. A request should be able to route to any instance. If you need to persist state, it should be saved to a distributed cache, such as Azure Redis cache.
- *Take advantage of platform autoscaling features*. Use built-in autoscaling features whenever possible, rather than custom or third-party mechanisms. Where possible, use scheduled scaling rules to ensure that resources are available without a startup delay, but add reactive autoscaling to the rules as appropriate, to cope with unexpected changes in demand. For more information, see Autoscaling guidance.
- *Scale-up aggressively*. A final practice would be to scale up aggressively so that you can quickly meet immediate spikes in traffic without losing business. And, then scale down (that is, remove unneeded resources) conservatively to keep the system stable. A simple way to implement this is to set the cool down period, which is the time to wait between scaling operations, to five minutes for adding resources and up to 15 minutes for removing instances.

Built-in retry in services

We encouraged the best practice of implementing programmatic retry operations in an earlier section. Keep in mind that many Azure services and their corresponding client SDKs also include retry mechanisms. The following list summarizes retry features in the many of the Azure services that are discussed in this book:

- *Azure Cosmos DB*. The DocumentClient class from the client API automatically retires failed attempts. The number of retries and maximum wait time are configurable. Exceptions thrown by the client API are either requests that exceed the retry policy or non-transient errors.

- *Azure Redis Cache*. The Redis StackExchange client uses a connection manager class that includes retries on failed attempts. The number of retries, specific retry policy and wait time are all configurable.
- *Azure Service Bus*. The Service Bus client exposes a RetryPolicy class that can be configured with a back-off interval, retry count, and TerminationTimeBuffer, which specifies the maximum time an operation can take. The default policy is nine maximum retry attempts with a 30-second backoff period between attempts.
- *Azure SQL Database*. Retry support is provided when using the Entity Framework Core library.
- *Azure Storage*. The storage client library support retry operations. The strategies vary across Azure storage tables, blobs, and queues. As well, alternate retries switch between primary and secondary storage services locations when the geo-redundancy feature is enabled.
- *Azure Event Hubs*. The Event Hub client library features a RetryPolicy property, which includes a configurable exponential backoff feature.

Resilient communications

Throughout this book, we've evangelized the merits of moving beyond traditional monolithic application design and embracing a microservice-based architecture where a set of distributed, self-contained services run independently and communicate with each other using standard communication protocols such as HTTP and HTTPS. While such an architecture buys you many important benefits, it also presents many challenges. Consider, for example, the following concerns:

- *Out-of-process network communication*. Each service communicates over a network protocol that introduces network congestion, latency, and transient faults.
- *Service discovery*. With each service running across a cluster of machines with its own IP address and port, how do services discover and communicate with each other?
- *Resiliency*. How do you manage short-lived failures and keep the system stable?
- *Load balancing*. How does inbound traffic get distributed across multiple instances of a service?
- *Security*. How are security concerns such as transport-level encryption and certificate management enforced?
- *Distributed Monitoring. - How do you correlate and capture traceability and monitoring for a single request across multiple consuming services?

While these concerns can be addressed with various libraries and frameworks, implementing them inside your codebase can be expensive, complex, and time-consuming. Moreover, you end up with a solution where infrastructure concerns are coupled to business logic.

Service mesh

A better approach is to consider a new and rapidly evolving technology entitled *Service Mesh*. A service mesh is a configurable infrastructure layer with built-in capabilities to handle service communication and many of the challenges mentioned above. It decouples these concerns from your business code and moves them into a service proxy, an instance of which accompanies each of your

services. Often referred to as the Sidecar pattern, the service mesh proxy is deployed into a separate process to provide isolation and encapsulation from your business code. However, the proxy is closely linked to the service being created along with it and sharing its lifecycle. Figure 6-9 shows this scenario.

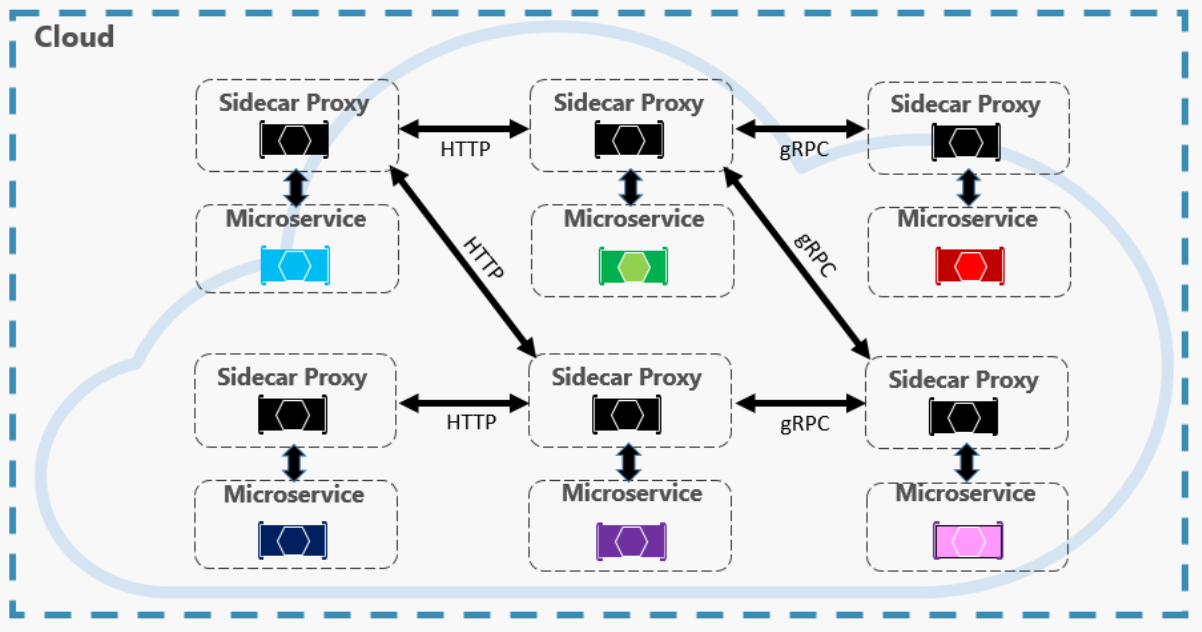


Figure 6-9. Service mesh with a side car

In the previous figure, note how the proxy intercepts and manages communication among the microservices and the cluster.

A service mesh is logically split into two disparate components: A data plane and control plane. Figure 6-10 shows these components and their responsibilities.

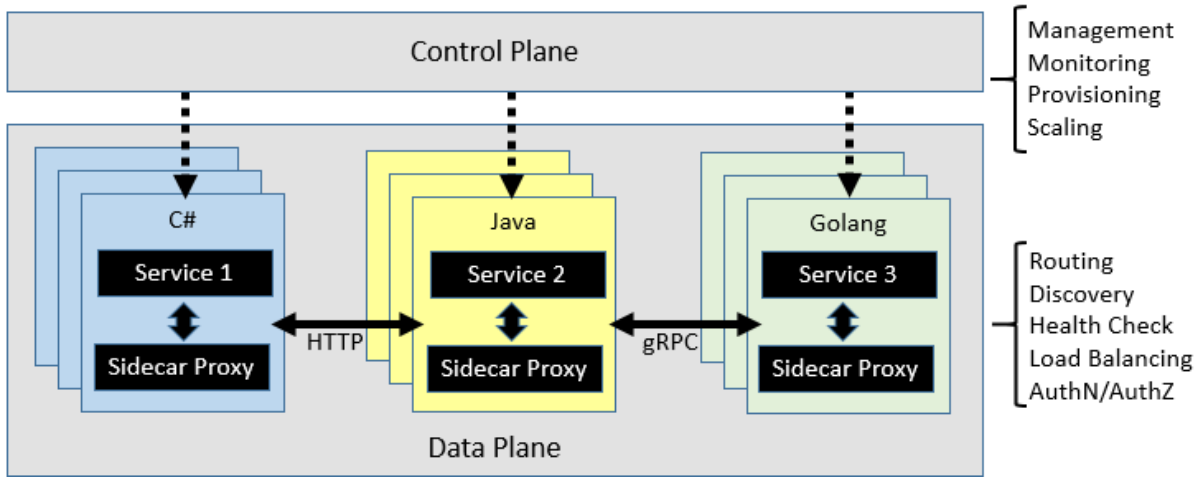


Figure 6-10. Service mesh control and data plane

Once configured, a service mesh is highly functional. It can retrieve a corresponding pool of instances from a service discovery endpoint. It can then send a request to a specific instance, recording the

latency and response type of the result. A mesh can choose the instance most likely to return a fast response based on many factors, including its observed latency for recent requests.

If an instance is unresponsive or fails, the mesh can retry the request on another instance. If a pool consistently returns errors, a mesh can evict it from the load-balancing pool to be retried periodically later after it heals. If a request times out, a mesh can fail and then retry the request. A mesh captures behavior in the form of metrics and distributed tracing, which then can be emitted to a centralized metrics system.

Istio and Envoy

While a few service mesh options currently exist, Istio is the most popular as of the time of this writing. A joint venture from IBM, Google, and Lyft, it's an open-source offering that can be integrated into a new or existing distributed application. It provides a consistent and complete solution to secure, connect, and monitor microservices. Its features include:

- Secure service-to-service communication in a cluster with strong identity-based authentication and authorization.
- Automatic load balancing for HTTP, gRPC, WebSocket, and TCP traffic.
- Fine-grained control of traffic behavior with rich routing rules, retries, failovers, and fault injection.
- A pluggable policy layer and configuration API supporting access controls, rate limits, and quotas.
- Automatic metrics, logs, and traces for all traffic within a cluster, including cluster ingress and egress.

A key component for an Istio implementation is a proxy service entitled the Envoy proxy. Originating from Lyft and subsequently contributed to the Cloud Native Computing Foundation (discussed in chapter 1), the Envoy proxy runs alongside each service and provides a platform-agnostic foundation for the following features:

- Dynamic service discovery.
- Load balancing.
- TLS termination.
- HTTP and gRPC proxies.
- Circuit breaker resiliency.
- Health checks.
- Rolling updates with canary deployments.

As previously discussed, Envoy is deployed as a sidecar to each microservice in the cluster.

Integration with Azure Kubernetes Services

The Azure cloud embraces Istio and provides direct support for it within Azure Kubernetes Services. The following links can help you get started:

- Installing Istio in AKS
- Using AKS and Istio

Monitoring and health

Microservices and cloud-native applications go hand in hand with good DevOps practices. DevOps is many things to many people but perhaps one of the better definitions comes from cloud advocate and DevOps evangelist Donovan Brown:

"DevOps is the union of people, process, and products to enable continuous delivery of value to our end users."

Unfortunately, with terse definitions, there's always room to say more things. One of the key components of DevOps is ensuring that the applications running in production are functioning properly and efficiently. To gauge the health of the application in production, it's necessary to monitor the various logs and metrics being produced from the servers, hosts, and the application proper. The number of different services running in support of a cloud-native application makes monitoring the health of individual components and the application as a whole a critical challenge.

Observability patterns

Just as patterns have been developed to aid in the layout of code in applications, there are patterns for operating applications in a reliable way. Three useful patterns in maintaining applications have emerged: logging, monitoring, and alerts.

When to use logging

No matter how careful we are, applications almost always behave in unexpected ways in production. When users report problems with an application, it's extremely useful to be able to see what was going on with the app when the problem occurred. One of the most tried and true ways of capturing information about what an application is doing while it's running is to have the application write down what it's doing. This process is known as logging. Anytime failures or problems occur in production, the goal should be to reproduce the conditions under which the failures occurred, in a non-production environment. Having good logging in place provides a roadmap for developers to follow in order to duplicate problems in an environment that can be tested and experimented with.

Logging in cloud-native applications

Every programming language has tooling that permits writing logs, and typically the overhead for writing these logs is low. Many of the logging libraries provide logging different kinds of criticalities, which can be tuned at run time. For instance, the Serilog library is a popular structured logging library for .NET that provides the following logging levels

- Verbose
- Debug
- Information
- Warning
- Error
- Fatal

These different log levels provide granularity in logging. When the application is functioning properly in production, it may be configured to only log important messages. When the application is misbehaving, then the log level can be increased so more verbose logs are gathered. This balances performance against ease of debugging.

The high performance of logging tools and the tunability of verbosity should encourage developers to log frequently. Many favor a pattern of logging the entry and exit of each method. This approach may sound like overkill, but it's infrequent that developers will wish for less logging. In fact, it's not uncommon to perform deployments for the sole purpose of adding logging around a problematic method. Err on the side of too much logging and not on too little. Note that some tools can be used to automatically provide this kind of logging.

In traditional applications, log files were typically stored on the local machine. In fact, on Unix-like operating systems, there's a folder structure defined to hold any logs, typically under `/var/log`. The usefulness of logging to a flat file on a single machine is vastly reduced in a cloud environment. Applications producing logs may not have access to the local disk or the local disk may be highly transient as containers are shuffled around physical machines.

Cloud-native applications developed using a microservices architecture also pose some challenges for file-based loggers. User requests may now span multiple services that are run on different machines, and may include serverless functions with no access to a local file system at all. It would be very challenging to correlate the logs from a user or a session across these many services and machines.

Finally, the number of users in some cloud-native applications is high. Imagine that each user generates a hundred lines of log messages when they log into an application. In isolation, that is manageable, but multiply that over 100,000 users and the volume of logs becomes large.

Fortunately, there are some fantastic alternatives to using file system-based logging. A centralized log server to which all logs are sent, fixes all these problems. Logs are collected by the applications and shipped to a central logging application which indexes and stores the logs. This class of system can ingest tens of gigabytes of logs every day.

It's also helpful to follow some standard practices when building logging that spans many services. For instance, generating a correlation ID at the start of a lengthy interaction, and then logging it in each message that is related to that interaction, makes it easier to search for all related messages. One

need only find a single message and extract the correlation ID to find all the related messages. Another example is ensuring that the log format is the same for every service, whatever the language or logging library it uses. This standardization makes reading logs much easier. Figure 7-1 demonstrates how a microservices architecture can leverage centralized logging as part of its workflow.

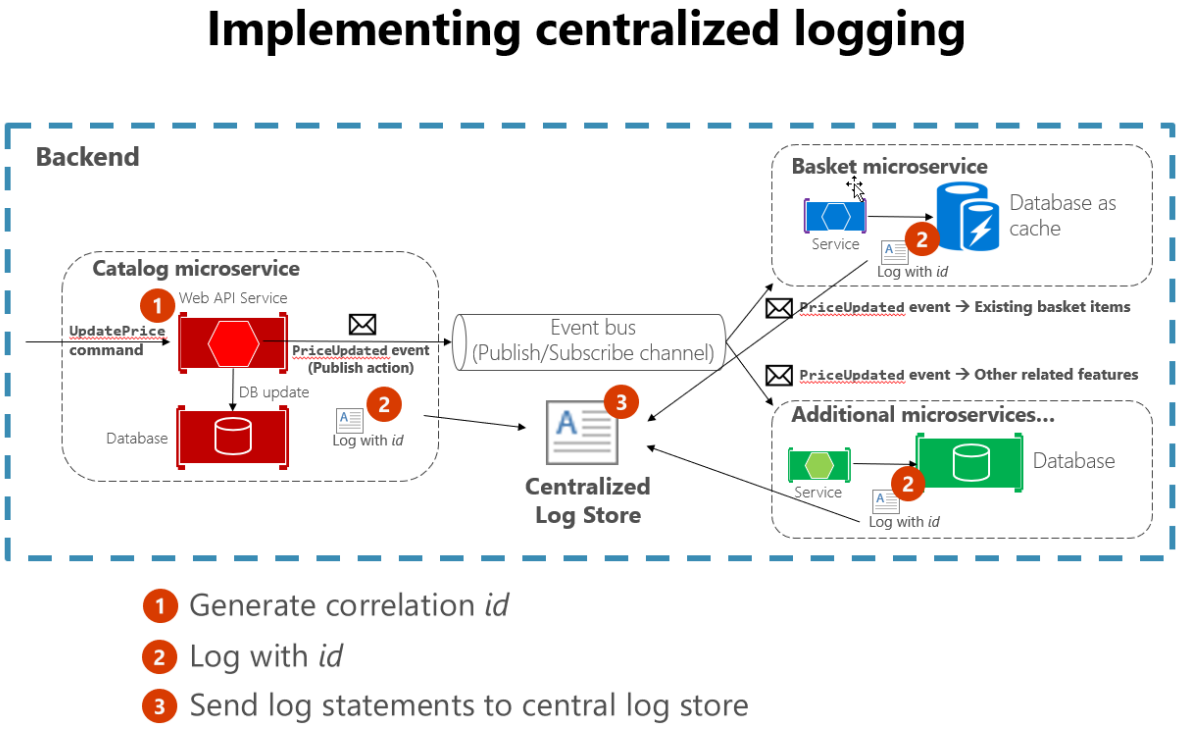


Figure 7-1. Logs from various sources are ingested into a centralized log store.

When to use monitoring

Some applications aren't mission-critical. Maybe they're only used internally, and when a problem occurs, the user can contact the team responsible and the application can be restarted. However, customers often have higher expectations for the applications they consume. If you need to know when problems occur with your application *before* users do, or before users notify you, you need to monitor its current state. Implemented properly, monitoring can let you know about conditions that will lead to problems, letting you address underlying conditions before they result in any user impact.

Monitoring considerations

Some centralized logging systems take on an additional role of collecting telemetry outside of pure logs. They can collect metrics, such as time to run a database query, average response time from a web server, and even CPU load averages and memory pressure as reported by the operating system. In conjunction with the logs, these systems can provide a holistic view of the health of nodes in the system and the application as a whole.

The metric gathering capabilities of the monitoring tools can also be fed manually from within the application. Business flows that are of particular interest such as new users signing up or orders being

placed, may be instrumented such that they increment a counter in the central monitoring system. This unlocks the monitoring tools to not only monitor the health of the application but the health of the business.

Queries can be constructed in the log aggregation tools to look for certain statistics or patterns, which can then be displayed in graphical form, on custom dashboards. Frequently, teams will invest in large, wall-mounted displays that rotate through the statistics related to an application. This way, it's simple to see the problems as they occur.

When to use alerts

If you need to react to problems with your application, you need some way to alert the right personnel. This is the third cloud-native application observability pattern, and depends on logging and monitoring. Your application needs to have logging in place to allow problems to be diagnosed, and in some cases to feed into monitoring tools. It needs monitoring to aggregate application metrics and health data in one place. Once this has been established, rules can be created that will trigger alerts when certain metrics fall outside of acceptable levels.

Alerts

You can craft queries against the monitoring tools to look for known failure conditions. For instance, queries could search through the incoming logs for indications of HTTP status code 500, which indicates a problem on a web server. As soon as one of these is detected, then an e-mail or an SMS could be sent to the owner of the originating service who can begin to investigate.

Typically though, a single 500 error isn't sufficient to determine that a problem has occurred. It could mean that a user mistyped their password or entered some malformed data. The alert queries can be crafted to only fire when a larger than average number of 500 errors are detected.

One of the most damaging patterns in alerting is to fire too many alerts for humans to investigate. Service owners will rapidly become desensitized to errors that they've previously investigated and found to be benign. When true errors occur then they'll be lost in the noise of hundreds of false positives. The parable of the Boy Who Cried Wolf is frequently told to children to warn them of this very danger. It's important to ensure that the alerts that do fire are indicative of a real problem.

Logging with Elastic Stack

There are many good centralized logging tools and they vary in cost from being free, open-source tools, to more expensive options. In many cases, the free tools are as good as or better than the paid offerings. One such tool is a combination of three open-source components: Elastic search, Logstash, and Kibana. Collectively these tools are known as the Elastic Stack or ELK stack.

What are the advantages of Elastic Stack?

Elastic Stack provides centralized logging in a low-cost, scalable, cloud-friendly manner. Its user interface streamlines data analysis so you can spend your time gleaning insights from your data instead of fighting with a clunky interface. It supports a wide variety of inputs so as your distributed

application spans more and different kinds of services, you can expect to continue to be able to feed log and metric data into the system. The Elastic Stack also supports fast searches even across large data sets, making it possible to for even large applications to log detailed data and still be able to have visibility into it in a performant fashion.

Logstash

The first component is Logstash. This tool is used to gather log information from a large variety of different sources. For instance, Logstash can read logs from disk and also receive messages from logging libraries like Serilog. Logstash can do some basic filtering and expansion on the logs as they arrive. For instance, if your logs contain IP addresses then Logstash may be configured to do a geographical lookup and obtain a country or even city of origin for that message.

Serilog is a logging library for .NET languages, which allows for parameterized logging. Instead of generating a textual log message that embeds fields, parameters are kept separate. This allows for more intelligent filtering and searching. A sample Serilog configuration for writing to Logstash appears in Figure 7-2.

```
var log = new LoggerConfiguration()
         .WriteTo.Http("http://localhost:8080")
         .CreateLogger();
```

Figure 7-2 Serilog config for writing log information directly to logstash over HTTP

Logstash would use a configuration like the one shown in Figure 7-3.

```
input {
    http {
        #default host 0.0.0.0:8080
        codec => json
    }
}

output {
    elasticsearch {
        hosts => "elasticsearch:9200"
        index=>"sales-%{+xxxx.ww}"
    }
}
```

Figure 7-3 - A Logstash configuration for consuming logs from Serilog

For scenarios where extensive log manipulation isn't needed there's an alternative to Logstash known as Beats. Beats is a family of tools that can gather a wide variety of data from logs to network data and uptime information. Many applications will use both Logstash and Beats.

Once the logs have been gathered by Logstash, it needs somewhere to put them. While Logstash supports many different outputs, one of the more exciting ones is Elastic search.

Elastic search

Elastic search is a powerful search engine that can index logs as they arrive. It makes running queries against the logs quick. Elastic search can handle huge quantities of logs and, in extreme cases, can be scaled out across many nodes.

Log messages that have been crafted to contain parameters or that have had parameters split from them through Logstash processing, can be queried directly as Elasticsearch preserves this information.

A query that searches for the top 10 pages visited by `jill@example.com`, appears in Figure 7-4.

```
"query": {
    "match": {
      "user": "jill@example.com"
    }
  },
  "aggregations": {
    "top_10_pages": {
      "terms": {
        "field": "page",
        "size": 10
      }
    }
  }
```

Figure 7-4 - An Elasticsearch query for finding top 10 pages visited by a user

Visualizing information with Kibana web dashboards

The final component of the stack is Kibana. This tool is used to provide interactive visualizations in a web dashboard. Dashboards may be crafted even by users who are non-technical. Most data that is resident in the Elasticsearch index, can be included in the Kibana dashboards. Individual users may have different dashboard desires and Kibana enables this customization through allowing user-specific dashboards.

Installing Elastic Stack on Azure

The Elastic stack can be installed on Azure in a number of ways. As always, it's possible to provision virtual machines and install Elastic Stack on them directly. This option is preferred by some experienced users as it offers the highest degree of customizability. Deploying on infrastructure as a service introduces significant management overhead forcing those who take that path to take ownership of all the tasks associated with infrastructure as a service such as securing the machines and keeping up-to-date with patches.

An option with less overhead is to make use of one of the many Docker containers on which the Elastic Stack has already been configured. These containers can be dropped into an existing Kubernetes cluster and run alongside application code. The sebp/elk container is a well-documented and tested Elastic Stack container.

Another option is a recently announced ELK-as-a-service offering.

References

- Install Elastic Stack on Azure

Monitoring in Azure Kubernetes Services

The built-in logging in Kubernetes is primitive. However, there are some great options for getting the logs out of Kubernetes and into a place where they can be properly analyzed. If you need to monitor your AKS clusters, configuring Elastic Stack for Kubernetes is a great solution.

Elastic Stack

The Elastic Stack is a powerful option for gathering information from a Kubernetes cluster. Kubernetes supports sending logs to an Elasticsearch endpoint, and for the most part, all you need to get started is to set the environment variables as shown in Figure 7-5:

```
KUBE_LOGGING_DESTINATION=elasticsearch
KUBE_ENABLE_NODE_LOGGING=true
```

Figure 7-5 - Configuration variables for Kubernetes

This will install Elasticsearch on the cluster and target sending all the cluster logs to it.

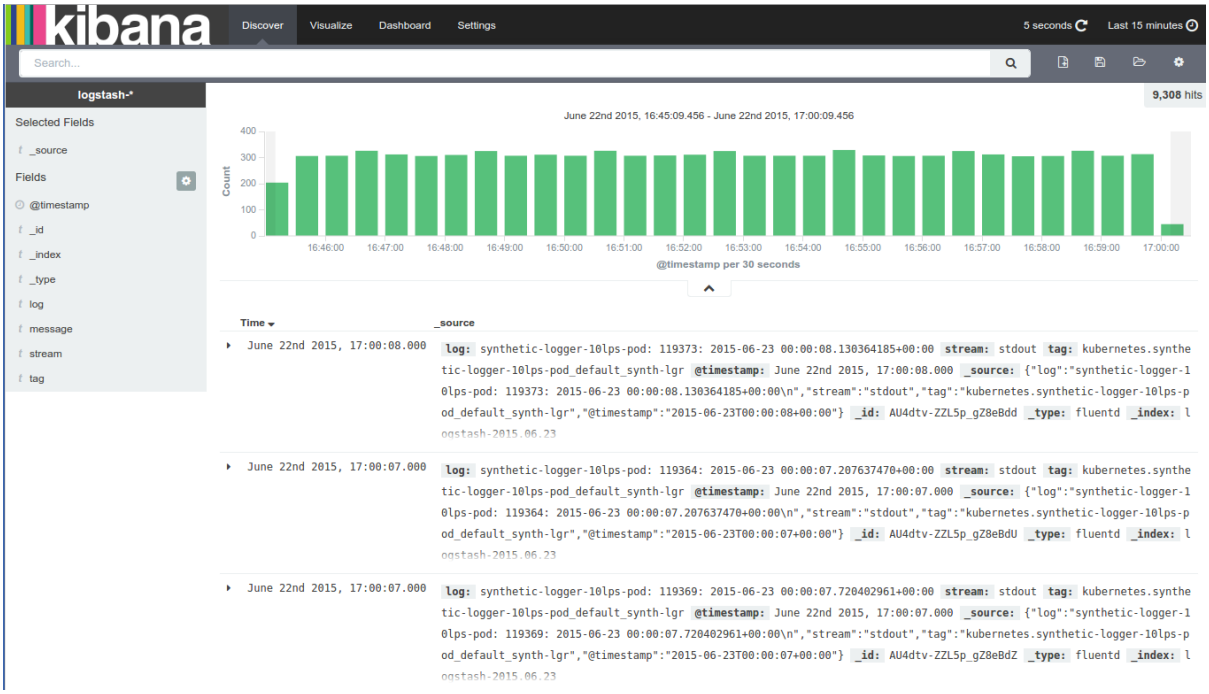


Figure 7-6. An example of a Kibana dashboard showing the results of a query against logs that are ingested from Kubernetes

Azure Container Monitoring

Azure Container Monitoring supports consuming logs from not just Kubernetes but also from other orchestration engines such as DC/OS, Docker Swarm, and Red Hat OpenShift.

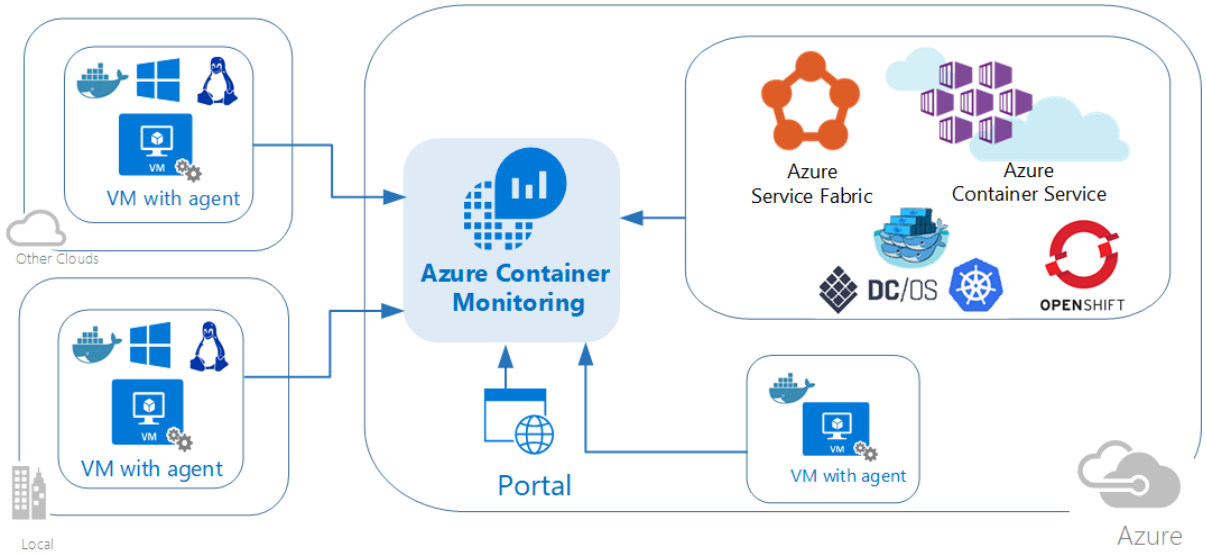


Figure 7-7. Consuming logs from various containers

Log and metric information is gathered not just from the containers running in the cluster but also from the cluster hosts themselves. It allows correlating log information from the two making it much easier to track down an error.

Installing the log collectors differs on Windows and Linux clusters. But in both cases the log collection is implemented as a Kubernetes DaemonSet, meaning that the log collector is run as a container on each of the nodes.

No matter which orchestrator or operating system is running the Azure Monitor daemon, the log information is forwarded to the same Azure Monitor tools with which users are familiar. This ensures a parallel experience in environments that mix different log sources such as a hybrid Kubernetes/Azure Functions environment.

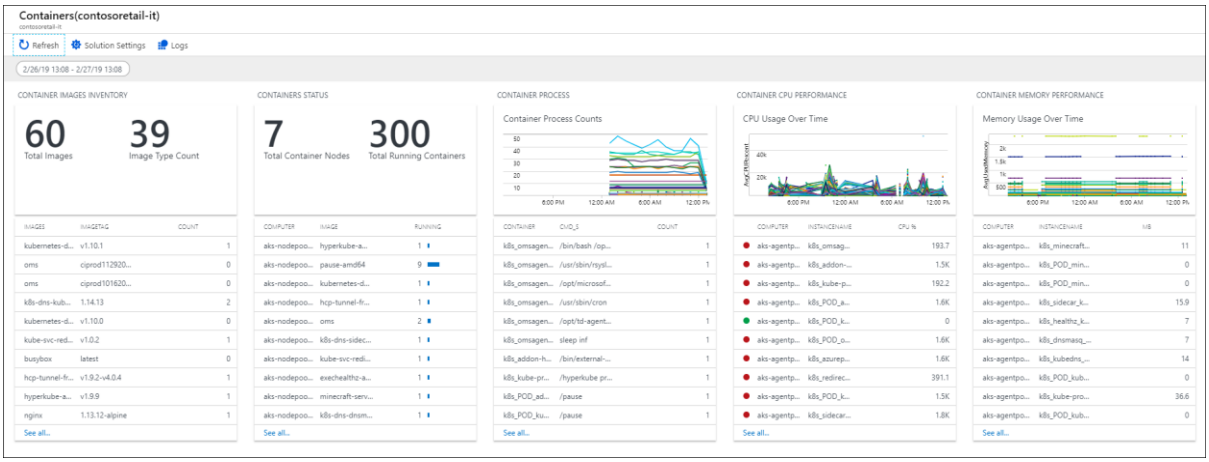


Figure 7-8. A sample dashboard showing logging and metric information from a number of running containers.

Log.Finalize()

Logging is one of the most overlooked and yet most important parts of deploying any application at scale. As the size and complexity of applications increase, then so does the difficulty of debugging them. Having top quality logs available makes debugging much easier and moves it from the realm of "nearly impossible" to "a pleasant experience".

Azure Monitor

No other cloud provider has as mature of a cloud application monitoring solution as that found in Azure. Azure Monitor is an umbrella name for a collection of tools designed to provide visibility into the state of your system, insights into any problems and optimization of your application.

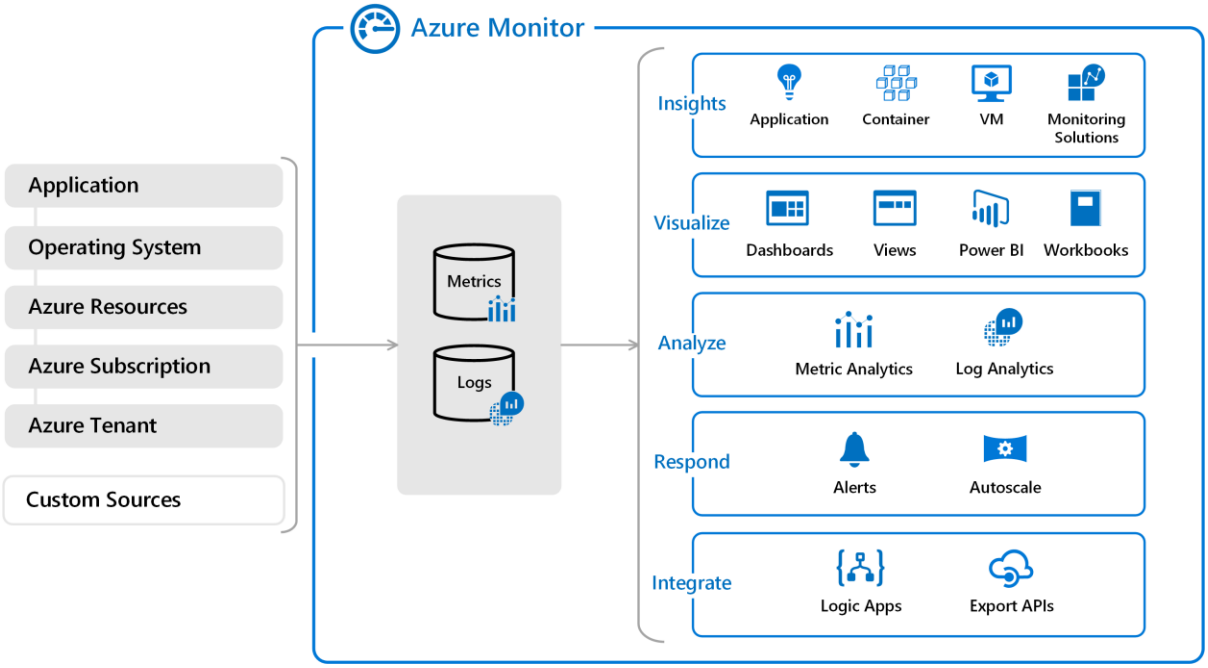


Figure 7-9. Azure Monitor, a collection to tools to provide insight into how a cloud-native application is functioning.

Gathering logs and metrics

The first step in any monitoring solution is to gather as much data as possible. The more data that can be gathered, the deeper the insights that can be obtained. Instrumenting systems has traditionally been difficult. Simple Network Management Protocol (SNMP) was the gold standard protocol for collecting machine level information but it required a great deal of knowledge and configuring. Fortunately, much of this hard work has been eliminated as the most common metrics are gathered automatically by Azure Monitor.

Application level metrics and events aren't possible to instrument automatically because they're local to the application being deployed. In order to gather these metrics, there are SDKs and APIs available to directly report such information, such as when a customer signs up or completes an order. Exceptions can also be captured and reported back into Azure Monitor via Application Insights. The

SDKs support most every language found in Cloud Native Applications including Go, Python, JavaScript, and the .NET languages.

The ultimate goal of gathering information about the state of your application is to ensure that your end users have a good experience. What better way to tell if users are experiencing issues than doing outside-in web tests? These tests can be as simple as pinging your website from locations around the world or as involved as having agents log into the site and do actions.

Reporting data

Once the data is gathered, it can be manipulated, summarized, and plotted into charts, which allow users to instantly see when there are problems. These charts can be gathered into dashboards or into Workbooks, a multi-page report designed to tell a story about some aspect of the system.

No modern application would be complete without some artificial intelligence or machine learning. To this end, data can be passed to the various machine learning tools in Azure to allow you to extract trends and information that would otherwise be hidden.

Application Insights provides a powerful query language called Kusto that can be used to find records, summarize them, and even plot charts. For instance, this query will locate all the records for the month of November 2007, group them by state, and plot the top 10 as a pie chart.

```
StormEvents
| where StartTime >= datetime(2007-11-01) and StartTime < datetime(2007-12-01)
| summarize count() by State
| top 10 by count_
| render piechart
```

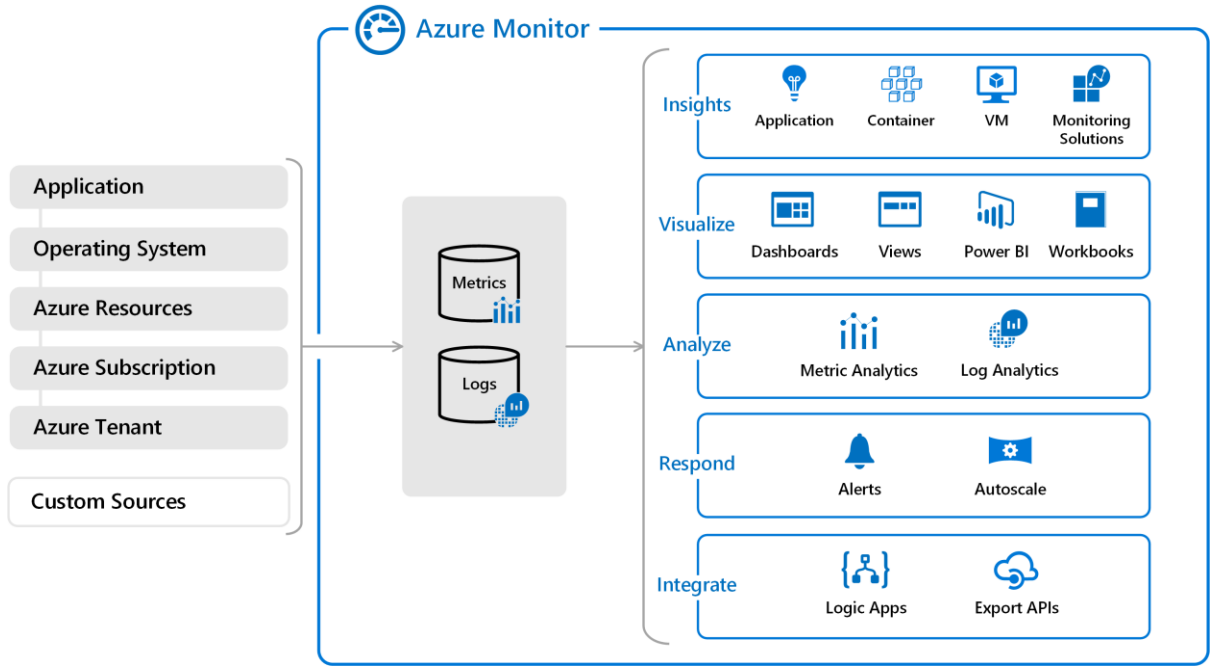


Figure 7-10. The result of the Application Insights Query.

There is a playground for experimenting with Kusto queries, which is a fantastic place to spend an hour or two. Reading sample queries can also be instructive.

Dashboards

There are several different dashboard technologies that may be used to surface the information from Azure Monitor. Perhaps the simplest is to just run queries in Application Insights and plot the data into a chart.

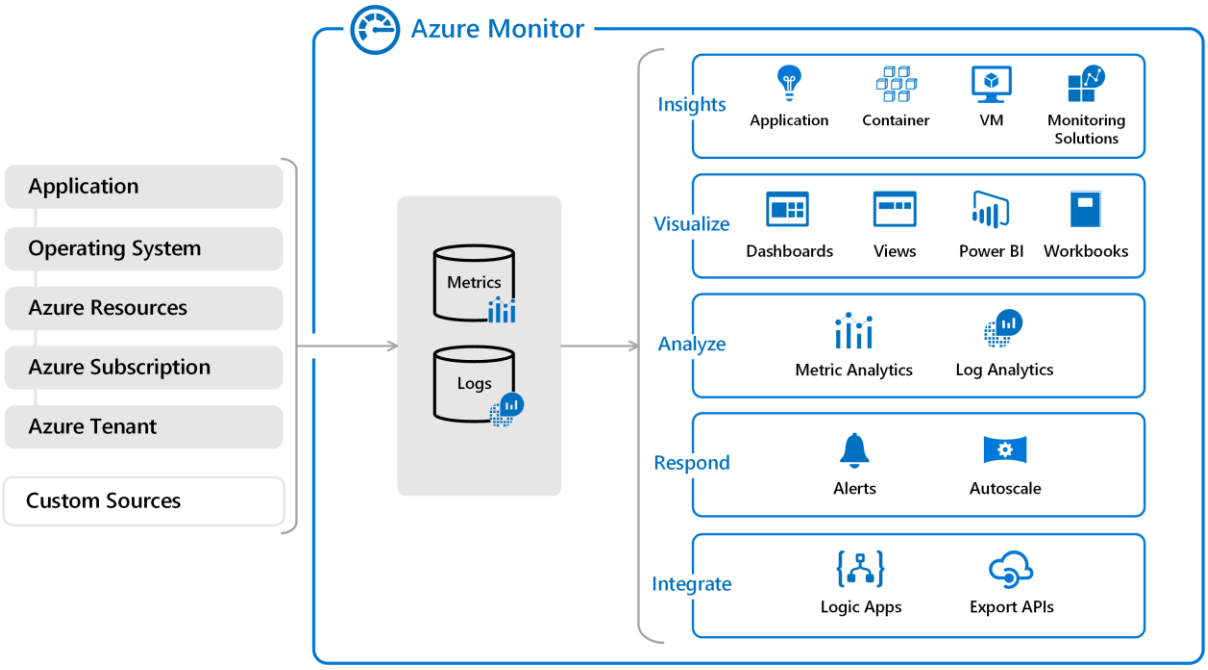


Figure 7-11. An example of Application Insights charts embedded in the main Azure Dashboard.

These charts can then be embedded in the Azure portal proper through use of the dashboard feature. For users with more exacting requirements such as being able to drill down into several tiers of data Azure Monitor data is available to Power BI. Power BI is an industry-leading, enterprise class, business intelligence tool that can aggregate data from many different data sources.

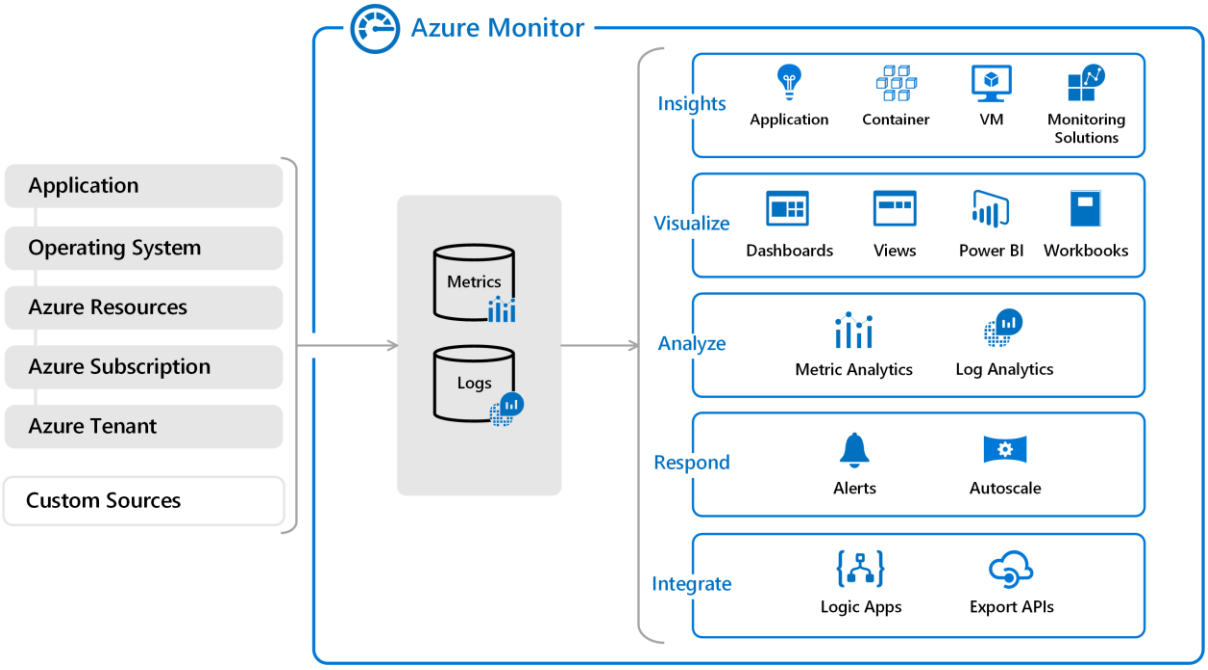


Figure 7-12. An example Power BI dashboard.

Alerts

Sometimes, having data dashboards is insufficient. If nobody is awake to watch the dashboards, then it can still be many hours before a problem is addressed, or even detected. To this end, Azure Monitor also provides a top notch alerting solution. Alerts can be triggered by a wide range of conditions including:

- Metric values
- Log search queries
- Activity Log events
- Health of the underlying Azure platform
- Tests for web site availability

When triggered, the alerts can perform a wide variety of tasks. On the simple side, the alerts may just send an e-mail notification to a mailing list or a text message to an individual. More involved alerts might trigger a workflow in a tool such as PagerDuty, which is aware of who is on call for a particular application. Alerts can trigger actions in Microsoft Flow unlocking near limitless possibilities for workflows.

As common causes of alerts are identified, the alerts can be enhanced with details about the common causes of the alerts and the steps to take to resolve them. Highly mature Cloud Native Application deployments may opt to kick off self-healing tasks, which perform actions such as removing failing nodes from a scale set or triggering an autoscaling activity. Eventually it may no longer be necessary to wake up on-call personnel at 2AM to resolve a live-site issue as the system will be able to adjust itself to compensate or at least limp along until somebody arrives at work the next morning.

Azure Monitor automatically leverages machine learning to understand the normal operating parameters of deployed applications. This enables it to detect services that are operating outside of their normal parameters. For instance, the typical weekday traffic on the site might be 10,000 requests per minute. And then, on a given week, suddenly the number of requests hits a highly unusual 20,000 requests per minute. Smart Detection will notice this deviation from the norm and trigger an alert. At the same time, the trend analysis is smart enough to avoid firing false positives when the traffic load is expected.

Identity

Most software applications need to have some knowledge of the user or process that is calling them. The user or process interacting with an application is known as a security principal, and the process of authenticating and authorizing these principals is known as identity management, or simply *identity*. Simple applications may include all of their identity management within the application, but this approach doesn't scale well with many applications and many kinds of security principals. Windows supports the use of Active Directory to provide centralized authentication and authorization.

While this solution is effective within corporate networks, it isn't designed for use by users or applications that are outside of the AD domain. With the growth of Internet-based applications and the rise of cloud-native apps, security models have evolved.

In today's cloud-native identity model, architecture is assumed to be distributed. Apps can be deployed anywhere and may communicate with other apps anywhere. Clients may communicate with these apps from anywhere, and in fact, clients may consist of any combination of platforms and devices. Cloud-native identity solutions leverage open standards to achieve secure application access from clients. These clients range from human users on PCs or phones, to other apps hosted anywhere online, to set-top boxes and IOT devices running any software platform anywhere in the world.

Modern cloud-native identity solutions typically leverage access tokens that are issued by a secure token service/server (STS) to a security principal once their identity is determined. The access token, typically a JSON Web Token (JWT), includes *claims* about the security principal. These claims will minimally include the user's identity but may also include additional claims that can be used by applications to determine the level of access to grant the principal.

Typically, the STS is only responsible for authenticating the principal. Determining their level of access to resources is left to other parts of the application.

References

- Microsoft identity platform

Authentication and authorization in cloud-native apps

Authentication is the process of determining the identity of a security principal. *Authorization* is the act of granting an authenticated principal permission to perform an action or access a resource. Sometimes authentication is shortened to `AuthN` and authorization is shortened to `AuthZ`. Cloud-

native applications need to rely on open HTTP-based protocols to authenticate security principals since both clients and applications could be running anywhere in the world on any platform or device. The only common factor is HTTP.

Many organizations still rely on local authentication services like Active Directory Federation Services (ADFS). While this approach has traditionally served organizations well for on premises authentication needs, cloud-native applications benefit from systems designed specifically for the cloud. A recent 2019 United Kingdom National Cyber Security Centre (NCSC) advisory states that "organizations using Azure AD as their primary authentication source will actually lower their risk compared to ADFS." Some reasons outlined in this analysis include:

- Access to full set of Microsoft credential protection technologies.
- Most organizations are already relying on Azure AD to some extent.
- Double hashing of NTLM hashes ensures compromise won't allow credentials that work in local Active Directory.

References

- Authentication basics
- Access tokens and claims
- It may be time to ditch your on premises authentication services

Azure Active Directory

Microsoft Azure Active Directory (Azure AD) offers identity and access management as a service. Customers use it to configure and maintain who users are, what information to store about them, who can access that information, who can manage it, and what apps can access it. AAD can authenticate users for applications configured to use it, providing a single sign-on (SSO) experience. It can be used on its own or be integrated with Windows AD running on premises.

Azure AD is built for the cloud. It's truly a cloud-native identity solution that uses a REST-based Graph API and OData syntax for queries, unlike Windows AD, which uses LDAP. On premises Active Directory can sync user attributes to the cloud using Identity Sync Services, allowing all authentication to take place in the cloud using Azure AD. Alternately, authentication can be configured via Connect to pass back to local Active Directory via ADFS to be completed by Windows AD on premises.

Azure AD supports company branded sign-in screens, multi-factory authentication, and cloud-based application proxies that are used to provide SSO for applications hosted on premises. It offers different kinds of security reporting and alert capabilities.

References

- Microsoft identity platform

IdentityServer for cloud-native applications

IdentityServer is an open-source authentication server that implements OpenID Connect (OIDC) and OAuth 2.0 standards for ASP.NET Core. It's designed to provide a common way to authenticate requests to all of your applications, whether they're web, native, mobile, or API endpoints. IdentityServer can be used to implement Single Sign-On (SSO) for multiple applications and application types. It can be used to authenticate actual users via sign-in forms and similar user interfaces as well as service-based authentication that typically involves token issuance, verification, and renewal without any user interface. IdentityServer is designed to be a customizable solution. Each instance is typically customized to suit an individual organization and/or set of applications' needs.

Common web app scenarios

Typically, applications need to support some or all of the following scenarios:

- Human users accessing web applications with a browser.
- Human users accessing back-end Web APIs from browser-based apps.
- Human users on mobile/native clients accessing back-end Web APIs.
- Other applications accessing back-end Web APIs (without an active user or user interface).
- Any application may need to interact with other Web APIs, using its own identity or delegating to the user's identity.

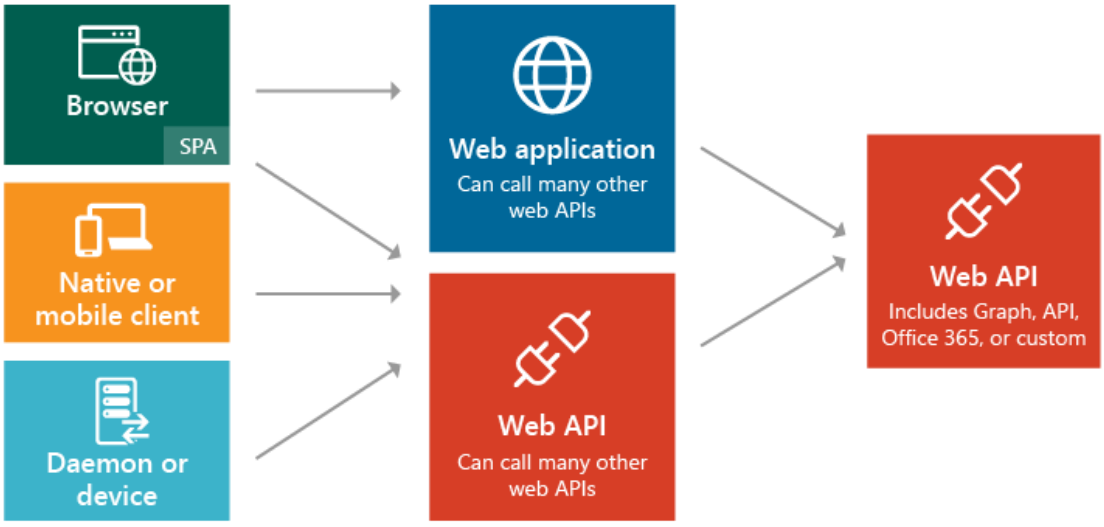


Figure 8-1. Application types and scenarios.

In each of these scenarios, the exposed functionality needs to be secured against unauthorized use. At a minimum, this typically requires authenticating the user or principal making a request for a resource. This authentication may use one of several common protocols such as SAML2p, WS-Fed, or OpenID Connect. Communicating with APIs typically uses the OAuth2 protocol and its support for security tokens. Separating these critical cross-cutting security concerns and their implementation details from the applications themselves ensures consistency and improves security and maintainability.

Outsourcing these concerns to a dedicated product like IdentityServer helps the requirement for every application to solve these problems itself.

IdentityServer provides middleware that runs within an ASP.NET Core application and adds support for OpenID Connect and OAuth2 (see supported specifications). Organizations would create their own ASP.NET Core app using IdentityServer middleware to act as the STS for all of their token-based security protocols. The IdentityServer middleware exposes endpoints to support standard functionality, including:

- Authorize (authenticate the end user)
- Token (request a token programmatically)
- Discovery (metadata about the server)
- User Info (get user information with a valid access token)
- Device Authorization (used to start device flow authorization)
- Introspection (token validation)
- Revocation (token revocation)
- End Session (trigger single sign-out across all apps)

Getting started

IdentityServer4 is open-source and free to use. You can add it to your applications using its NuGet packages. The main package is IdentityServer4 that has been downloaded over four million times. The base package doesn't include any user interface code and only supports in memory configuration. To use it with a database, you'll also want a data provider like IdentityServer4.EntityFramework that uses Entity Framework Core to store configuration and operational data for IdentityServer. For user interface, you can copy files from the Quickstart UI repository into your ASP.NET Core MVC application to add support for sign in and sign out using IdentityServer middleware.

Configuration

IdentityServer supports different kinds of protocols and social authentication providers that can be configured as part of each custom installation. This is typically done in the ASP.NET Core application's `Startup` class in the `ConfigureServices` method. The configuration involves specifying the supported protocols and the paths to the servers and endpoints that will be used. Figure 8-2 shows an example configuration taken from the IdentityServer4 Quickstart UI project:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();

        // some details omitted
        services.AddIdentityServer();

          services.AddAuthentication()
            .AddGoogle("Google", options =>
            {
                options.SignInScheme =
IdentityServerConstants.ExternalCookieAuthenticationScheme;
```

```
                options.ClientId = "<insert here>";
                options.ClientSecret = "<inser here>";
            })
            .AddOpenIdConnect("demoidsrv", "IdentityServer", options =>
            {
                options.SignInScheme =
IdentityServerConstants.ExternalCookieAuthenticationScheme;
                options.SignOutScheme = IdentityServerConstants.SignoutScheme;

                options.Authority = "https://demo.identityserver.io/";
                options.ClientId = "implicit";
                options.ResponseType = "id_token";
                options.SaveTokens = true;
                options.CallbackPath = new PathString("/signin-idsrv");
                options.SignedOutCallbackPath = new PathString("/signout-callback-idsrv");
                options.RemoteSignOutPath = new PathString("/signout-idsrv");

                options.TokenValidationParameters = new TokenValidationParameters
                {
                    NameClaimType = "name",
                    RoleClaimType = "role"
                };
            });
    }
}
```

Figure 8-2. Configuring IdentityServer.

IdentityServer also hosts a public demo site that can be used to test various protocols and configurations. It's located at https://demo.identityserver.io/ and includes information on how to configure its behavior based on the `client_id` provided to it.

JavaScript clients

Many cloud-native applications leverage server-side APIs and rich client single page applications (SPAs) on the front end. IdentityServer ships a JavaScript client (`oidc-client.js`) via NPM that can be added to SPAs to enable them to use IdentityServer for sign in, sign out, and token-based authentication of web APIs.

References

- IdentityServer documentation
- Application types
- JavaScript OIDC client

Security

Not a day goes by where the news doesn't contain some story about a company being hacked or somehow losing their customers' data. Even countries aren't immune to the problems created by treating security as an afterthought. For years, companies have treated the security of customer data and, in fact, their entire networks as something of a "nice to have". Windows servers were left unpatched, ancient versions of PHP kept running and MongoDB databases left wide open to the world.

However, there are starting to be real-world consequences for not maintaining a security mindset when building and deploying applications. Many companies learned the hard way what can happen when servers and desktops aren't patched during the 2017 outbreak of NotPetya. The cost of these attacks has easily reached into the billions, with some estimates putting the losses from this single attack at 10 billion US dollars.

Even governments aren't immune to hacking incidents. The city of Baltimore was held ransom by criminals making it impossible for citizens to pay their bills or use city services.

There has also been an increase in legislation that mandates certain data protections for personal data. In Europe, GDPR has been in effect for more than a year and, more recently, California passed their own version called CCDA, which comes into effect January 1, 2020. The fines under GDPR can be so punishing as to put companies out of business. Google has already been fined 50 million Euros for violations, but that's just a drop in the bucket compared with the potential fines.

In short, security is serious business.

Azure security for cloud-native apps

Cloud-native applications can be both easier and more difficult to secure than traditional applications. On the downside, you need to secure more smaller applications and dedicate more energy to build out the security infrastructure. The heterogeneous nature of programming languages and styles in most service deployments also means you need to pay more attention to security bulletins from many different providers.

On the flip side, smaller services, each with their own data store, limit the scope of an attack. If an attacker compromises one system, it's probably more difficult for the attacker to make the jump to another system than it is in a monolithic application. Process boundaries are strong boundaries. Also, if a database backup leaks, then the damage is more limited, as that database contains only a subset of data and is unlikely to contain personal data.

Threat modeling

No matter if the advantages outweigh the disadvantages of cloud-native applications, the same holistic security mindset must be followed. Security and secure thinking must be part of every step of the development and operations story. When planning an application ask questions like:

- What would be the impact of this data being lost?
- How can we limit the damage from bad data being injected into this service?
- Who should have access to this data?
- Are there auditing policies in place around the development and release process?

All these questions are part of a process called threat modeling. This process tries to answer the question of what threats there are to the system, how likely the threats are, and the potential damage from them.

Once the list of threats has been established, you need to decide whether they're worth mitigating. Sometimes a threat is so unlikely and expensive to plan for that it isn't worth spending energy on it. For instance, some state level actor could inject changes into the design of a process that is used by millions of devices. Now, instead of running a certain piece of code in Ring 3, that code is run in Ring 0. This allows an exploit that can bypass the hypervisor and run the attack code on the bare metal machines, allowing attacks on all the virtual machines that are running on that hardware.

The altered processors are difficult to detect without a microscope and advanced knowledge of the on silicon design of that processor. This scenario is unlikely to happen and expensive to mitigate, so probably no threat model would recommend building exploit protection for it.

More likely threats, such as broken access controls permitting `Id` incrementing attacks (replacing `Id=2` with `Id=3` in the URL) or SQL injection, are more attractive to build protections against. The mitigations for these threats are quite reasonable to build and prevent embarrassing security holes that smear the company's reputation.

Principle of least privilege

One of the founding ideas in computer security is the Principle of Least Privilege (POLP). It's actually a foundational idea in most any form of security be it digital or physical. In short, the principle is that any user or process should have the smallest number of rights possible to execute its task.

As an example, think of the tellers at a bank: accessing the safe is an uncommon activity. So, the average teller can't open the safe themselves. To gain access, they need to escalate their request through a bank manager, who performs additional security checks.

In a computer system, a fantastic example is the rights of a user connecting to a database. In many cases, there's a single user account used to both build the database structure and run the application. Except in extreme cases, the account running the application doesn't need the ability to update schema information. There should be several accounts that provide different levels of privilege. The application should only use the permission level that grants read and write access to the data in the tables. This kind of protection would eliminate attacks that aimed to drop database tables or introduce malicious triggers.

Almost every part of building a cloud-native application can benefit from remembering the principle of least privilege. You can find it at play when setting up firewalls, network security groups, roles, and scopes in Role-based access control (RBAC).

Penetration testing

As applications become more complicated the number of attack vectors increases at an alarming rate. Threat modeling is flawed in that it tends to be executed by the same people building the system. In the same way that many developers have trouble envisioning user interactions and then build unusable user interfaces, most developers have difficulty seeing every attack vector. It's also possible that the developers building the system aren't well versed in attack methodologies and miss something crucial.

Penetration testing or "pen testing" involves bringing in external actors to attempt to attack the system. These attackers may be an external consulting company or other developers with good security knowledge from another part of the business. They're given carte blanche to attempt to subvert the system. Frequently, they'll find extensive security holes that need to be patched. Sometimes the attack vector will be something totally unexpected like exploiting a phishing attack against the CEO.

Azure itself is constantly undergoing attacks from a team of hackers inside Microsoft. Over the years, they've been the first to find dozens of potentially catastrophic attack vectors, closing them before they can be exploited externally. The more tempting a target, the more likely that eternal actors will attempt to exploit it and there are a few targets in the world more tempting than Azure.

Monitoring

Should an attacker attempt to penetrate an application, there should be some warning of it. Frequently, attacks can be spotted by examining the logs from services. Attacks leave telltale signs that can be spotted before they succeed. For instance, an attacker attempting to guess a password will make many requests to a login system. Monitoring around the login system can detect weird patterns that are out of line with the typical access pattern. This monitoring can be turned into an alert that can, in turn, alert an operations person to activate some sort of countermeasure. A highly mature monitoring system might even take action based on these deviations proactively adding rules to block requests or throttle responses.

Securing the build

One place where security is often overlooked is around the build process. Not only should the build run security checks, such as scanning for insecure code or checked-in credentials, but the build itself should be secure. If the build server is compromised, then it provides a fantastic vector for introducing arbitrary code into the product.

Imagine that an attacker is looking to steal the passwords of people signing into a web application. They could introduce a build step that modifies the checked-out code to mirror any login request to another server. The next time code goes through the build, it's silently updated. The source code vulnerability scanning won't catch this as it runs before the build. Equally, nobody will catch it in a

code review because the build steps live on the build server. The exploited code will go to production where it can harvest passwords. Probably there's no audit log of the build process changes, or at least nobody monitoring the audit.

This is a perfect example of a seemingly low value target that can be used to break into the system. Once an attacker breaches the perimeter of the system, they can start working on finding ways to elevate their permissions to the point that they can cause real harm anywhere they like.

Building secure code

The .NET Framework is already a quite secure framework. It avoids some of the pitfalls of unmanaged code, such as walking off the ends of arrays. Work is actively done to fix security holes as they're discovered. There's even a bug bounty program that pays researchers to find issues in the framework and report them instead of exploiting them.

There are many ways to make .NET code more secure. Following guidelines such as the Secure coding guidelines for .NET article is a reasonable step to take to ensure that the code is secure from the ground up. The OWASP top 10 is another invaluable guide to build secure code.

The build process is a good place to put scanning tools to detect problems in source code before they make it into production. Most every project has dependencies on some other packages. A tool that can scan for outdated packages will catch problems in a nightly build. Even when building Docker images, it's useful to check and make sure that the base image doesn't have known vulnerabilities. Another thing to check is that nobody has accidentally checked in credentials.

Built-in security

Azure is designed to balance usability and security for the majority of users. Different users are going to have different security requirements, so they need to fine-tune their approach to cloud security. Microsoft publishes a great deal of security information in the Trust Center. This resource should be the first stop for those professionals interested in understanding how the built-in attack mitigation technologies work.

Within the Azure portal, the Azure Advisor is a system that is constantly scanning an environment and making recommendations. Some of these recommendations are designed to save users money, but others are designed to identify potentially insecure configurations, such as having a storage container open to the world and not protected by a Virtual Network.

Azure network infrastructure

In an on-premises deployment environment, a great deal of energy is dedicated to setting up networking. Setting up routers, switches, and the such is complicated work. Networks allow certain resources to talk to other resources and prevent access in some cases. A frequent network rule is to restrict access to the production environment from the development environment on the off chance that a half-developed piece of code runs awry and deletes a swath of data.

Out of the box, most PaaS Azure resources have only the most basic and permissive networking setup. For instance, anybody on the Internet can access an app service. New SQL Server instances typically

come restricted, so that external parties can't access them, but the IP address ranges used by Azure itself are permitted through. So, while the SQL server is protected from external threats, an attacker only needs to set up an Azure bridgehead from where they can launch attacks against all SQL instances on Azure.

Fortunately, most Azure resources can be placed into an Azure Virtual Network that allows finer grained access control. Similar to the way that on-premises networks establish private networks that are protected from the wider world, virtual networks are islands of private IP addresses that are located within the Azure network.

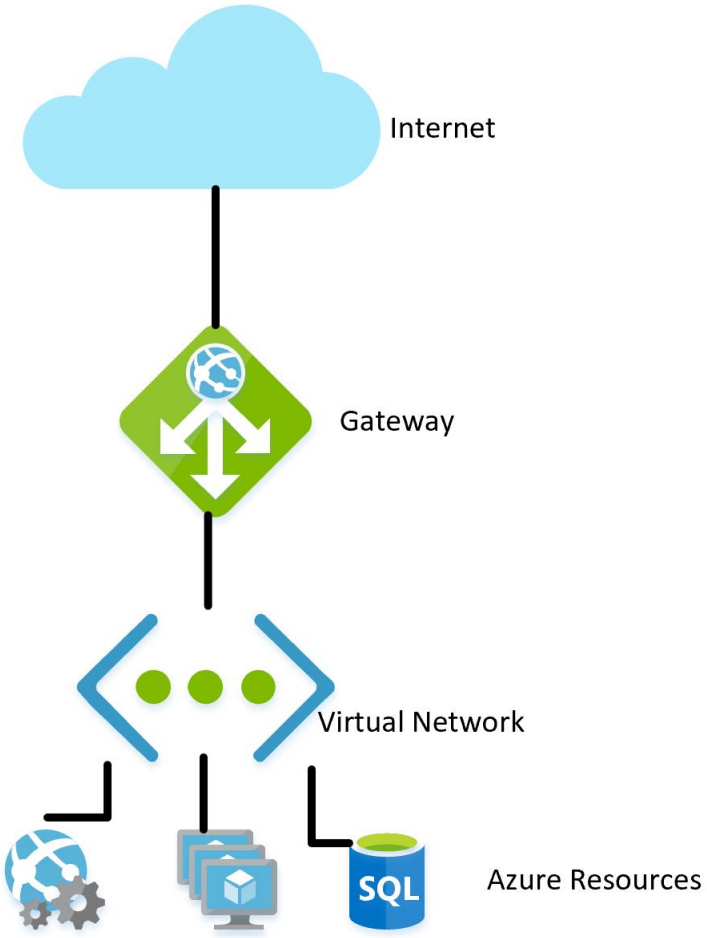


Figure 10-1. A virtual network in Azure.

In the same way that on-premises networks have a firewall governing access to the network, you can establish a similar firewall at the boundary of the virtual network. By default, all the resources on a virtual network can still talk to the Internet. It's only incoming connections that require some form of explicit firewall exception.

With the network established, internal resources like storage accounts can be set up to only allow for access by resources that are also on the Virtual Network. This firewall provides an extra level of security, should the keys for that storage account be leaked, attackers wouldn't be able to connect to it to exploit the leaked keys. This is another example of the principle of least privilege.

The nodes in an Azure Kubernetes cluster can participate in a virtual network just like other resources that are more native to Azure. This functionality is called Azure Container Networking Interface. In effect, it allocates a subnet within the virtual network on which virtual machines and container images are allocated.

Continuing down the path of illustrating the principle of least privilege, not every resource within a Virtual Network needs to talk to every other resource. For instance, in an application that provides a web API over a storage account and a SQL database, it's unlikely that the database and the storage account need to talk to one another. Any data sharing between them would go through the web application. So, a network security group (NSG) could be used to deny traffic between the two services.

A policy of denying communication between resources can be annoying to implement, especially coming from a background of using Azure without traffic restrictions. On some other clouds, the concept of network security groups is much more prevalent. For instance, the default policy on AWS is that resources can't communicate among themselves until enabled by rules in an NSG. While slower to develop this, more restrictive environment provides a more secure default. Making use of proper DevOps practices, especially using Azure Resource Manager or Terraform to manage permissions can make controlling the rules easier.

Virtual Networks can also be useful when setting up communication between on-premises and cloud resources. A virtual private network can be used to seamlessly attach the two networks together. This allows running a virtual network without any sort of gateway for scenarios where all the users are on-site. There are a number of technologies that can be used to establish this network. The simplest is to use a site-to-site VPN that can be established between many routers and Azure. Traffic is encrypted and tunneled over the Internet at the same cost per byte as any other traffic. For scenarios where more bandwidth or more security is desirable, Azure offers a service called Express Route that uses a private circuit between an on-premises network and Azure. It's more costly and difficult to establish but also more secure.

Role-based access control for restricting access to Azure resources

RBAC is a system that provides an identity to applications running in Azure. Applications can access resources using this identity instead of or in addition to using keys or passwords.

Security Principals

The first component in RBAC is a security principal. A security principal can be a user, group, service principal, or managed identity.

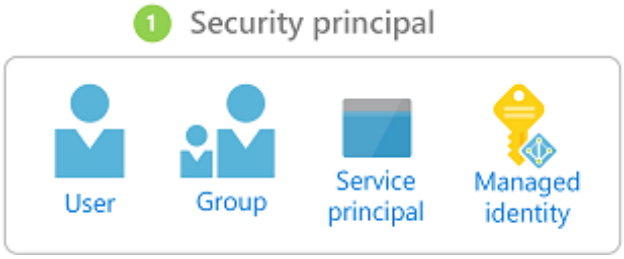


Figure 10-2. Different types of security principals.

- User - Any user who has an account in Azure Active Directory is a user.
- Group - A collection of users from Azure Active Directory. As a member of a group, a user takes on the roles of that group in addition to their own.
- Service principal - A security identity under which services or applications run.
- Managed identity - An Azure Active Directory identity managed by Azure. Managed identities are typically used when developing cloud applications that manage the credentials for authenticating to Azure services.

The security principal can be applied to most any resource. This means that it's possible to assign a security principal to a container running within Azure Kubernetes, allowing it to access secrets stored in Key Vault. An Azure Function could take on a permission allowing it to talk to an Active Directory instance to validate a JWT for a calling user. Once services are enabled with a service principal, their permissions can be managed granularly using roles and scopes.

Roles

A security principal can take on many roles or, using a more sartorial analogy, wear many hats. Each role defines a series of permissions such as "Read messages from Azure Service Bus endpoint". The effective permission set of a security principal is the combination of all the permissions assigned to all the roles that security principal has. Azure has a large number of built-in roles and users can define their own roles.

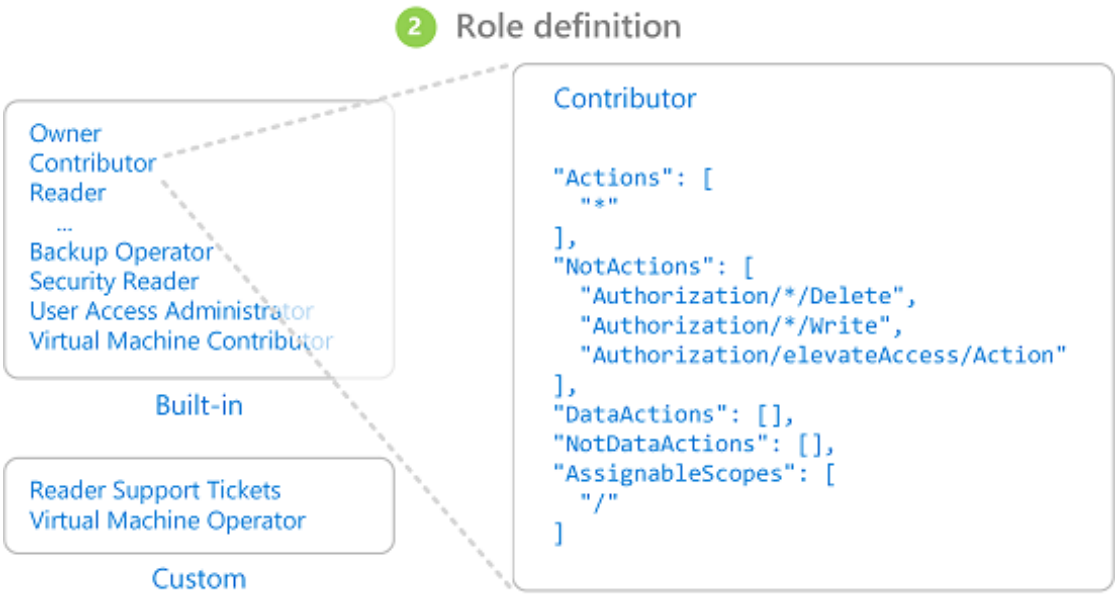


Figure 10-3. RBAC role definitions.

Built into Azure are also a number of high-level roles such as Owner, Contributor, Reader, and User Account Administrator. With the Owner role, a security principal can access all resources and assign permissions to others. A contributor has the same level of access to all resources but they can't assign permissions. A Reader can only view existing Azure resources and a User Account Administrator can manage access to Azure resources.

More granular built-in roles such as DNS Zone Contributor have rights limited to a single service. Security principals can take on any number of roles.

Scopes

Roles can be applied to a restricted set of resources within Azure. For instance, applying scope to the previous example of reading from a Service Bus queue, you can narrow the permission to a single queue: "Read messages from Azure Service Bus endpoint `blah.servicebus.windows.net/queue1`"

The scope can be as narrow as a single resource or it can be applied to an entire resource group, subscription, or even management group.

When testing if a security principal has a certain permission, the combination of role and scope are taken into account. This combination provides a powerful authorization mechanism.

Deny

Previously, only "allow" rules were permitted for RBAC. This behavior made some scopes complicated to build. For instance, allowing a security principal access to all storage accounts except one required granting explicit permission to a potentially endless list of storage accounts. Every time a new storage account was created, it would have to be added to this list of accounts. This added management overhead that certainly wasn't desirable.

Deny rules take precedence over allow rules. Now representing the same "allow all but one" scope could be represented as two rules "allow all" and "deny this one specific one". Deny rules not only ease management but allow for resources that are extra secure by denying access to everybody.

Checking access

As you can imagine, having a large number of roles and scopes can make figuring out the effective permission of a service principal quite difficult. Piling deny rules on top of that, only serves to increase the complexity. Fortunately, there's a permissions calculator that can show the effective permissions for any service principal. It's typically found under the IAM tab in the portal, as shown in Figure 10-3.

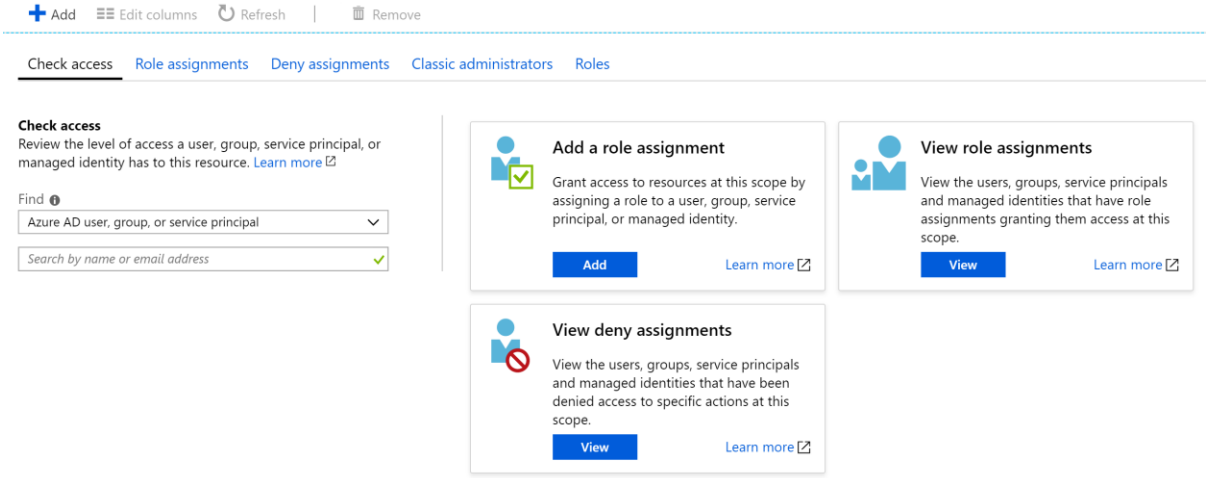


Figure 10-4. Permission calculator for an app service.

Securing secrets

Passwords and certificates are a common attack vector for attackers. Password-cracking hardware can do a brute-force attack and try to guess billions of passwords per second. So it's important that the passwords that are used to access resources are strong, with a large variety of characters. These passwords are exactly the kind of passwords that are near impossible to remember. Fortunately, the passwords in Azure don't actually need to be known by any human.

Many security experts suggest that using a password manager to keep your own passwords is the best approach. While it centralizes your passwords in one location, it also allows using highly complex passwords and ensuring they're unique for each account. The same system exists within Azure: a central store for secrets.

Azure Key Vault

Azure Key Vault provides a centralized location to store passwords for things such as databases, API keys, and certificates. Once a secret is entered into the Vault, it's never shown again and the commands to extract and view it are purposefully complicated. The information in the safe is

protected using either software encryption or FIPS 140-2 Level 2 validated Hardware Security Modules.

Access to the key vault is provided through RBACs, meaning that not just any user can access the information in the vault. Say a web application wishes to access the database connection string stored in Azure Key Vault. To gain access, applications need to run using a service principal. Under this assumed role, they can read the secrets from the safe. There are a number of different security settings that can further limit the access that an application has to the vault, so that it can't update secrets but only read them.

Access to the key vault can be monitored to ensure that only the expected applications are accessing the vault. The logs can be integrated back into Azure Monitor, unlocking the ability to set up alerts when unexpected conditions are encountered.

Kubernetes

Within Kubernetes, there's a similar service for maintaining small pieces of secret information. Kubernetes Secrets can be set via the typical `kubectl` executable.

Creating a secret is as simple as finding the base64 version of the values to be stored:

```
echo -n 'admin' | base64
YWRtaW4=
echo -n '1f2d1e2e67df' | base64
MWYyZDFlMmU2N2Rm
```

Then adding it to a secrets file named `secret.yml` for example that looks similar to the following example:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

Finally, this file can be loaded into Kubernetes by running the following command:

```
kubectl apply -f ./secret.yaml
```

These secrets can then be mounted into volumes or exposed to container processes through environment variables. The Twelve-factor app approach to building applications suggests using the lowest common denominator to transmit settings to an application. Environment variables are the lowest common denominator, because they're supported no matter the operating system or application.

An alternative to use the built-in Kubernetes secrets is to access the secrets in Azure Key Vault from within Kubernetes. The simplest way to do this is to assign an RBAC role to the container looking to load secrets. The application can then use the Azure Key Vault APIs to access the secrets. However, this approach requires modifications to the code and doesn't follow the pattern of using environment variables. Instead, it's possible to inject values into a container through the use of the Azure Key Vault

Injector. This approach is actually more secure than using the Kubernetes secrets directly, as they can be accessed by users on the cluster.

Encryption in transit and at rest

Keeping data safe is important whether it's on disk or transiting between various different services. The most effective way to keep data from leaking is to encrypt it into a format that can't be easily read by others. Azure supports a wide range of encryption options.

In transit

There are several ways to encrypt traffic on the network in Azure. The access to Azure services is typically done over connections that use Transport Layer Security (TLS). For instance, all the connections to the Azure APIs require TLS connections. Equally, connections to endpoints in Azure storage can be restricted to work only over TLS encrypted connections.

TLS is a complicated protocol and simply knowing that the connection is using TLS isn't sufficient to ensure security. For instance, TLS 1.0 is chronically insecure, and TLS 1.1 isn't much better. Even within the versions of TLS, there are various settings that can make the connections easier to decrypt. The best course of action is to check and see if the server connection is using up-to-date and well configured protocols.

This check can be done by an external service such as SSL labs' SSL Server Test. A test run against a typical Azure endpoint, in this case a service bus endpoint, yields a near perfect score of A.

Even services like Azure SQL databases use TLS encryption to keep data hidden. The interesting part about encrypting the data in transit using TLS is that it isn't possible, even for Microsoft, to listen in on the connection between computers running TLS. This should provide comfort for companies concerned that their data may be at risk from Microsoft proper or even a state actor with more resources than the standard attacker.

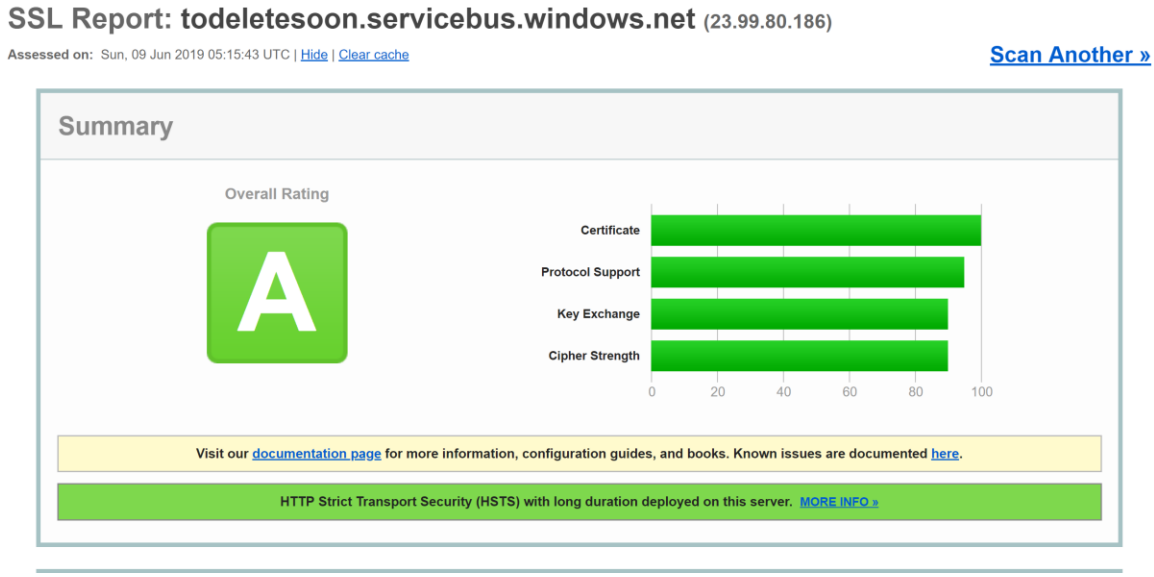


Figure 10-5. SSL labs report showing a score of A for a Service Bus endpoint.

While this level of encryption isn't going to be sufficient for all time, it should inspire confidence that Azure TLS connections are quite secure. Azure will continue to evolve its security standards as encryption improves. It's nice to know that there's somebody watching the security standards and updating Azure as they improve.

At rest

In any application, there are a number of places where data rests on disk. The application code itself is loaded from some storage mechanism. Most applications also use some kind of database such as SQL Server, Cosmos DB, or even the amazingly price-efficient Table Storage. These databases all use heavily encrypted storage to ensure that nobody other than the applications with proper permissions can read your data. Even the system operators can't read data that has been encrypted. So customers can remain confident their secret information remains secret.

Storage

The underpinning of much of Azure is the Azure Storage engine. Virtual machine disks are mounted on top of Azure Storage. Azure Kubernetes Services run on virtual machines that, themselves, are hosted on Azure Storage. Even serverless technologies, such as Azure Functions Apps and Azure Container Instances, run out of disk that is part of Azure Storage.

If Azure Storage is well encrypted, then it provides for a foundation for most everything else to also be encrypted. Azure Storage is encrypted with FIPS 140-2 compliant 256-bit AES. This is a well-regarded encryption technology having been the subject of extensive academic scrutiny over the last 20 or so years. At present, there's no known practical attack that would allow someone without knowledge of the key to read data encrypted by AES.

By default, the keys used for encrypting Azure Storage are managed by Microsoft. There are extensive protections in place to ensure to prevent malicious access to these keys. However, users with particular encryption requirements can also provide their own storage keys, that are managed in Azure Key Vault. These keys can be revoked at any time, which would effectively render the contents of the Storage account using them inaccessible.

Virtual machines use encrypted storage, but it's possible to provide another layer of encryption by using technologies like BitLocker on Windows or DM-Crypt on Linux. These technologies mean that even if the disk image was leaked off of storage, it would remain near impossible to read it.

Azure SQL

Databases hosted on Azure SQL use a technology called Transparent Data Encryption (TDE) to ensure data remains encrypted. It's enabled by default on all newly created SQL databases, but must be enabled manually for legacy databases. TDE executes real-time encryption and decryption of not just the database, but also the backups and transaction logs.

The encryption parameters are stored in the `master` database and, on startup, are read into memory for the remaining operations. This means that the `master` database must remain unencrypted. The actual key is managed by Microsoft. However, users with exacting security requirements may provide their own key in Key Vault in much the same way as is done for Azure Storage. The Key Vault provides for such services as key rotation and revocation.

The "Transparent" part of TDS comes from the fact that there aren't client changes needed to use an encrypted database. While this approach provides for good security, leaking the database password is enough for users to be able to decrypt the data. There's another approach that encrypts individual columns or tables in a database. Always Encrypted ensures that at no point the encrypted data appears in plain text inside the database.

Setting up this tier of encryption requires running through a wizard in SQL Server Management Studio to select the sort of encryption and where in Key Vault to store the associated keys.

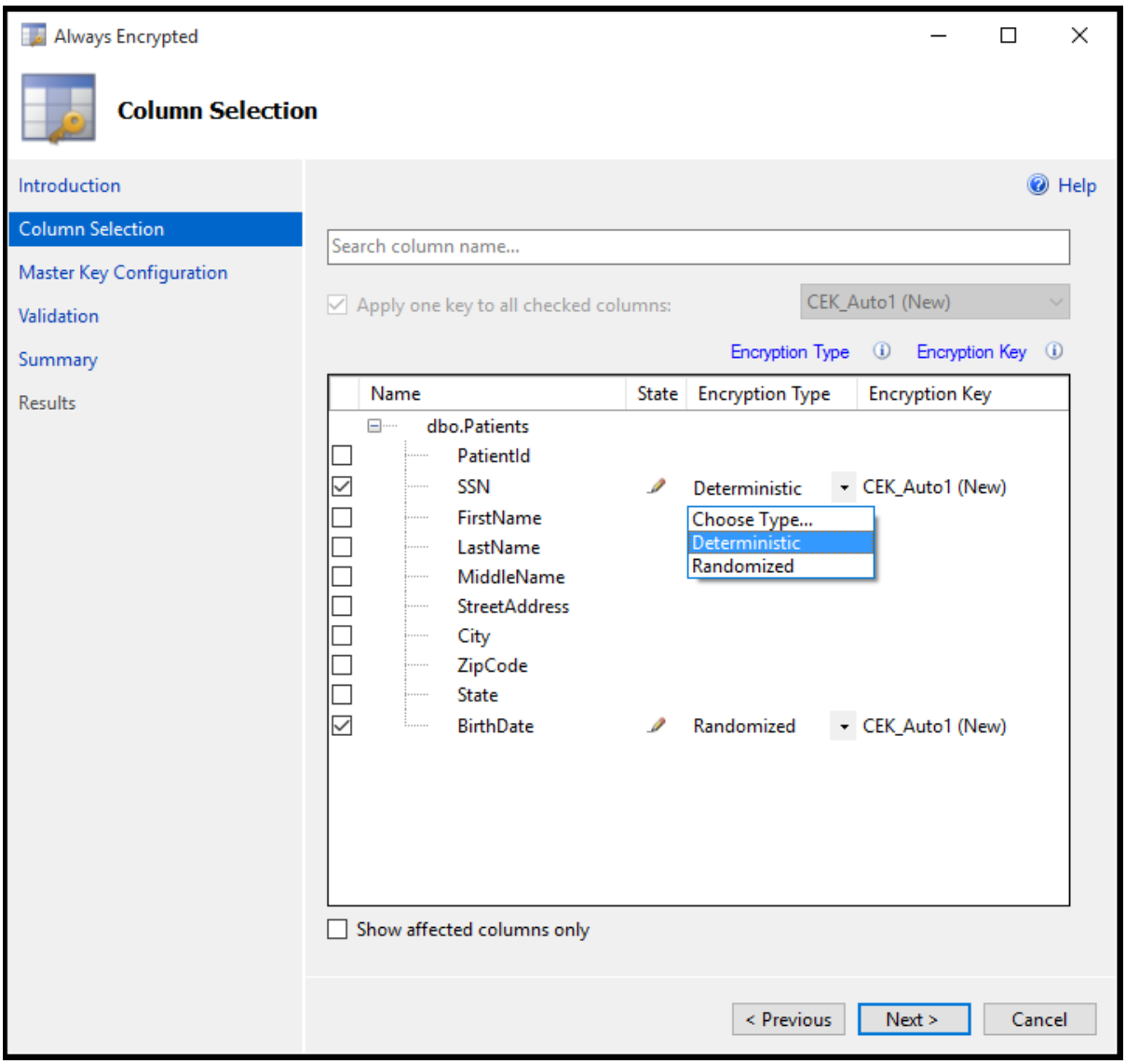


Figure 10-6. Selecting columns in a table to be encrypted using Always Encrypted.

Client applications that read information from these encrypted columns need to make special allowances to read encrypted data. Connection strings need to be updated with `Column Encryption Setting=Enabled` and client credentials must be retrieved from the Key Vault. The SQL Server client must then be primed with the column encryption keys. Once that is done, the remaining actions use the standard interfaces to SQL Client. That is, tools like Dapper and Entity Framework, which are built

on top of SQL Client, will continue to work without changes. Always Encrypted may not yet be available for every SQL Server driver on every language.

The combination of TDE and Always Encrypted, both of which can be used with client-specific keys, ensures that even the most exacting encryption requirements are supported.

Cosmos DB

Cosmos DB is the newest database provided by Microsoft in Azure. It has been built from the ground up with security and cryptography in mind. AES-256bit encryption is standard for all Cosmos DB databases and can't be disabled. Coupled with the TLS 1.2 requirement for communication, the entire storage solution is encrypted.

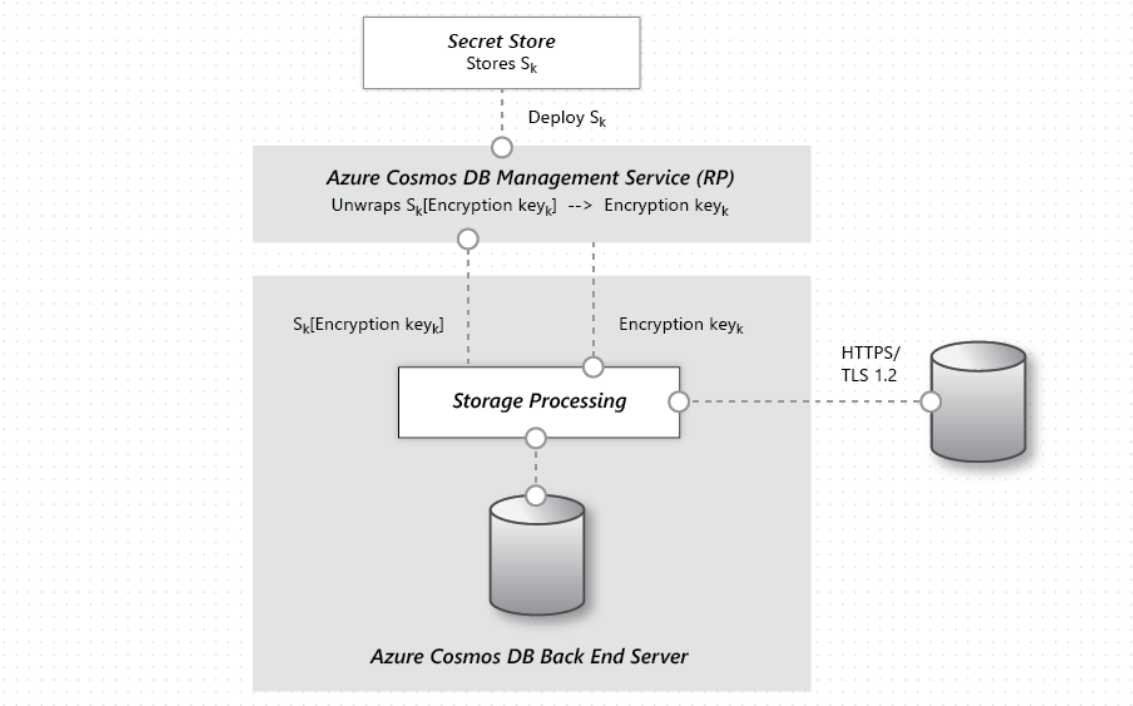


Figure 10-7. The flow of data encryption within Cosmos DB.

While Cosmos DB doesn't provide for supplying customer encryption keys, there has been significant work done by the team to ensure it remains PCI-DSS compliant without that. Cosmos DB also doesn't support any sort of single column encryption similar to Azure SQL's Always Encrypted yet.

Keeping secure

Azure has all the tools necessary to release a highly secure product. However, a chain is only as strong as its weakest link. If the applications deployed on top of Azure aren't developed with a proper security mindset and good security audits, then they become the weak link in the chain. There are many great static analysis tools, encryption libraries, and security practices that can be used to ensure that the software installed on Azure is as secure as Azure itself. static analysis tools, encryption libraries, and security practices, LibreSSL and Red vs. Blue - Internal security penetration testing of Microsoft Azure are examples of that, respectively.

Cloud Native DevOps

The favorite mantra of software consultants is to answer "It depends" to any question posed. It isn't because software consultants are fond of not taking a position. It's because there's no one true answer to any questions in software. There's no absolute right and wrong, but rather a balance between opposites.

Take, for instance, the two major schools of developing web applications: Single Page Applications (SPAs) versus server-side applications. On the one hand, the user experience tends to be better with SPAs and the amount of traffic to the web server can be minimized making it possible to host them on something as simple as static hosting. On the other hand, SPAs tend to be slower to develop and more difficult to test. Which one is the right choice? Well, it depends on your situation.

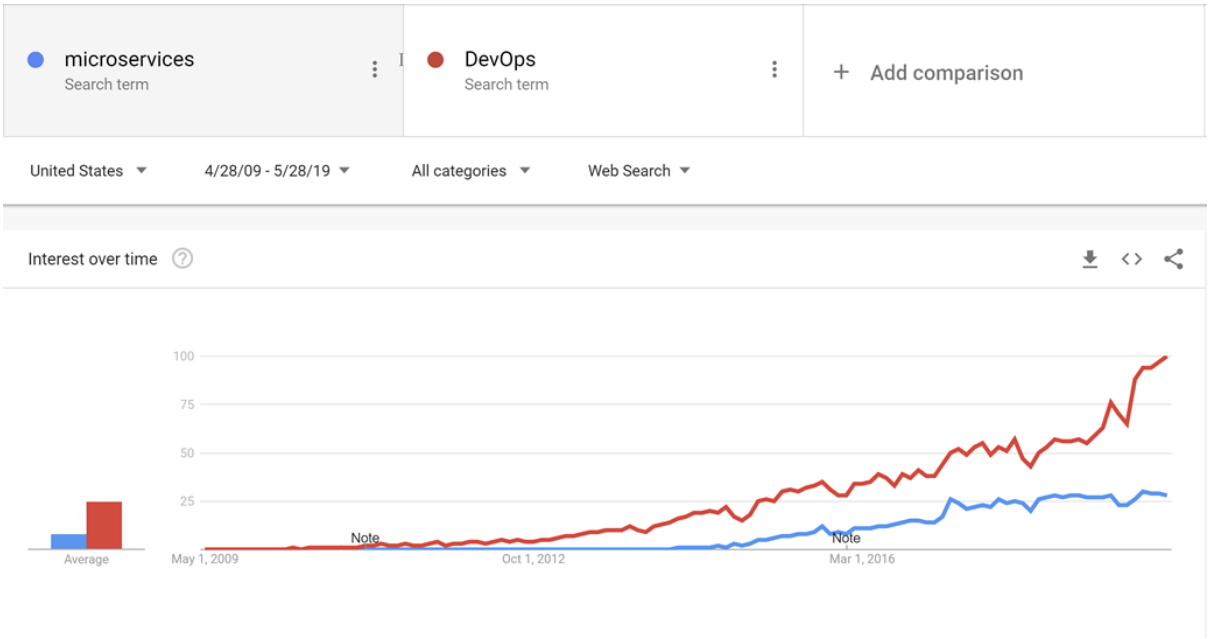
Cloud-native applications aren't immune to that same dichotomy. They have clear advantages in terms of speed of development, stability, and scalability, but managing them can be quite a bit more difficult.

Years ago, it wasn't uncommon for the process of moving an application from development to production to take a month, or even more. Companies released software on a 6-month or even every year cadence. One needs to look no further than Microsoft Windows to get an idea for the cadence of releases that were acceptable before the ever-green days of Windows 10. Five years passed between Windows XP and Vista, a further 3 between Vista and Windows 7.

It's now fairly well established that being able to release software rapidly gives fast-moving companies a huge market advantage over their more sloth-like competitors. It's for that reason that major updates to Windows 10 are now approximately every six months.

The patterns and practices that enable faster, more reliable releases to deliver value to the business are collectively known as DevOps. They consist of a wide range of ideas spanning the entire software development life cycle from specifying an application all the way up to delivering and operating that application.

DevOps emerged before microservices and it's likely that the movement towards smaller, more fit to purpose services wouldn't have been possible without DevOps to make releasing and operating not just one but many applications in production easier.



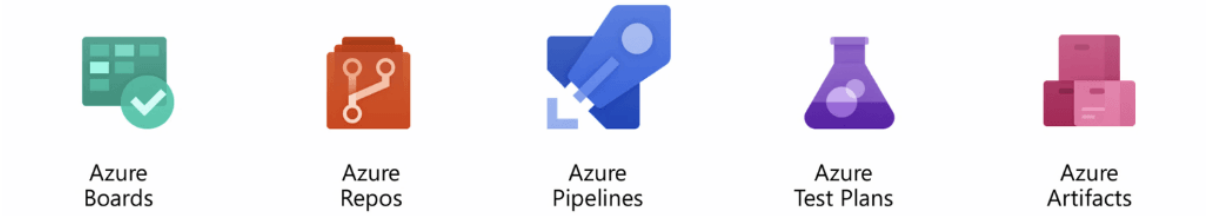
Through good DevOps practices, it's possible to realize the advantages of cloud-native applications without suffocating under a mountain of work actually operating the applications.

There's no golden hammer when it comes to DevOps. Nobody can sell a complete and all-encompassing solution for releasing and operating high-quality applications. This is because each application is wildly different from all others. However, there are tools that can make DevOps a far less daunting proposition. One of these tools is known as Azure DevOps.

Azure DevOps

Azure DevOps has a long pedigree. It can trace its roots back to when Team Foundation Server first moved online and through the various name changes: Visual Studio Online and Visual Studio Team Services. Through the years, however, it has become far more than its predecessors.

Azure DevOps is divided into five major components:



Azure Boards - Provides an issue and work item tracking tool that strives to allow users to pick the workflows that work best for them. It comes with a number of pre-configured templates including ones to support SCRUM and Kanban styles of development.

Azure Repos - Source code management that supports the venerable Team Foundation Version Control (TFVC) and the industry favorite git. Pull requests provide a way to enable social coding by fostering discussion of changes as they're made.

Azure Pipelines - A build and release management system that supports tight integration with Azure. Builds can be run on a variety of platforms from Windows to Linux to MacOS. Build agents may be provisioned in the cloud or on-premises.

Azure Test Plans - No QA person will be left behind with the test management and exploratory testing support offered by the Test Plans feature.

Azure Artifacts - An artifact feed that allows companies to create their own, internal, versions of NuGet, npm, and others. It serves a double purpose of acting as a cache of upstream packages if there's a failure of a centralized repository.

The top-level organizational unit in Azure DevOps is known as a Project. Within each project the various components, such as Azure Artifacts, can be turned on and off. If users want to manage their source code in GitHub but still take advantage of Azure Pipelines, then that's perfectly possible. In fact, many open-source projects leverage the free builds offered by Azure DevOps while keeping their source code on GitHub. Some significant open-source projects such as Visual Studio Code, yarn, gulp, and NumPy have made the transition.

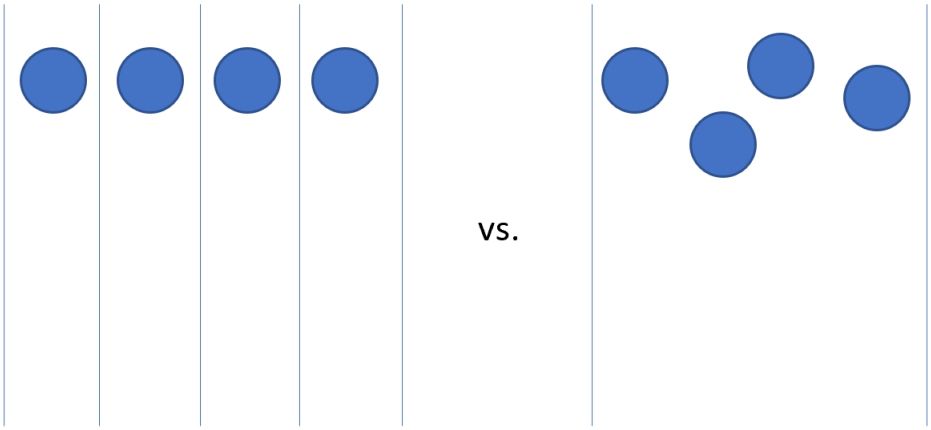
Each of these components provides some advantages for cloud-native applications, but the three most useful are the source control, boards and pipelines.

Source control

Organizing the code for a cloud-native application can be challenging. Instead of a single giant application, the cloud-native applications tend to be made up of a web of smaller applications that talk with one another. As with all things in computing, the best arrangement of code remains an open question. There are examples of successful applications using different kinds of layouts, but two variants seem to have the most popularity.

Before getting down into the actual source control itself, it's probably worth deciding on how many projects are appropriate. Within a single project, there's support for multiple repositories, and build pipelines. Boards are a little more complicated but there too the tasks can easily be assigned to multiple teams within a single project. It's certainly possible to support hundreds, even thousands of developers, out of a single Azure DevOps project. Doing so is likely the best approach as it provides a single place for all developer to work out of and reduces the confusion of finding that one application when developers are unsure in which project in which it resides.

Splitting up code for microservices within the Azure DevOps project can be slightly more challenging.



Repository per Microservice Single Repository

Repository per microservice

At first glance, this seems like the most logical approach to splitting up the source code for microservices. Each repository can contain the code needed to build the one microservice. The advantages to this approach are readily visible:

1. Instructions for building and maintaining the application can be added to a README file at the root of each repository. When flipping through the repositories, it's easy to find these instructions, reducing spin-up time for developers.
2. Every service is located in a logical place, easily found by knowing the name of the service.
3. Builds can easily be set up such that they're only triggered when a change is made to the owning repository.
4. The number of changes coming into a repository is limited to the small number of developers working on the project.
5. Security is easy to set up by restricting the repositories to which developers have read and write permissions.
6. Repository level settings can be changed by the owning team with a minimum of discussion with others.

One of the key ideas behind microservices is that services should be siloed and separated from each other. When using Domain Driven Design to decide on the boundaries for services the services act as transactional boundaries. Database updates shouldn't span multiple services. This collection of related data is referred to as a bounded context. This idea is reflected by the isolation of microservice data to a database separate and autonomous from the rest of the services. It makes a great deal of sense to carry this idea all the way through to the source code.

However, this approach isn't without its issues. One of the more gnarly development problems of our time is managing dependencies. Consider the number of files that make up the average

`node_modules` directory. A fresh install of something like `create-react-app` is likely to bring with it thousands of packages. The question of how to manage these dependencies is a difficult one.

If a dependency is updated, then downstream packages must also update this dependency. Unfortunately, that takes development work so, invariably, the `node_modules` directory ends up with multiple versions of a single package, each one a dependency of some other package that is versioned at a slightly different cadence. When deploying an application, which version of a dependency should be used? The version that is currently in production? The version that is currently in Beta but is likely to be in production by the time the consumer makes it to production? Difficult problems that aren't resolved by just using microservices.

There are libraries that are depended upon by a wide variety of projects. By dividing the microservices up with one in each repository the internal dependencies can best be resolved by using the internal repository, Azure Artifacts. Builds for libraries will push their latest versions into Azure Artifacts for internal consumption. The downstream project must still be manually updated to take a dependency on the newly updated packages.

Another disadvantage presents itself when moving code between services. Although it would be nice to believe that the first division of an application into microservices is 100% correct, the reality is that rarely we're so prescient as to make no service division mistakes. Thus, functionality and the code that drives it will need to move from service to service: repository to repository. When leaping from one repository to another, the code loses its history. There are many cases, especially in the event of an audit, where having full history on a piece of code is invaluable.

The final and perhaps most important disadvantage is coordinating changes. In a true microservices application, there should be no deployment dependencies between services. It should be possible to deploy services A, B, and C in any order as they have loose coupling. In reality, however, there are times when it's desirable to make a change that crosses multiple repositories at the same time. Some examples include updating a library to close a security hole or changing a communication protocol used by all services.

To do a cross-repository change requires a commit to each repository be made in succession. Each change in each repository will need to be pull-requested and reviewed separately. This can be difficult to coordinate and generally annoying to do.

An alternative to using many repositories is to put all the source code together in a giant, all knowing, single repository.

Single repository

In this approach, sometimes referred to as a monorepository, all the source code for every service is put into the same repository. At first, this seems like a terrible idea likely to make dealing with source code unwieldy. There are, however, some marked advantages to working this way.

The first advantage is that it's easier to manage dependencies between projects. Instead of relying on some external artifact feed, projects can directly import one another. This means that updates are instant, and conflicting versions are likely to be found at compile time on the developer's workstation. In effect, shifting some of the integration testing left.

When moving code between projects, it's now easier to preserve the history as the files will be detected as having been moved rather than being rewritten.

Another advantage is that wide ranging changes that cross service boundaries can be made in a single commit. This reduces the overhead of having potentially dozens of changes to review individually.

There are many tools that can perform static analysis of code to detect insecure programming practices or problematic use of APIs. In a multi-repository world, each repository will need to be iterated over to find the problems in them. The single repository allows running the analysis all in one place.

There are also many disadvantages to the single repository approach. One of the most worrying ones is that having a single repository raises security concerns. If the contents of a repository are leaked in a repository per service model, the amount of code lost is minimal. With a single repository, everything the company owns could be lost. There have been many examples in the past of this happening and derailing entire game development efforts. Having multiple repositories exposes less surface area, which is a very desirable trait in most security practices.

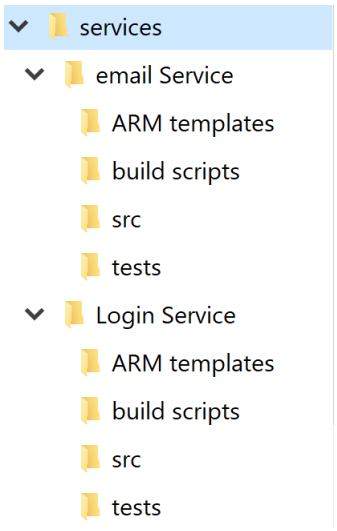
The size of the single repository is likely to become unmanageable rapidly. This presents some interesting performance implications. It may become necessary to use specialized tools such as Virtual File System for Git, which was originally designed to improve the experience for developers on the Windows team.

Frequently the argument for using a single repository boils down to an argument that Facebook or Google use this method for source code arrangement. If the approach is good enough for these companies, then, surely, it's the correct approach for all companies. The truth of the matter is that very few companies operate on anything like the scale of Facebook or Google. The problems that occur at those scales are different from those most developers will face. What is good for the goose may not be good for the gander.

In the end, either solution can be used to host the source code for microservices. However, in most cases, the management and engineering overhead of operating in a single repository isn't worth the meager advantages. Splitting code up over multiple repositories encourages better separation of concerns and encourages autonomy among development teams.

Standard directory structure

Regardless of the single versus multiple repositories debate each service will have its own directory. One of the best optimizations to allow developers to cross between projects quickly is to maintain a standard directory structure.



Whenever a new project is created, a template that puts in place the correct structure should be used. This template can also include such useful items as a skeleton README file and an `azure-pipelines.yml`. In any microservice architecture, a high degree of variance between projects makes bulk operations against the services more difficult.

There are many tools that can provide templating for an entire directory, containing several source code directories. Yeoman is popular in the JavaScript world and GitHub have recently released Repository Templates, which provide much of the same functionality.

Task management

Managing tasks in any project can be difficult. Up front there are countless questions to be answered about the sort of workflows to set up to ensure optimal developer productivity.

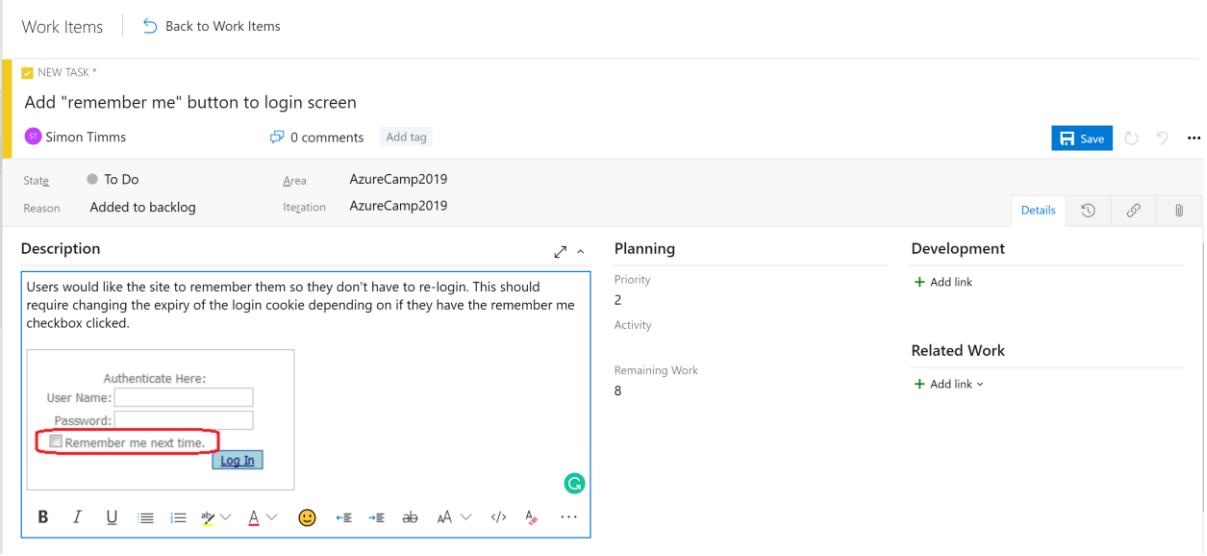
Cloud-native applications tend to be smaller than traditional software products or at least they're divided into smaller services. Tracking of issues or tasks related to these services remains as important as with any other software project. Nobody wants to lose track of some work item or explain to a customer that their issue wasn't properly logged. Boards are configured at the project level but within each project, areas can be defined. These allow breaking down issues across several components. The advantage to keeping all the work for the entire application in one place is that it's easy to move work items from one team to another as they're understood better.

Azure DevOps comes with a number of popular templates pre-configured. In the most basic configuration, all that is needed to know is what's in the backlog, what people are working on, and what's done. It's important to have this visibility into the process of building software, so that work can be prioritized and completed tasks reported to the customer. Of course, very few software projects stick to a process as simple as `to do`, `doing`, and `done`. It doesn't take long for people to start adding steps like `QA` or `Detailed Specification` to the process.

One of the more important parts of Agile methodologies is self-introspection at regular intervals. These reviews are meant to provide insight into what problems the team is facing and how they can be improved. Frequently, this means changing the flow of issues and features through the

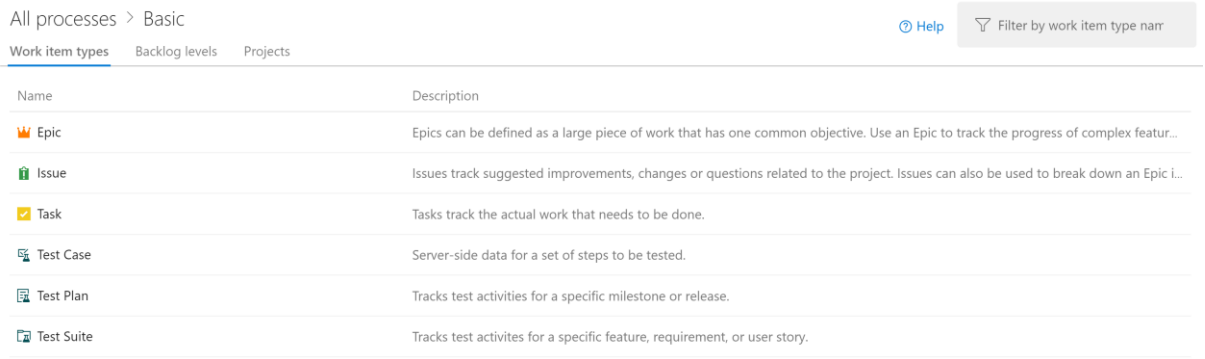
development process. So, it's perfectly healthy to expand the layouts of the boards with additional stages.

The stages in the boards aren't the only organizational tool. Depending on the configuration of the board, there's a hierarchy of work items. The most granular item that can appear on a board is a task. Out of the box a task contains fields for a title, description, a priority, an estimate of the amount of work remaining and the ability to link to other work items or development items (branches, commits, pull requests, builds, and so forth). Work items can be classified into different areas of the application and different iterations (sprints) to make finding them easier.



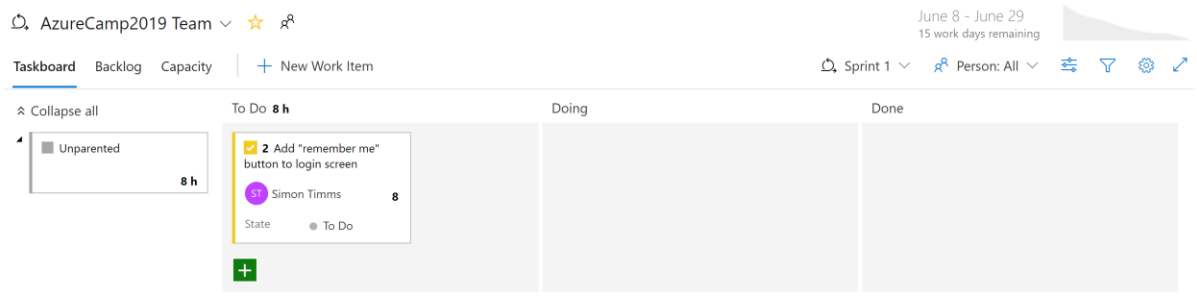
The description field supports the normal styles you'd expect (bold, italic underscore and strike through) and the ability to insert images. This makes it a very powerful tool for use when specifying work or bugs.

Tasks can be rolled up into features, which define a larger unit of work. Features, in turn, can be rolled up into epics. Classifying tasks in this hierarchy makes it much easier to understand how close a large feature is to rolling out.



There are different kinds of views into the issues in Azure Boards. Items that aren't yet scheduled appear in the backlog. From there, they can be assigned to a sprint. A sprint is a time box during

which it's expected some quantity of work will be completed. This work can include tasks but also the resolution of tickets. Once there, the entire sprint can be managed from the Sprint board section. This view shows how work is progressing and includes a burn down chart to give an ever-updating estimate of if the sprint will be successful.



By now, it should be apparent that there's a great deal of power in the Boards in Azure DevOps. For developers, there are easy views of what is being worked on. For project managers views into upcoming work as well as an overview of existing work. For managers, there are plenty of reports about resourcing and capacity. Unfortunately, there's nothing magical about cloud-native applications that eliminate the need to track work. But if you must track work, there are a few places where the experience is better than in Azure DevOps.

CI/CD pipelines

Almost no change in the software development life cycle has been so revolutionary as the advent of continuous integration (CI) and continuous delivery (CD). Building and running automated tests against the source code of a project as soon as a change is checked in catches mistakes early. Prior to the advent of continuous integration builds, it wouldn't be uncommon to pull code from the repository and find that it didn't pass tests or couldn't even be built. This resulted in a lot of tracking down the source of the breakage.

Traditionally shipping software to the production environment required extensive documentation and a list of steps. Each one of these steps needed to be manually completed in a very error prone process.

The sister of continuous integration is continuous delivery in which the freshly built packages are deployed to an environment. The manual process can't scale to match the speed of development so automation becomes more important. Checklists are replaced by scripts that can execute the same tasks faster and more accurately than any human.

The environment to which continuous delivery delivers might be a test environment or, as is being done by many major technology companies, it could be the production environment. The latter requires an investment in high-quality tests that can give confidence that a change isn't going to break production for users. In the same way that continuous integration caught issues in the code early continuous delivery catches issues in the deployment process early.

The importance of automating the build and delivery process is accentuated by cloud-native applications. Deployments happen more frequently and to more environments so manually deploying borders on impossible.

Azure Builds

Azure DevOps provides a set of tools to make continuous integration and deployment easier than ever. These tools are located under Azure Pipelines. The first of them is Azure Builds, which is a tool for running YAML-based build definitions at scale. Users can either bring their own build machines (great for if the build requires a meticulously set up environment) or use a machine from a constantly refreshed pool of Azure hosted virtual machines. These hosted build agents come pre-installed with a wide range of development tools for not just .NET development but for everything from Java to Python to iPhone development.

DevOps includes a wide range of out of the box build definitions that can be customized for any build. The build definitions are defined in a file called `azure-pipelines.yml` and checked into the repository so they can be versioned along with the source code. This makes it much easier to make changes to the build pipeline in a branch as the changes can be checked into just that branch. An example `azure-pipelines.yml` for building an ASP.NET web application on full framework is show in Figure 11-8.

```
name: $(rev:r)

variables:
  version: 9.2.0.$(Build.BuildNumber)
  solution: Portals.sln
  artifactName: drop
  buildPlatform: any cpu
  buildConfiguration: release

pool:
  name: Hosted VS2017
  demands:
  - msbuild
  - visualstudio
  - vstest

steps:
- task: NuGetToolInstaller@0
  displayName: 'Use NuGet 4.4.1'
  inputs:
    versionSpec: 4.4.1

- task: NuGetCommand@2
  displayName: 'NuGet restore'
  inputs:
    restoreSolution: '$(solution)'

- task: VSBuild@1
  displayName: 'Build solution'
  inputs:
    solution: '$(solution)'
    msbuildArgs: '/p:DeployOnBuild=true /p:WebPublishMethod=Package
/p:PackageAsSingleFile=true /p:SkipInvalidConfigurations=true
/p:PackageLocation="$(build.artifactstagingdirectory)\\"'
    platform: '$(buildPlatform)'
```

```
    configuration: '$(buildConfiguration)'

- task: VSTest@2
  displayName: 'Test Assemblies'
  inputs:
    testAssemblyVer2: |
     **\$(buildConfiguration)\**\*test*.dll
     !**\obj\**
     !**\*testadapter.dll
    platform: '$(buildPlatform)'
    configuration: '$(buildConfiguration)'

- task: CopyFiles@2
  displayName: 'Copy UI Test Files to: $(build.artifactstagingdirectory)'
  inputs:
    SourceFolder: UITests
    TargetFolder: '$(build.artifactstagingdirectory)/uitests'

- task: PublishBuildArtifacts@1
  displayName: 'Publish Artifact'
  inputs:
    PathtoPublish: '$(build.artifactstagingdirectory)'
    ArtifactName: '$(artifactName)'
  condition: succeededOrFailed()
```

Figure 11-8 - A sample azure-pipelines.yml

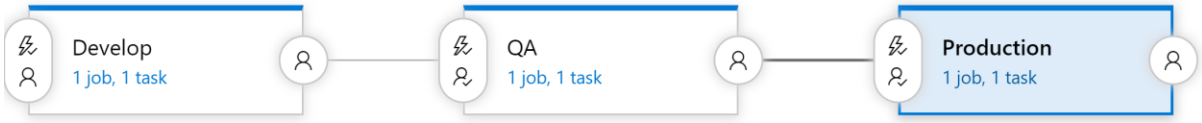
This build definition uses a number of built-in tasks that make creating builds as simple as building a Lego set (simpler than the giant Millennium Falcon). For instance, the NuGet task restores NuGet packages, while the VSBuild task calls the Visual Studio build tools to perform the actual compilation. There are hundreds of different tasks available in Azure DevOps, with thousands more that are maintained by the community. It's likely that no matter what build tasks you're looking to run, somebody has built one already.

Builds can be triggered manually, by a check-in, on a schedule, or by the completion of another build. In most cases, building on every check-in is desirable. Builds can be filtered so that different builds run against different parts of the repository or against different branches. This allows for scenarios like running fast builds with reduced testing on pull requests and running a full regression suite against the trunk on a nightly basis.

The end result of a build is a collection of files known as build artifacts. These artifacts can be passed along to the next step in the build process or added to an Azure Artifact feed, so they can be consumed by other builds.

Azure DevOps releases

Builds take care of compiling the software into a shippable package, but the artifacts still need to be pushed out to a testing environment to complete continuous delivery. For this, Azure DevOps uses a separate tool called Releases. Releases make use of the same tasks' library that were available to the Build but introduce a concept of "stages". A stage is an isolated environment into which the package is installed. For instance, a product might make use of a development, a QA, and a production environment. Code is continuously delivered into the development environment where automated tests can be run against it. Once those tests pass the release moves onto the QA environment for manual testing. Finally, the code is pushed to production where it's visible to everybody.



Each stage in the build can be automatically triggered by the completion of the previous phase. In many cases, however, this isn't desirable. Moving code into production might require approval from somebody. Releases supports this by allowing approvers at each step of the release pipeline. Rules can be set up such that a specific person or group of people must sign off on a release before it makes into production. These gates allow for manual quality checks and also for compliance with any regulatory requirements related to control what goes into production.

Everybody gets a build pipeline

There's no cost to configuring many build pipelines, so it's advantageous to have at least one build pipeline per microservice. Ideally, microservices are independently deployable to any environment so having each one able to be released via its own pipeline without releasing a mass of unrelated code is perfect. Each pipeline can have its own set of approvals allowing for variations in build process for each service.

Versioning releases

One drawback to using the Releases functionality is that it can't be defined in a checked-in `azure-pipelines.yml` file. There are many reasons you might want to do that from having per-branch release definitions to including a release skeleton in your project template. Fortunately, work is ongoing to shift some of the stages support into the Build component. This will be known as multi-stage build and the first version is available now!

Infrastructure as code

Cloud-native applications tend to make use of all sorts of fantastic platform as a service (PaaS) components. On a cloud platform like Azure, these components might include things like storage, Service Bus, and the SignalR service. As applications become more complicated, the number of these services in use is likely to grow. Just as how continuous delivery broke the traditional model of deploying to an environment manually, the rapid pace of change also broke the model of having a centralized IT group manage environments.

Building environments can, and should, also be automated. There's a wide range of well thought out tools that can make the process easy.

Azure Resource Manager templates

Azure Resource Manager templates are a JSON-based language for defining various resources in Azure. The basic schema looks something like Figure 11-10.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-
01/deploymentTemplate.json#",
  "contentVersion": "",
  "apiProfile": "",
  "parameters": {  },
  "variables": {  },
  "functions": [  ],
  "resources": [  ],
  "outputs": {  }
}
```

Figure 11-10 - The schema for a Resource Manager template

Within this template, one might define a storage container inside the resources section like so:

```
"resources": [
    {
      "type": "Microsoft.Storage/storageAccounts",
      "name": "[variables('storageAccountName')]",
      "location": "[parameters('location')]",
      "apiVersion": "2018-07-01",
      "sku": {
        "name": "[parameters('storageAccountType')]"
      },
      "kind": "StorageV2",
      "properties": {}
    }
  ],
```

Figure 11-11 - An example of a storage account defined in a Resource Manager template

The templates can be parameterized so that one template can be reused with different settings to define development, QA, and production environments. This helps eliminate surprises when migrating to a higher environment that is set up differently from the lower environments. The resources defined in a template are typically all created within a single resource group on Azure (it's possible to define multiple resource groups in a single Resource Manager template but unusual). This makes it very easy to delete an environment by just deleting the resource group as a whole. Cost analysis can also be run at the resource group level, allowing for quick accounting of how much each environment is costing.

There are many example templates defined in the Azure Quickstart Templates project on GitHub that will give a leg up when starting on a new template or adding to an existing one.

Resource Manager templates can be run in a variety of ways. Perhaps the simplest way is to simply paste them into the Azure portal. For experimental deployments, this method can be very quick. They can also be run as part of a build or release process in Azure DevOps. There are tasks that will leverage connections into Azure to run the templates. Changes to Resource Manager templates are applied incrementally, meaning that to add a new resource requires just adding it to the template. The tooling will handle diffing the current resource group with the desired resource group defined in the template. Resources will then be created or altered so they match what is defined in the template.

Terraform

A perceived disadvantage of Resource Manager templates is that they are specific to the Azure cloud. It's unusual to create applications that include resources from more than one cloud, but in cases

where the business relies on spectacular uptime, the cost of supporting multiple clouds might be worthwhile. If there were one templating language that could be used across every cloud, then it would also allow for developer skills to be much more portable.

Several technologies exist which do just that! The most mature offering in that space is known as Terraform. Terraform supports every major cloud player such as Azure, Google Cloud Platform, AWS, and AliCloud, and it also supports dozens of minor players such as Heroku and DigitalOcean. Instead of using JSON as the template definition language, it uses the slightly more terse YAML.

An example Terraform file that does the same as the previous Resource Manager template (Figure 11-11) is shown in Figure 11-12:

```
provider "azurerm" {
  version = "=1.28.0"
}

resource "azurerm_resource_group" "test" {
  name     = "production"
  location = "West US"
}

resource "azurerm_storage_account" "testsa" {
  name                     = "${var.storageAccountName}"
  resource_group_name      = "${azurerm_resource_group.testrg.name}"
  location                 = "${var.region}"
  account_tier             = "${var.tier}"
  account_replication_type = "${var.replicationType}"

}
```

Figure 11-12 - An example of a Resource Manager template

Terraform does a better job of providing sensible error messages when a resource can't be deployed because of an error in the template. This is an area where Resource Manager templates have some ongoing challenges. There's also a very handy validate task that can be used in the build phase to catch template errors early.

As with Resource Manager templates, there are command-line tools that can be used to deploy Terraform templates. There are also community-created tasks in Azure Pipelines that can validate and apply Terraform templates.

In the event that the Terraform or Resource Manager template outputs interesting values such as the connection string to a newly created database they can be captured in the build pipeline and used in subsequent tasks.

Cloud Native Application Bundles

A key property of cloud-native applications is that they leverage the properties of the cloud to speed up development. This often means that a full application uses different kinds of technologies. Applications may be shipped in Docker containers, some services may use Azure Functions while other parts may run directly on virtual machines allocated on large metal servers with hardware GPU

acceleration. No two cloud-native applications are the same, so it's been difficult to provide a single mechanism for shipping them.

The Docker containers may run on Kubernetes using a Helm Chart for deployment. The Azure Functions may be allocated using Terraform templates. Finally, the virtual machines may be allocated using Terraform but built out using Ansible. This is a whole mess of technologies and there has been no way to package them all together into a reasonable package. Until now.

Cloud Native Application Bundles (CNABs) are a joint effort of a number of community-minded companies such as Microsoft, Docker, and HashiCorp to develop a specification to package distributed applications.

The effort was announced in December of 2018, so there's still a fair bit of work to do to expose the effort to the greater community. However, there's already an open specification and a reference implementation known as Duffle. This tool, which was written in Go, is a joint effort between Docker and Microsoft.

The CNABs can contain different kinds of installation technologies. This allows things like Helm Charts, Terraform templates, and Ansible Playbooks to coexist in the same package. Once built, the packages are self-contained and portable; they can be installed from a USB stick. The packages are cryptographically signed to ensure they originate from the party they claim.

The core of a CNAB is a file called `bundle.json`. This file defines the contents of the bundle, be they Terraform or images or anything else. Figure 11-9 defines a CNAB that invokes some Terraform. Notice, however, that it actually defines an invocation image that is used to invoke the Terraform. When packaged up, the Docker file that is located in the *cnab* directory is built into a Docker image, which will be included in the bundle. Having Terraform installed inside a Docker container in the bundle means that users don't need to have Terraform installed on their machine to run the bundling.

```
{
    "name": "terraform",
    "version": "0.1.0",
    "schemaVersion": "v1.0.0-WD",
    "parameters": {
        "backend": {
            "type": "boolean",
            "defaultValue": false,
            "destination": {
                "env": "TF_VAR_backend"
            }
        }
    },
    "invocationImages": [
        {
        "imageType": "docker",
        "image": "cnab/terraform:latest"
        }
    ],
    "credentials": {
        "tenant_id": {
            "env": "TF_VAR_tenant_id"
        },
        "client_id": {
            "env": "TF_VAR_client_id"
```

```
        },
        "client_secret": {
            "env": "TF_VAR_client_secret"
        },
        "subscription_id": {
            "env": "TF_VAR_subscription_id"
        },
        "ssh_authorized_key": {
            "env": "TF_VAR_ssh_authorized_key"
        }
    },
    "actions": {
        "status": {
            "modifies": true
        }
    }
}
```

Figure 11-13 - An example Terraform file

The `bundle.json` also defines a set of parameters that are passed down into the Terraform. Parameterization of the bundle allows for installation in a variety of different environments.

The CNAB format is also flexible, allowing it to be used against any cloud. It can even be used against on-premises solutions such as OpenStack.

DevOps Decisions

There are so many great tools in the DevOps space these days and even more fantastic books and papers on how to succeed. A favorite book to get started on the DevOps journey is The Phoenix Project, which follows the transformation of a fictional company from NoOps to DevOps. One thing is for certain: DevOps is no longer a "nice to have" when deploying complex, Cloud Native Applications. It's a requirement and should be planned for and resourced at the start of any project.

Important

PREVIEW EDITION

This book is a preview edition and is currently under construction. If you have any feedback, submit it at https://aka.ms/ebookfeedback.