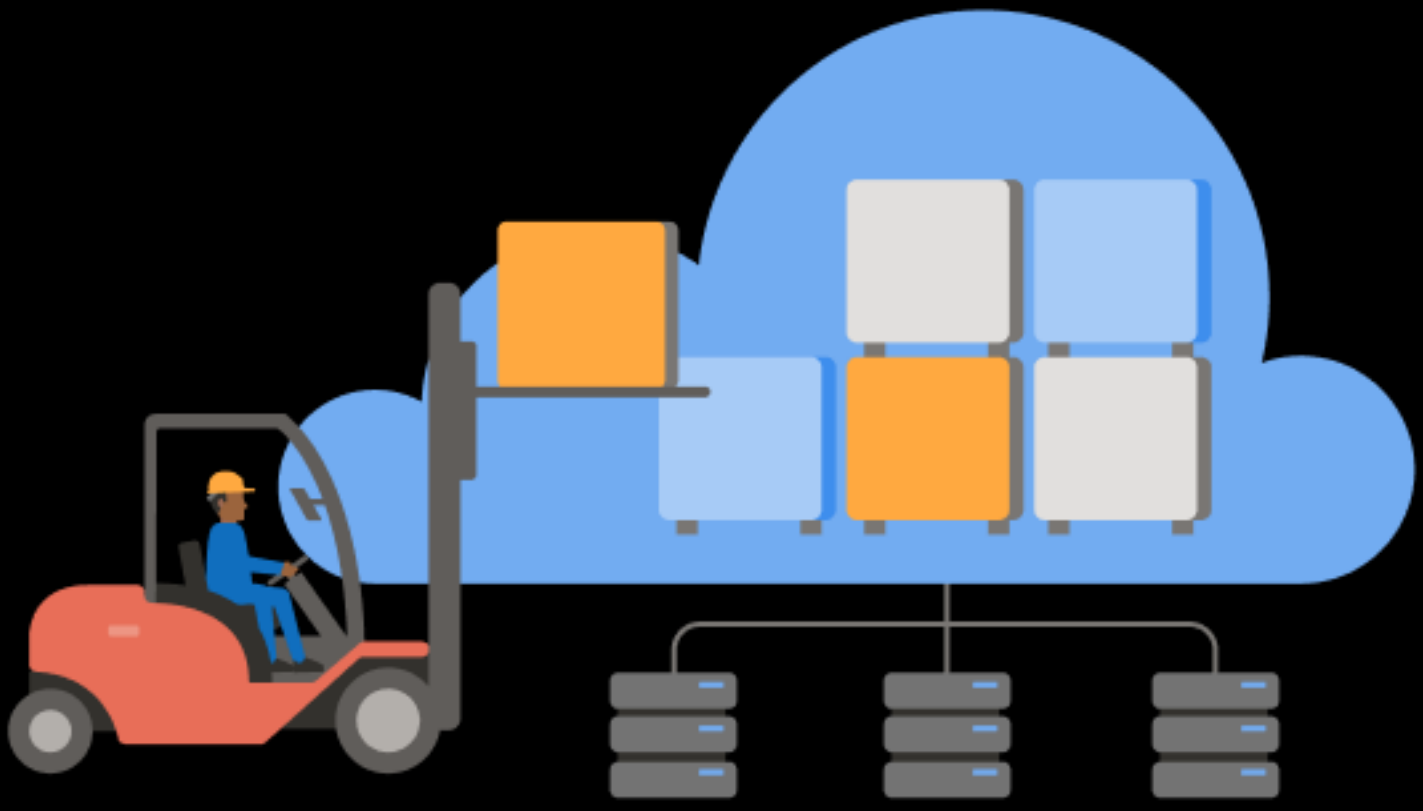
Microsoft

Architecting Cloud-Native .NET Apps for Azure



Robert Vettor

Steve "ardalis" Smith

Version 1.0

EDITION v1.0.2

Refer changelog for the book updates and community contributions.

PUBLISHED BY

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

A division of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Authors:

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

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

Participants and Reviewers:

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

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

Jeremy Likness, Senior Program Manager, .NET team, Microsoft

Cecil Phillip, Senior Cloud Advocate, Microsoft

Sumit Ghosh, Principal Consultant at Neudesic

Editors:

Maira Wenzel, Program Manager, .NET team, Microsoft

David Pine, Senior Content Developer, .NET docs, Microsoft

Version

This guide has been written to cover **.NET 5** version along with many additional updates related to the same "wave" of technologies (that is, Azure and additional third-party technologies) coinciding in time with the .NET 5 release.

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.

Send your feedback

This book and related samples are constantly evolving, so your feedback is welcomed! If you have comments about how this book can be improved, use the feedback section at the bottom of any page built on GitHub issues.

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 compete with the 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.

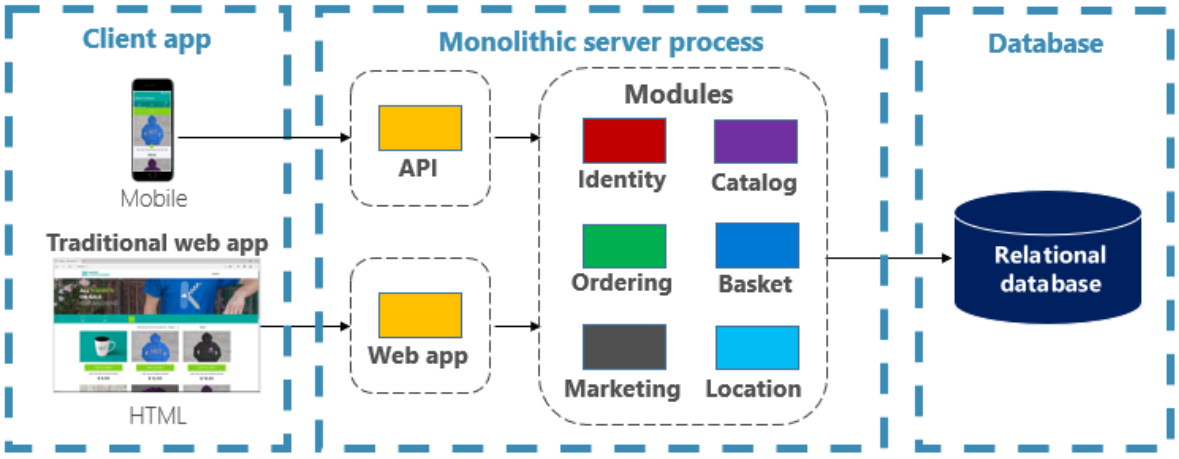


Figure 1-1. Traditional monolithic design

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 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 possible and 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.

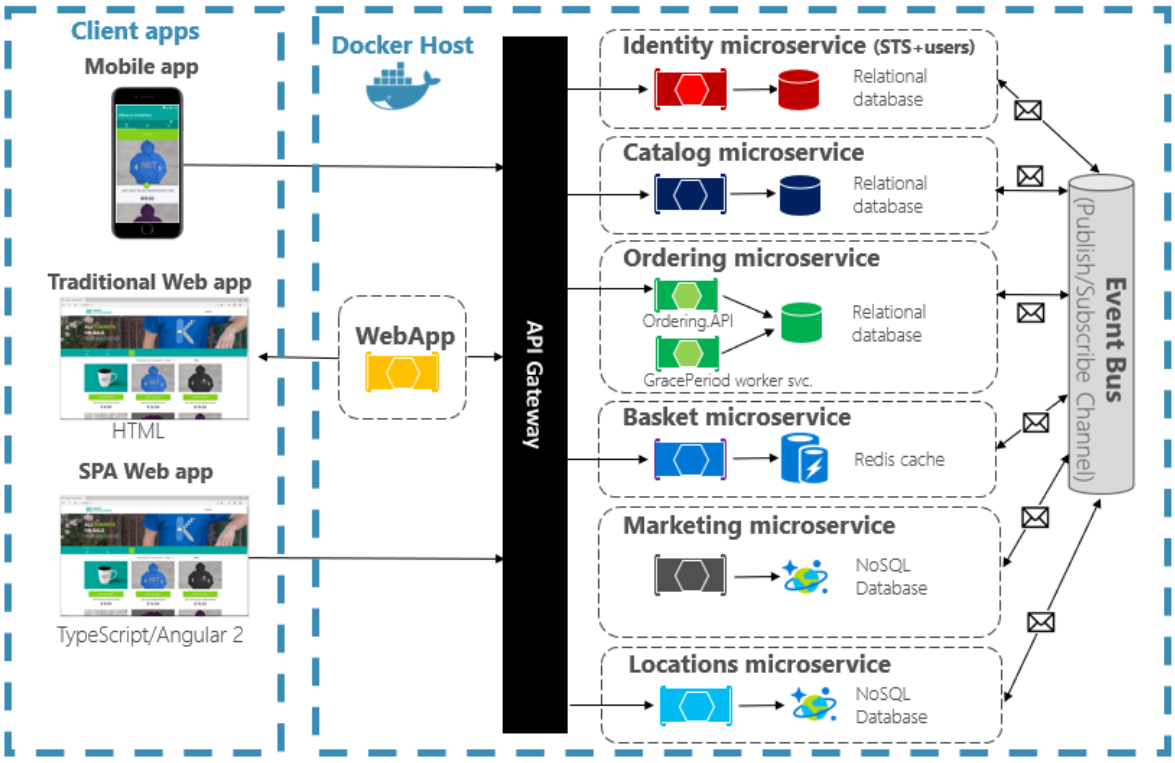


Figure 1-2. Cloud-native design

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 directing traffic to the core back-end services and enforcing many cross-cutting concerns. Most importantly, the application takes full advantage of the scalability, availability, and resiliency features found in modern cloud platforms.

Cloud-native computing

Hmm... We just used the term, *Cloud Native*. Your 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, this book presents 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 ten of your colleagues. Ask them to define the term "Cloud Native". Good chance you'll get ten different answers.

Cloud native is all about changing the way you 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 about *speed* and *agility*. Business systems are evolving from enabling business capabilities to being weapons of strategic transformation that accelerate 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 hundred times per day.
Uber	Has 1,000+ services in production. Deploys several thousand times each week.
WeChat	Has 3,000+ services in production. Deploys 1,000 times a 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.

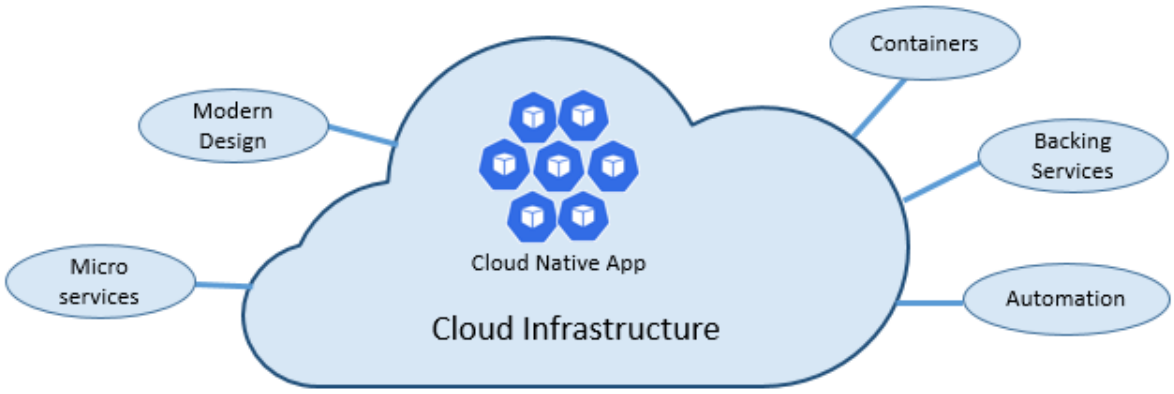


Figure 1-3. Cloud-native foundational pillars

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 Twelve-Factor 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, he discusses 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 out-of-process network communication. In a distributed architecture, what happens when Service B isn't responding to a network call from Service A? Or, what happens when Service C becomes temporarily unavailable and other services calling it are blocked?

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.

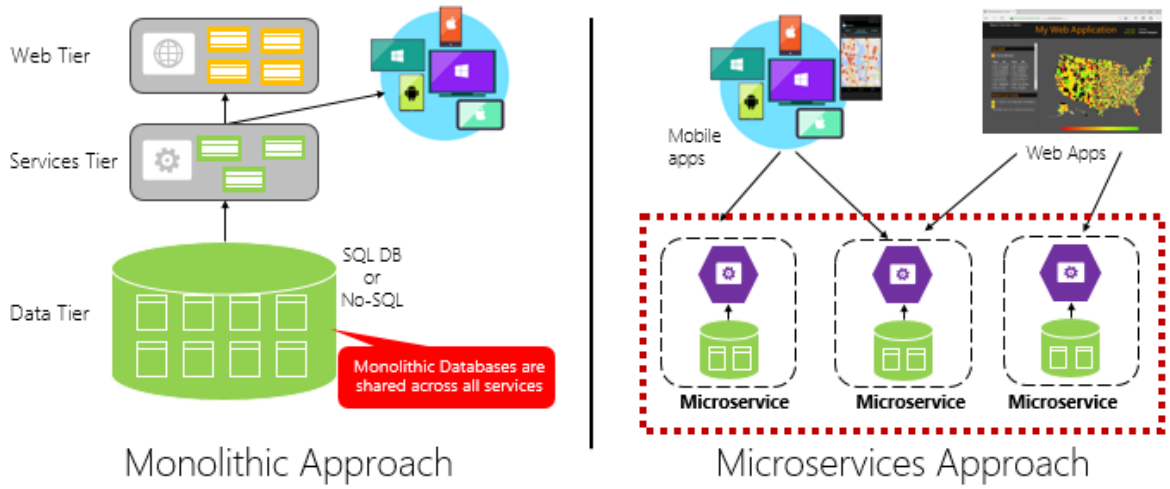


Figure 1-4. Monolithic deployment versus microservices

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

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

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

Containers

Nowadays, it's 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.

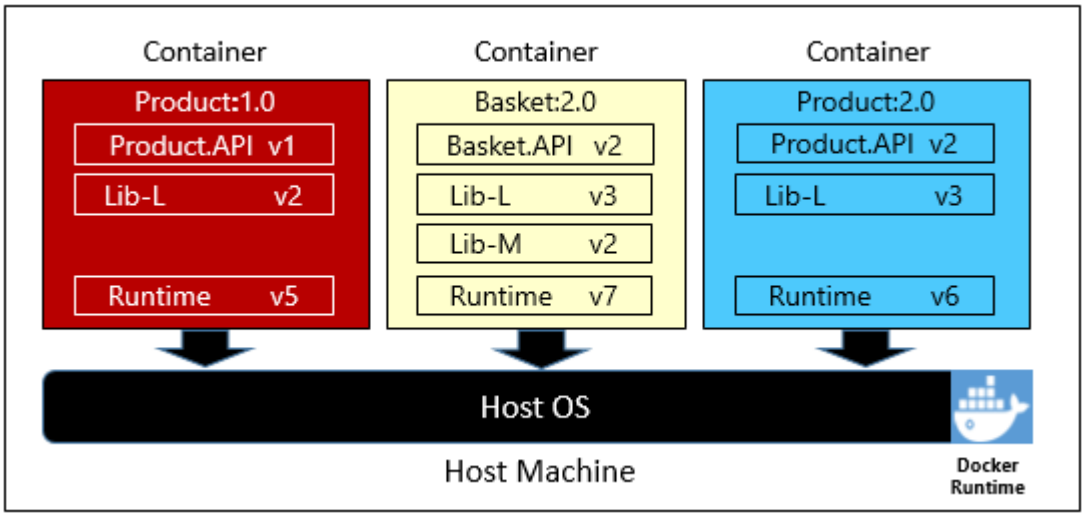


Figure 1-5. Multiple containers running on a container host

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.

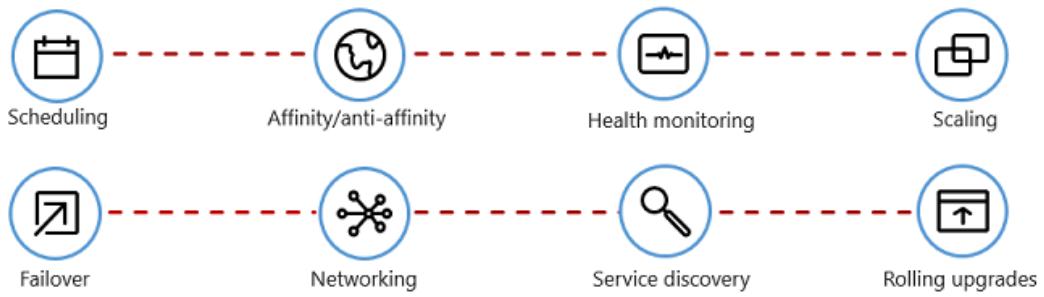


Figure 1-6. What container orchestrators do

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.

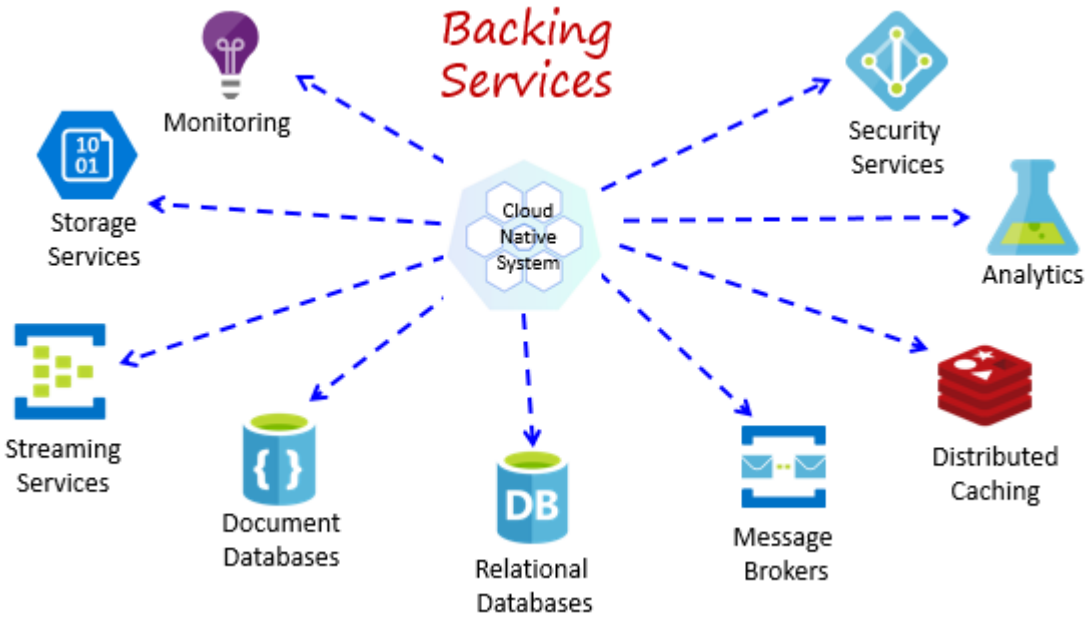


Figure 1-7. Common backing services

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.

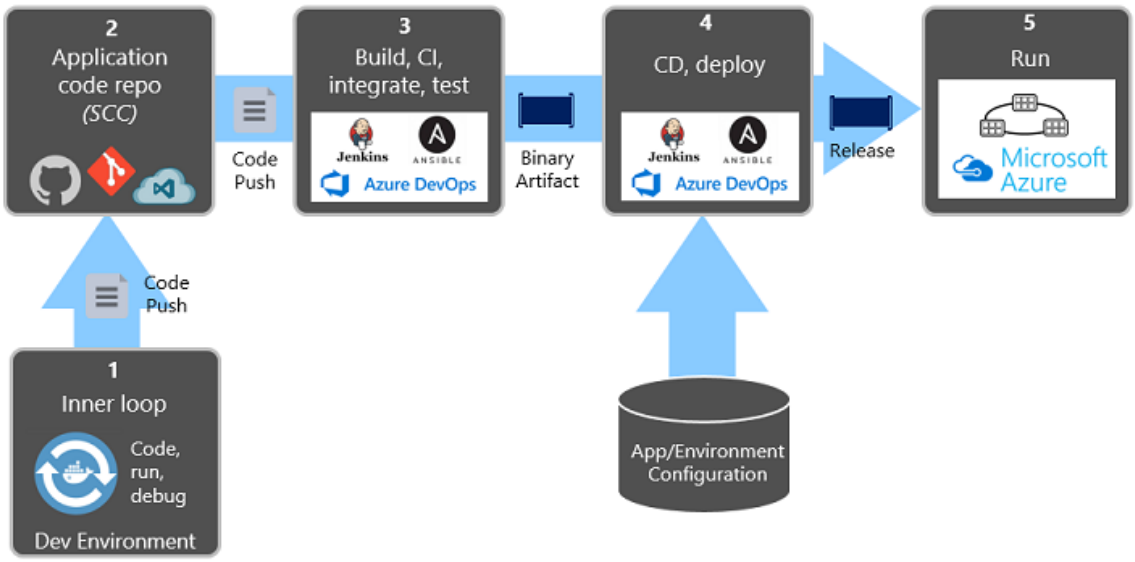


Figure 1-8. Deployment steps in a CI/CD Pipeline

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?

- 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.

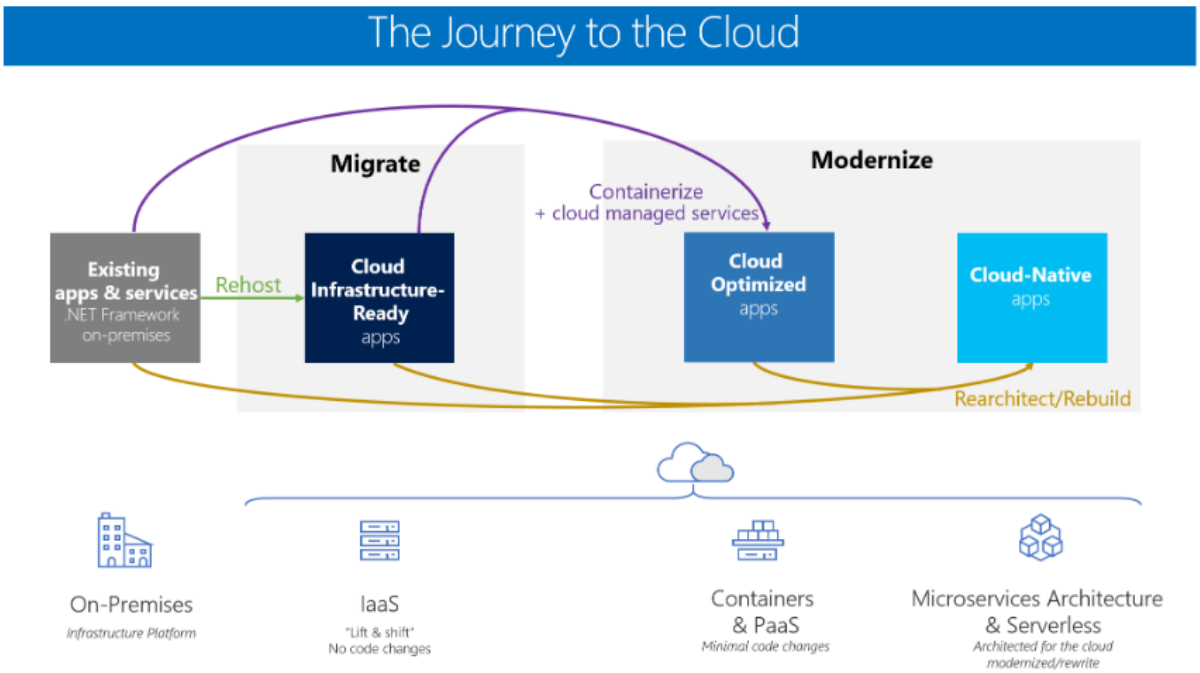


Figure 1-10. Strategies for migrating legacy workloads

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 at 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

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 and Docker, and optionally Azure, Kubernetes, and Visual Studio, to build an online storefront.

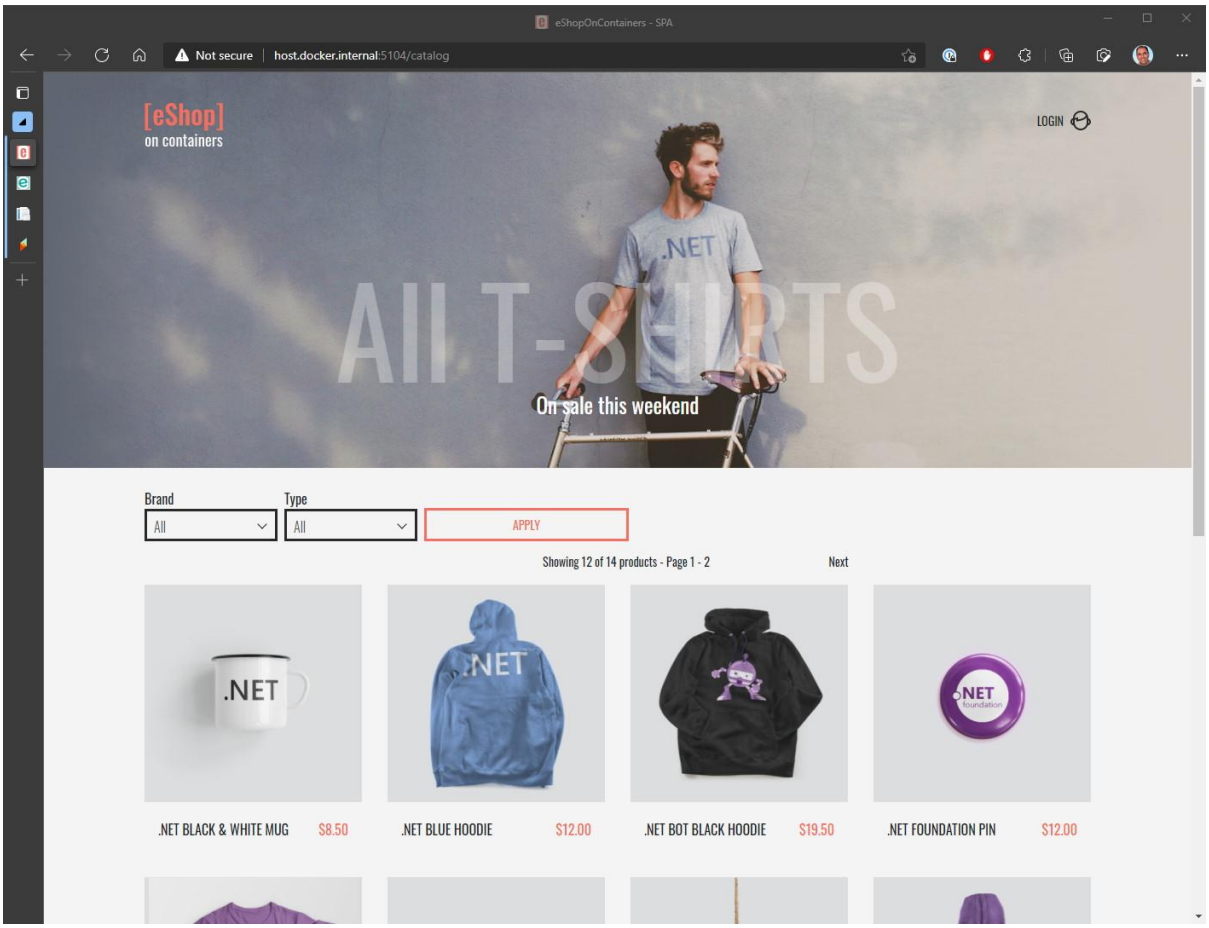


Figure 2-1. eShopOnContainers Sample App Screenshot.

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.

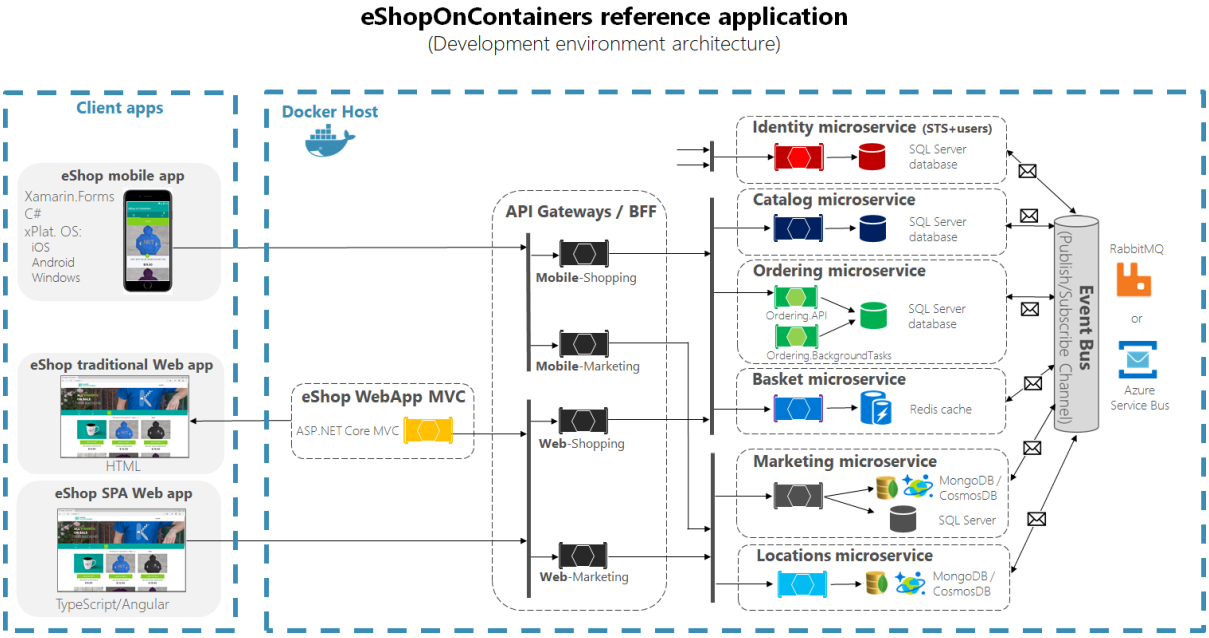


Figure 2-2. eShopOnContainers reference application development architecture.

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 many 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. 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 by using a message bus.

Each of the different microservices is designed differently, based on their individual requirements. This aspect means their technology stack may differ, although they're all built using .NET 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.

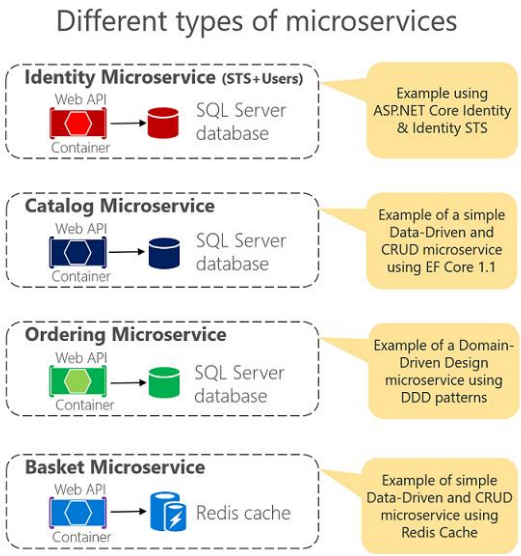


Figure 2-3. Different kinds of microservices.

Overview of the code

Because it uses 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.

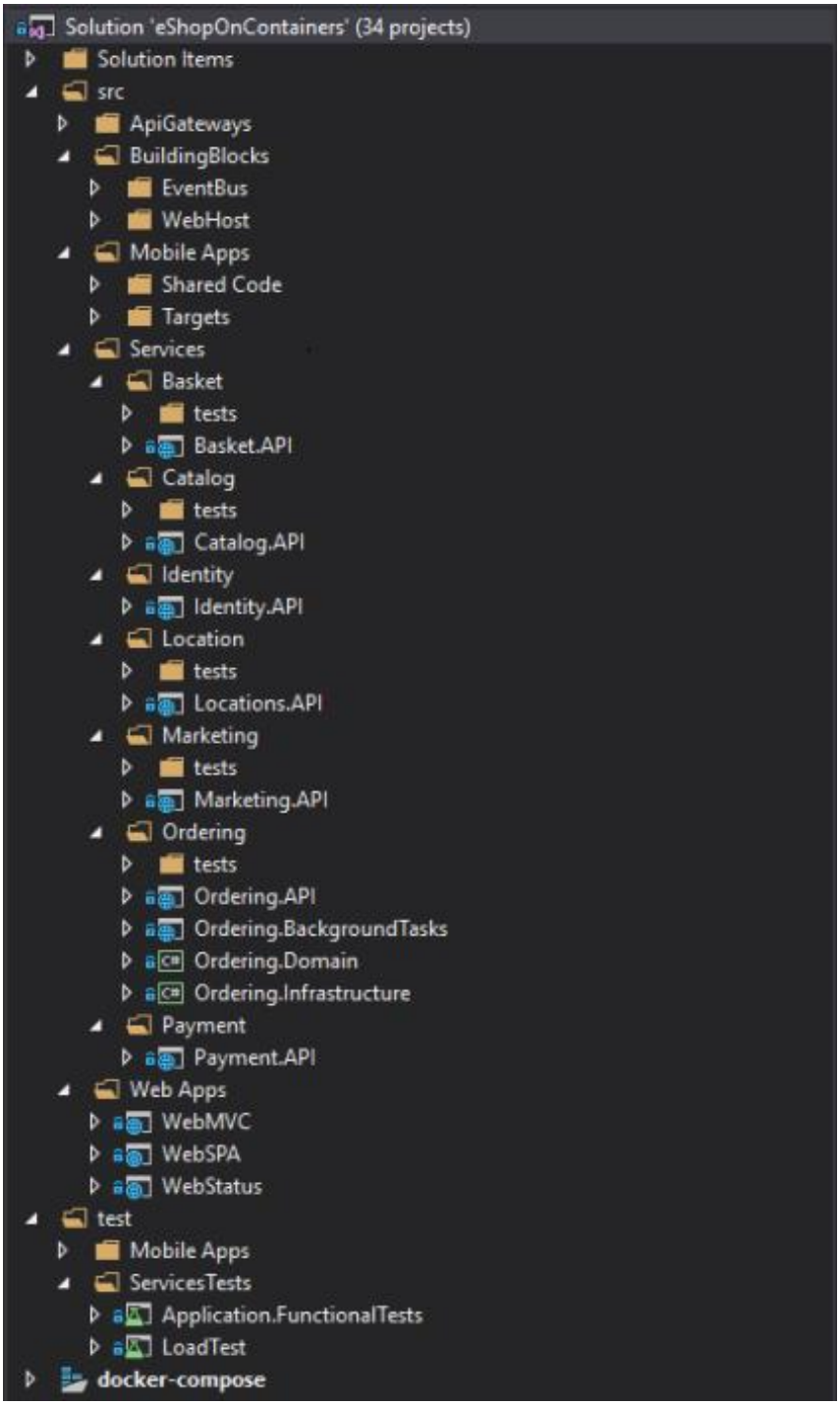


Figure 2-4. Projects in Visual Studio solution.

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, and 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.

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, 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.

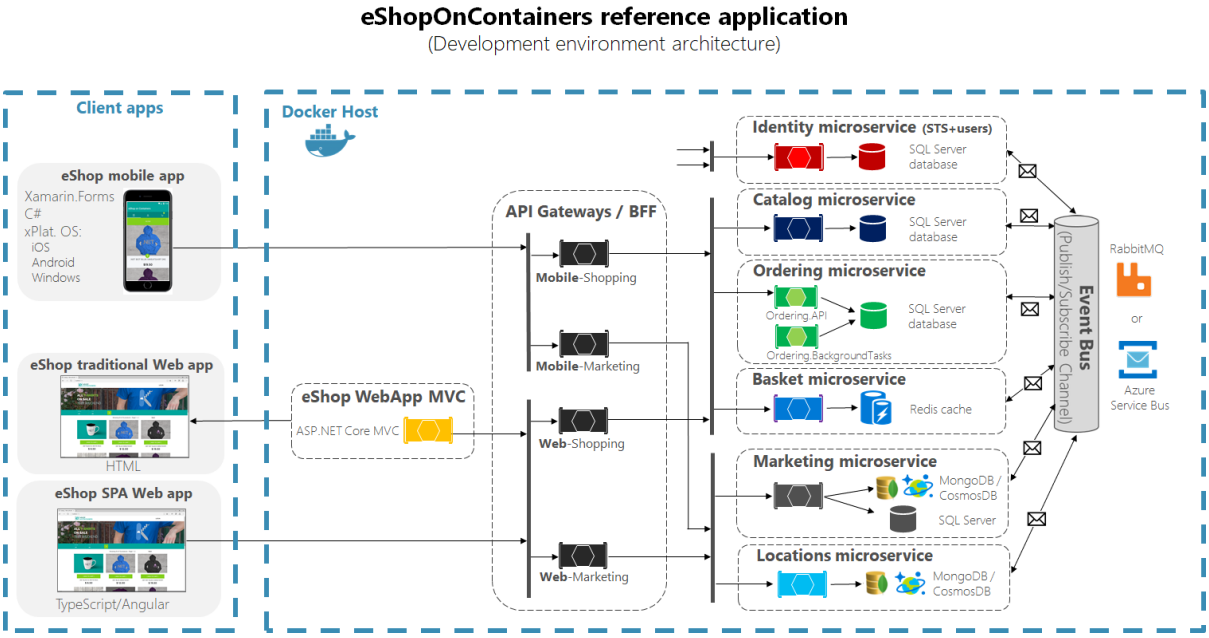


Figure 2-5. The eShopOnContainers Architecture.

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 containers for each microservice in the AKS cluster, as shown in the architecture diagram above. This approach allows each individual service to scale 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 approach 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 aspect 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 various 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.

Deploying eShopOnContainers to Azure

The eShopOnContainers application can be deployed to various Azure platforms. The recommended approach is to deploy the application to Azure Kubernetes Services (AKS). Helm, a Kubernetes deployment tool, is available to reduce deployment complexity. Optionally, developers may implement Azure Dev Spaces for Kubernetes to streamline their development process.

Azure Kubernetes Service

To host eShop in AKS, the first step is to create an AKS cluster. To do so, you might use the Azure portal, which will walk you through the required steps. You could also create a cluster from the Azure CLI, taking care to enable Role-Based Access Control (RBAC) and application routing. The eShopOnContainers' documentation details the steps for creating your own AKS cluster. Once created, you can access and manage the cluster from the Kubernetes dashboard.

You can now deploy the eShop application to the cluster using Helm.

Deploying to Azure Kubernetes Service using Helm

Helm is an application package manager tool that works directly with Kubernetes. It helps you define, install, and upgrade Kubernetes applications. While simple apps can be deployed to AKS with custom CLI scripts or simple deployment files, complex apps can contain many Kubernetes objects and benefit from Helm.

Using Helm, applications include text-based configuration files, called Helm charts, which declaratively describe the application and configuration in Helm packages. Charts use standard YAML-formatted

files to describe a related set of Kubernetes resources. They're versioned alongside the application code they describe. Helm Charts range from simple to complex depending on the requirements of the installation they describe.

Helm is composed of a command-line client tool, which consumes helm charts and launches commands to a server component named, Tiller. Tiller communicates with the Kubernetes API to ensure the correct provisioning of your containerized workloads. Helm is maintained by the Cloud-native Computing Foundation.

The following yaml file presents a Helm template:

```
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.app.svc.marketing }}
  labels:
    app: {{ template "marketing-api.name" . }}
    chart: {{ template "marketing-api.chart" . }}
    release: {{ .Release.Name }}
    heritage: {{ .Release.Service }}
spec:
  type: {{ .Values.service.type }}
  ports:
    - port: {{ .Values.service.port }}
      targetPort: http
      protocol: TCP
      name: http
  selector:
    app: {{ template "marketing-api.name" . }}
    release: {{ .Release.Name }}
```

Note how the template describes a dynamic set of key/value pairs. When the template is invoked, values that enclosed in curly braces are pulled in from other yaml-based configuration files.

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.

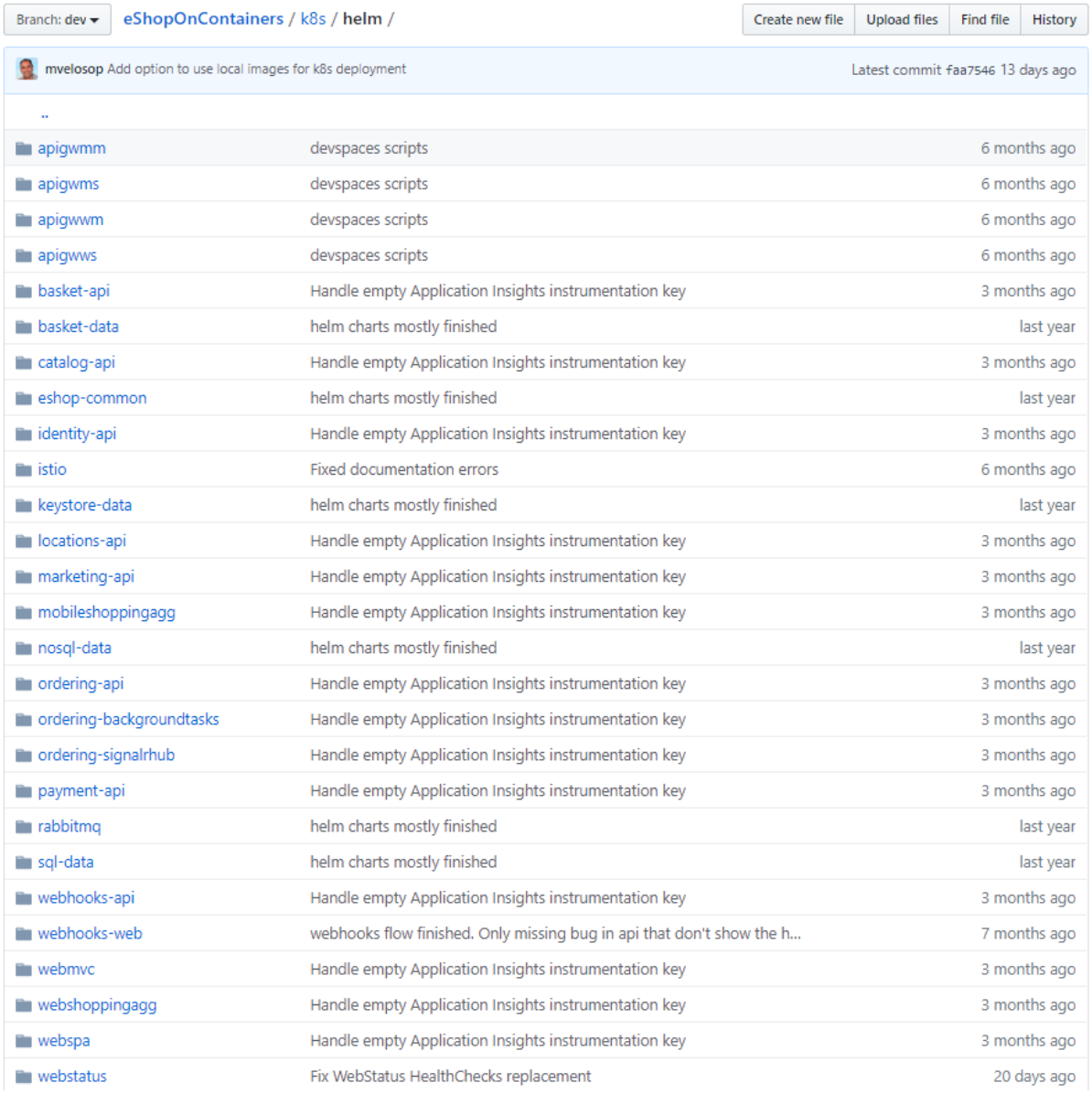


Figure 2-6. The eShopOnContainers helm folder.

Each individual component is installed using a `helm install` command. eShop includes a "deploy all" script that loops through and installs the components 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.

Note that version 3 of Helm officially removes the need for the Tiller server component. More information on this enhancement can be found here.

Azure Dev Spaces

Cloud-native applications can quickly grow large and complex, requiring significant compute resources to run. In these scenarios, the entire application can't be hosted on a development machine

(especially a laptop). Azure Dev Spaces is designed to address this problem using AKS. It enables developers to work with a local version of their services while hosting the rest of the application in an AKS development cluster.

Developers share a running (development) instance in an AKS cluster that contains the entire containerized application. But they use personal spaces set up on their machine to locally develop their services. When ready, they test from end-to-end in the AKS cluster - without replicating dependencies. Azure Dev Spaces merges code from the local machine with services in AKS. Team members can see how their changes will behave in a real AKS environment. Developers can rapidly iterate and debug code directly in Kubernetes using Visual Studio 2017 or Visual Studio Code.

In Figure 2-7, you can see that Developer Susie has deployed an updated 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).

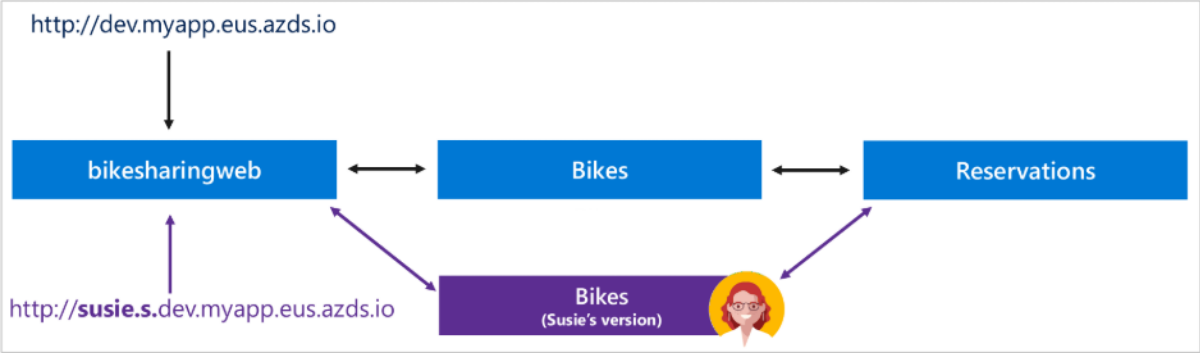


Figure 2-7. Developer Susie deploys her own version of the Bikes microservice and tests it.

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

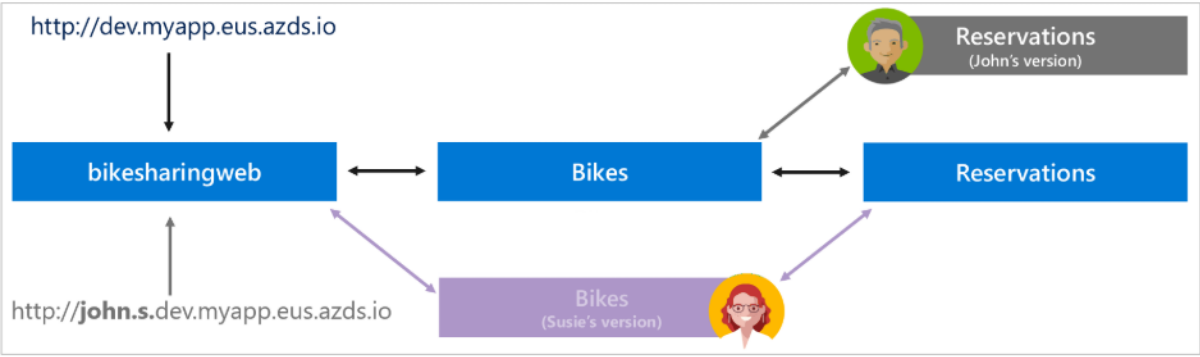


Figure 2-8. Developer John deploys his own version of the Reservations microservice and tests it without conflicting with other developers.

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 track marketing campaign details for a given campaign ID. Rather than creating a full microservice, a single Azure Function is simpler and sufficient. Azure Functions have a simple build and deployment model, especially when configured to run in Kubernetes. Deploying the function is scripted using Azure Resource Manager (ARM) templates and the Azure CLI. This campaign service isn't customer-facing and invokes a single operation, making it a great candidate for Azure Functions. The function requires minimal configuration, including a database connection string data and image base URI settings. You configure Azure Functions in the Azure portal.

Centralized configuration

Unlike a monolithic app in which everything runs within a single instance, a cloud-native application consists of independent services distributed across virtual machines, containers, and geographic regions. Managing configuration settings for dozens of interdependent services can be challenging. Duplicate copies of configuration settings across different locations are error prone and difficult to manage. Centralized configuration is a critical requirement for distributed cloud-native applications.

As discussed in Chapter 1, the Twelve-Factor App recommendations require strict separation between code and configuration. Configuration must be stored externally from the application and read-in as needed. Storing configuration values as constants or literal values in code is a violation. The same configuration values are often be used by many services in the same application. Additionally, we must support the same values across multiple environments, such as dev, testing, and production. The best practice is store them in a centralized configuration store.

The Azure cloud presents several great options.

Azure App Configuration

Azure App Configuration is a fully managed Azure service that stores non-secret configuration settings in a secure, centralized location. Stored values can be shared among multiple services and applications.

The service is simple to use and provides several benefits:

- Flexible key/value representations and mappings
- Tagging with Azure labels
- Dedicated UI for management
- Encryption of sensitive information
- Querying and batch retrieval

Azure App Configuration maintains changes made to key-value settings for seven days. The point-in-time snapshot feature enables you to reconstruct the history of a setting and even rollback for a failed deployment.

App Configuration automatically caches each setting to avoid excessive calls to the configuration store. The refresh operation waits until the cached value of a setting expires to update that setting, even when its value changes in the configuration store. The default cache expiration time is 30 seconds. You can override the expiration time.

App Configuration encrypts all configuration values in transit and at rest. Key names and labels are used as indexes for retrieving configuration data and aren't encrypted.

Although App Configuration provides hardened security, Azure Key Vault is still the best place for storing application secrets. Key Vault provides hardware-level encryption, granular access policies, and management operations such as certificate rotation. You can create App Configuration values that reference secrets stored in a Key Vault.

Azure Key Vault

Key Vault is a managed service for securely storing and accessing secrets. A secret is anything that you want to tightly control access to, such as API keys, passwords, or certificates. A vault is a logical group of secrets.

Key Vault greatly reduces the chances that secrets may be accidentally leaked. When using Key Vault, application developers no longer need to store security information in their application. This practice eliminates the need to store this information inside your code. For example, an application may need to connect to a database. Instead of storing the connection string in the app's code, you can store it securely in Key Vault.

Your applications can securely access the information they need by using URIs. These URIs allow the applications to retrieve specific versions of a secret. There's no need to write custom code to protect any of the secret information stored in Key Vault.

Access to Key Vault requires proper caller authentication and authorization. Typically, each cloud-native microservice uses a ClientId/ClientSecret combination. It's important to keep these credentials outside source control. A best practice is to set them in the application's environment. Direct access to Key Vault from AKS can be achieved using Key Vault FlexVolume.

Configuration in eShop

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. Injecting secrets in environment variables 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.

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

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 compromising performance for each user. It's most often achieved by breaking up an application into small pieces that can each be given whatever resources they require. Cloud vendors enable massive scalability anytime and anywhere in the world.

In this chapter, we discuss 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.

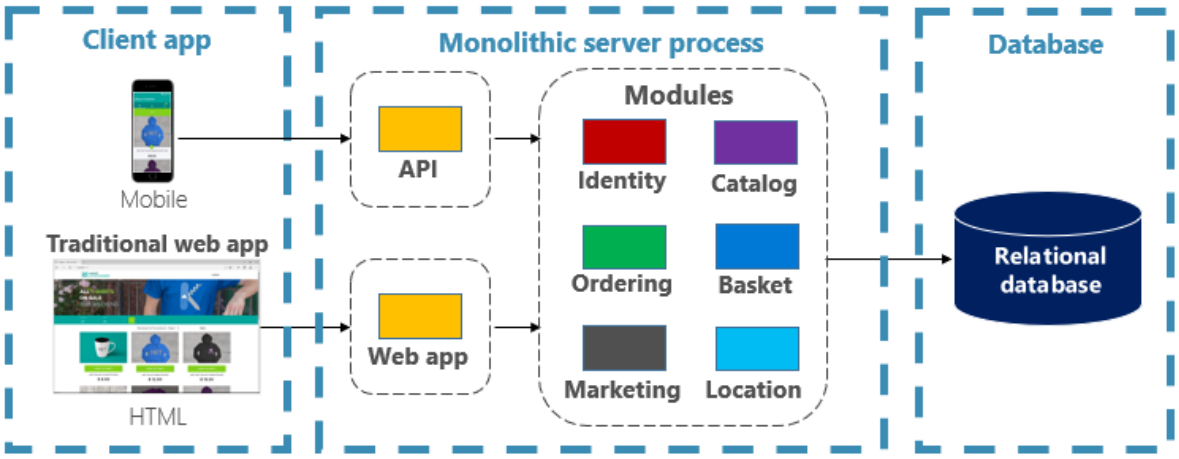


Figure 3-1. Monolithic architecture.

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

Deployment

Additionally, they require a restart of the application, which may temporarily impact availability if zero-downtime techniques are not applied while deploying.

Scaling

A monolithic application is hosted entirely on a single machine instance, often requiring high-capability hardware. If any part of the monolith requires scaling, another copy of the entire application must be deployed to another machine. With a monolith, you can't scale application components individually - it's all or nothing. Scaling components that don't require scaling results in inefficient and costly resource usage.

Environment

Monolithic applications are typically deployed to a hosting environment with a pre-installed operating system, runtime, and library dependencies. This environment may not match that upon which the application was developed or tested. Inconsistencies across application environments are a common source of problems for monolithic deployments.

Coupling

A monolithic application is likely to experience high coupling across its functional components. Without hard boundaries, system changes often result in unintended and costly side effects. New features/fixes become tricky, time-consuming, and expensive to implement. Updates require extensive testing. Coupling also makes it difficult to refactor components or swap in alternative implementations. Even when constructed with a strict separation of concerns, architectural erosion sets in as the monolithic code base deteriorates with never-ending "special cases."

Platform lock-in

A monolithic application is constructed with a single technology stack. While offering uniformity, this commitment can become a barrier to innovation. New features and components will be built using the application's current stack - even when more modern technologies may be a better choice. A longer-term risk is your technology stack becoming outdated and obsolete. Rearchitecting an entire application to a new, more modern platform is at best expensive and risky.

What are the benefits of containers and orchestrators?

We introduced containers in Chapter 1. We highlighted how the Cloud Native Computing Foundation (CNCF) ranks containerization as the first step in their Cloud-Native Trail Map - guidance for enterprises beginning their cloud-native journey. In this section, we discuss the benefits of containers.

Docker is the most popular container management platform. It works with containers on both Linux or Windows. Containers provide separate but reproducible application environments that run the same way on any system. This aspect makes them perfect for developing and hosting cloud-native services. Containers are isolated from one another. Two containers on the same host hardware can have different versions of software, without causing conflicts.

Containers are defined by simple text-based files that become project artifacts and are checked into source control. While full servers and virtual machines require manual effort to update, containers are easily version-controlled. 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 define a container, you can recreate and run it exactly the same way. This immutability lends itself to component-based design. If some parts of an application evolve differently than 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.

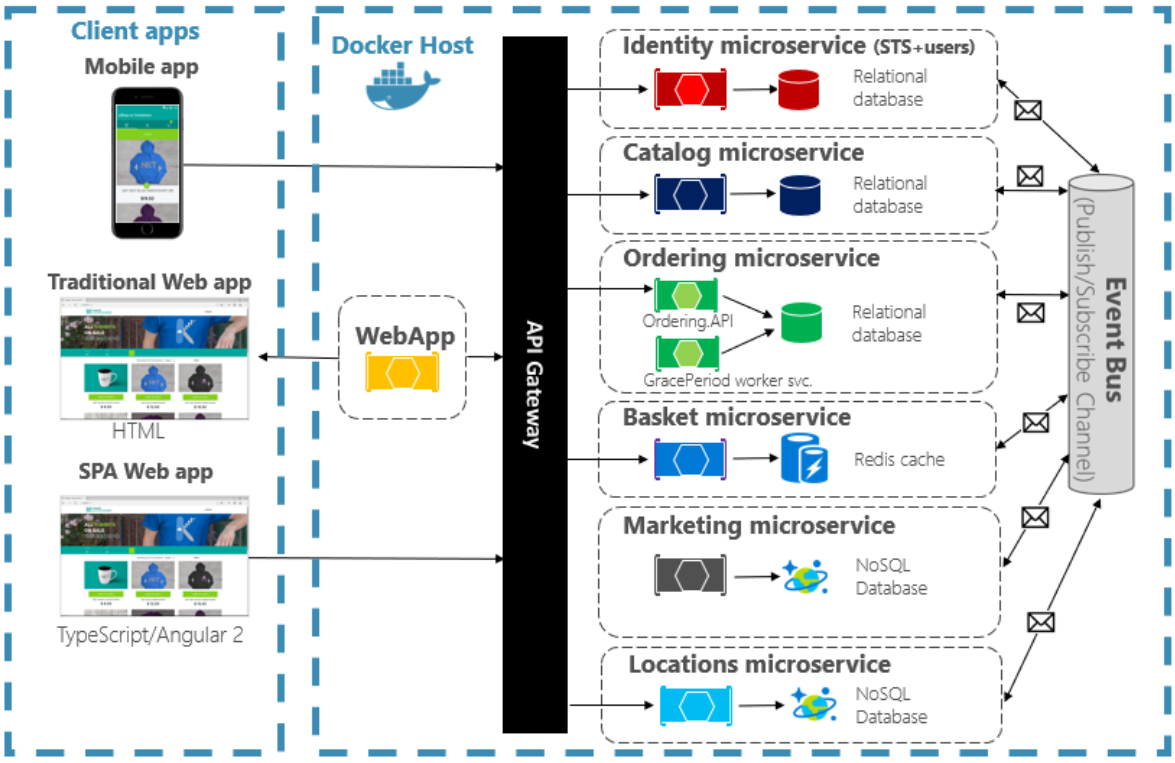


Figure 3-2. Decomposing a monolithic app to embrace microservices.

Each cloud-native service is built and deployed in a separate container. Each can update as needed. Individual services can be hosted on nodes with resources appropriate to each service. The environment each service runs in is immutable, shared across dev, test, and production environments, and easily versioned. Coupling between different areas of the application occurs explicitly as calls or messages between services, not compile-time dependencies within the monolith. You can also choose the technology that best suites a given capability without requiring changes to the rest of the app.

Containerized services require automated management. It wouldn't be feasible to manually administer a large set of independently deployed containers. For example, consider the following tasks:

- How will container instances be provisioned across a cluster of many machines?
- Once deployed, how will containers discover and communicate with each other?
- How can containers scale in or out on-demand?
- How do you monitor the health of each container?
- How do you protect a container against hardware and software failures?
- How do upgrade containers for a live application with zero downtime?

Container orchestrators address and automate these and other concerns.

In the cloud-native eco-system, Kubernetes has become the de facto container orchestrator. It's an open-source platform managed by the Cloud Native Computing Foundation (CNCF). Kubernetes automates the deployment, scaling, and operational concerns of containerized workloads across a machine cluster. However, installing and managing Kubernetes is notoriously complex.

A much better approach is to leverage Kubernetes as a managed service from a cloud vendor. The Azure cloud features a fully managed Kubernetes platform entitled Azure Kubernetes Service (AKS). AKS abstracts the complexity and operational overhead of managing Kubernetes. You consume Kubernetes as a cloud service; Microsoft takes responsibility for managing and supporting it. AKS also tightly integrates with other Azure services and dev tools.

AKS is a cluster-based technology. A pool of federated virtual machines, or nodes, is deployed to the Azure cloud. Together they form a highly available environment, or cluster. The cluster appears as a seamless, single entity to your cloud-native application. Under the hood, AKS deploys your containerized services across these nodes following a predefined strategy that evenly distributes the load.

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 have multiple containers that need to work together, you should 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 creates an abstraction layer over groups of containers and organizes them into *pods*. Pods run on worker machines referred to as *nodes*. This organized structure is referred to as a *cluster*. Figure 3-3 shows the different components of a Kubernetes cluster.

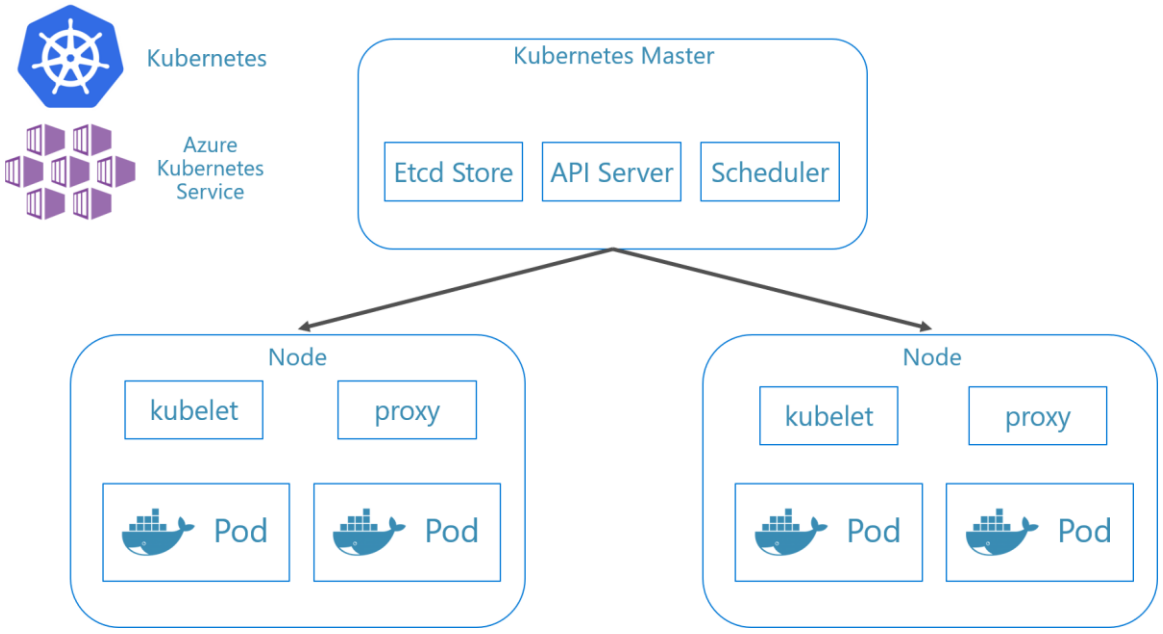


Figure 3-3. Kubernetes cluster components.

Scaling containerized workloads is a key feature of container orchestrators. AKS supports automatic scaling across two dimensions: Container instances and compute nodes. Together they give AKS the ability to quickly and efficiently respond to spikes in demand and add additional resources. We discuss scaling in AKS later in this chapter.

Declarative versus imperative

Kubernetes supports both declarative and imperative 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 create a configuration file, called a manifest, to describe what you want instead of what to do. Kubernetes reads the manifest and transforms your desired end state into actual end state.

Imperative commands are great for learning and interactive experimentation. However, you'll want to declaratively create Kubernetes manifest files to embrace an infrastructure as code approach, providing for reliable and repeatable deployments. The manifest file becomes a project artifact and is used in your CI/CD pipeline for automating Kubernetes deployments.

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 command produces a manifest similar to one shown below:

```
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.

You can also use declarative configuration with other Kubernetes features, one of which being deployments. Declarative deployments help manage releases, updates, and scaling. They instruct the Kubernetes deployment controller on how to deploy new changes, scale out load, or roll back to a previous revision. If a cluster is unstable, a declarative deployment will automatically return the cluster back to a desired state. For example, if a node should crash, the deployment mechanism will redeploy a replacement to achieve your desired state

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

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. They can be developed in containers, tested across versioned environments, and deployed into 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 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. Beyond a 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 unable to build your application following the Twelve-Factor App principles, you should consider avoiding containers and orchestrators. In these cases, consider a VM-based hosting platform, or possibly some hybrid system. With it, you can always 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 can also run locally on your development machine. While you may work on individual microservices independently, there may be times when you'll need to run the entire system locally - just as it will run when deployed to production. There are several tools that can help: 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. It is your only option if you're using Windows Containers, and is a great choice for non-Windows containers as well. Figure 3-4 shows how to enable local Kubernetes support when running Docker Desktop.

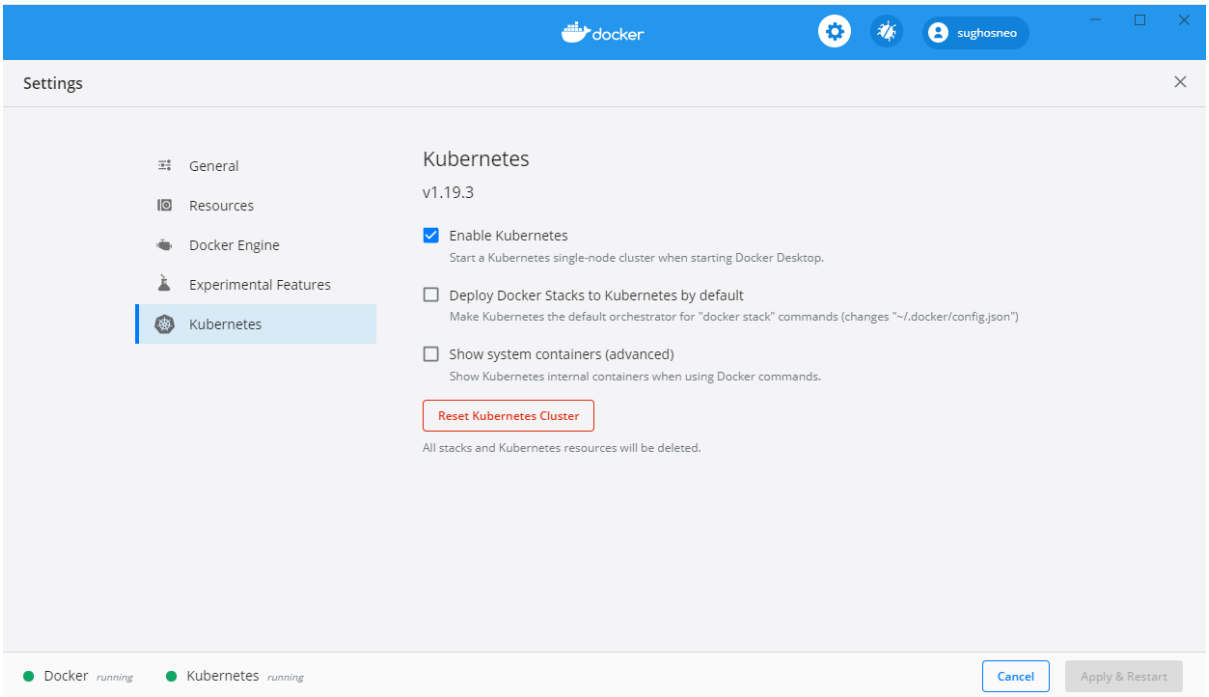


Figure 3-4. Configuring Kubernetes in Docker Desktop.

Docker Desktop is 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. It supports both Linux and Windows containers. Once you push your images to an image registry, like Azure Container Registry or Docker Hub, AKS can pull and deploy them to production.

Visual Studio Docker Tooling

Visual Studio supports Docker development for web-based applications. When you create a new ASP.NET Core application, you have an option to configure it with Docker support, as shown in Figure 3-5.

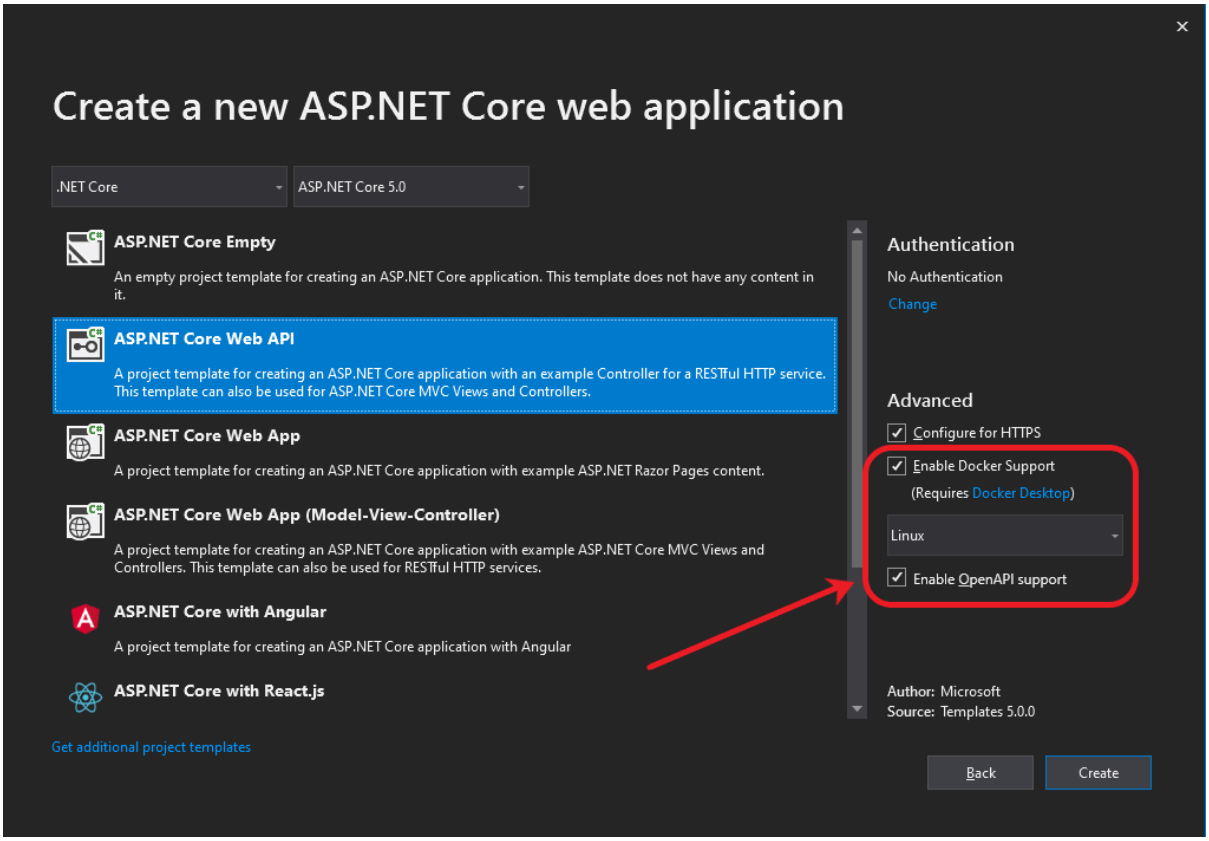


Figure 3-5. Visual Studio Enable Docker Support

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.git

```
FROM mcr.microsoft.com/dotnet/aspnet:5.0-buster-slim AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:5.0-buster-slim AS build
WORKDIR /src
COPY ["eShopWeb/eShopWeb.csproj", "eShopWeb/"]
RUN dotnet restore "eShopWeb/eShopWeb.csproj"
COPY . .
WORKDIR "/src/eShopWeb"
RUN dotnet build "eShopWeb.csproj" -c Release -o /app/build

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

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "eShopWeb.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.

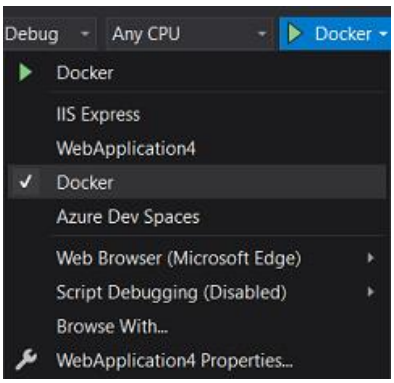


Figure 3-7. Visual Studio Docker Run Options

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 3-7, you can also run the application in Azure Dev Spaces.

Also, at any time you can add Docker support to an existing ASP.NET Core application. From the Visual Studio Solution Explorer, right-click on the project and select **Add** > **Docker Support**, as shown in Figure 3-8.

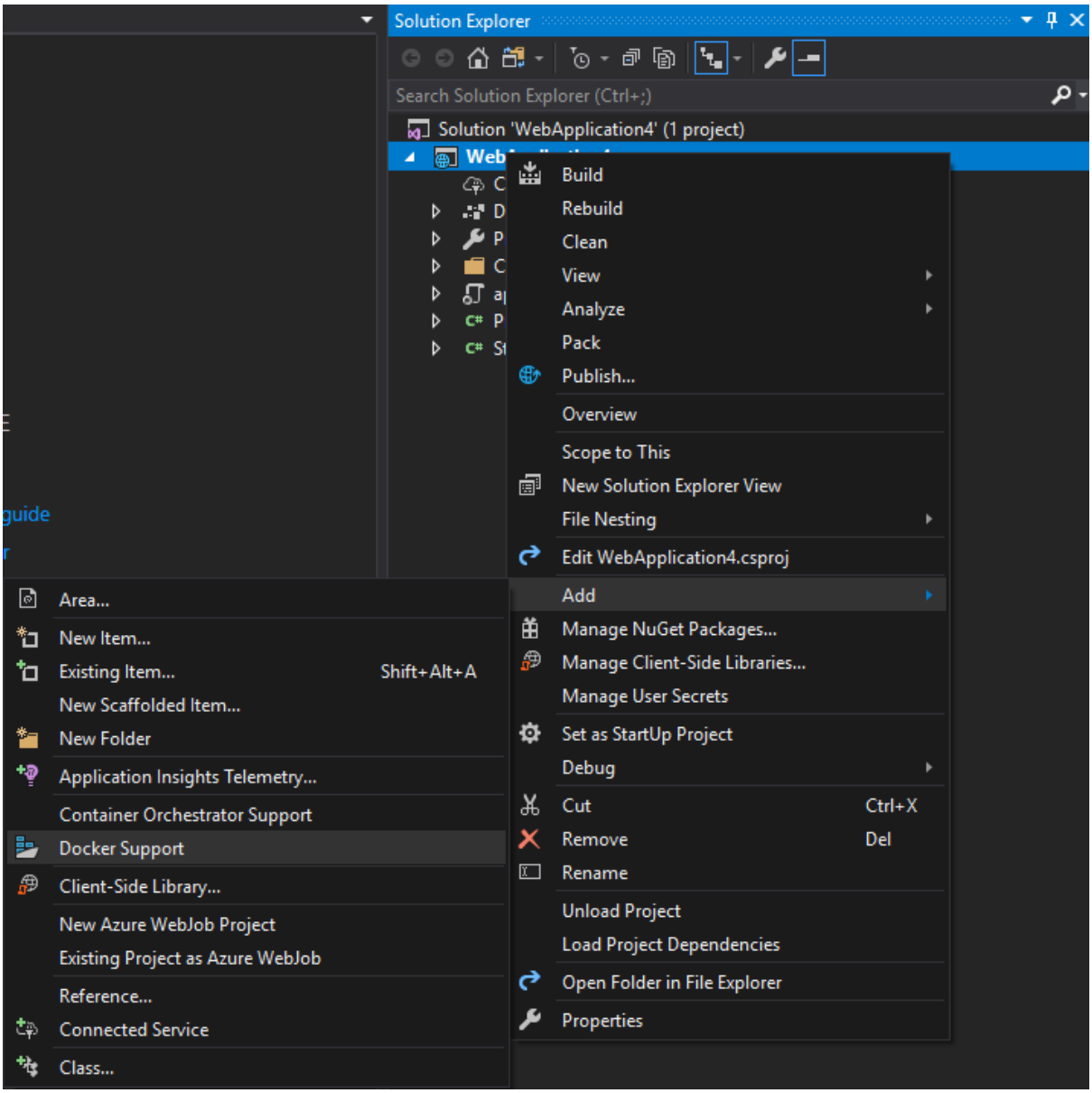


Figure 3-8. Adding Docker support to Visual Studio

You can also add Container Orchestration Support, also shown in Figure 3-8. 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.

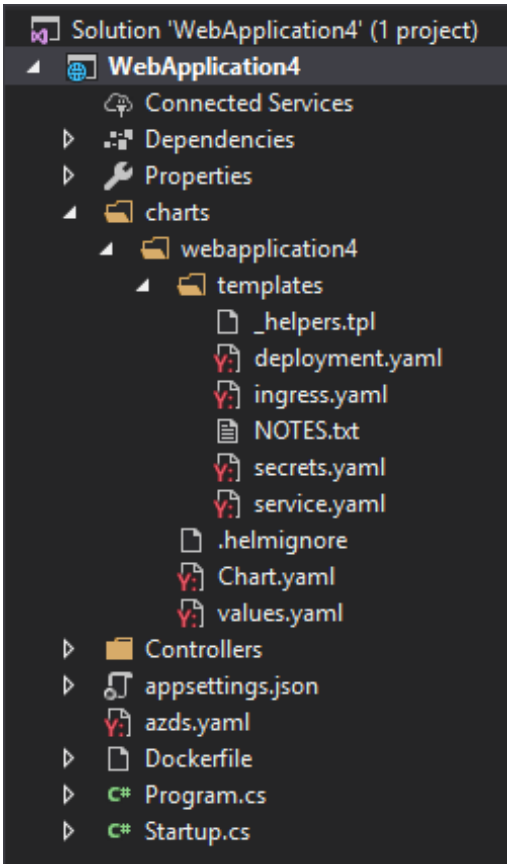


Figure 3-9. Adding orchestration support to Visual Studio

Visual Studio Code Docker Tooling

There are a number of extensions available for Visual Studio Code that support Docker development.

Microsoft provides the Docker for Visual Studio Code extension. This extension simplifies the process of adding container support to applications. It scaffolds required files, builds Docker images, and enables you to debug your app inside a container. The extension features a visual explorer that makes it easy to take actions on containers and images such as start, stop, inspect, remove, and more. The extension also supports Docker Compose enabling you to manage multiple running containers as a single unit.

Leveraging serverless functions

In the spectrum from managing physical machines to leveraging cloud capabilities, serverless lives at the extreme end. Your only responsibility is your code, and you only pay when your code runs. Azure Functions provides a way to build serverless capabilities into your cloud-native applications.

What is serverless?

Serverless is a relatively new service model of cloud computing. It doesn't mean that servers are optional - your code still runs on a server somewhere. The distinction is that the application team no

longer concerns itself with managing server infrastructure. Instead, the cloud vendor own this responsibility. The development team increases its productivity by delivering business solutions to customers, not plumbing.

Serverless computing uses event-triggered stateless containers to host your services. They can scale out and in to meet demand as-needed. Serverless platforms like Azure Functions have tight integration with other Azure services like queues, events, and storage.

What challenges are solved by serverless?

Serverless platforms address many time-consuming and expensive concerns:

- 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 allocate large budgets to support hardware infrastructure concerns. Moving to the cloud can help reduce these costs; shifting applications to serverless can help eliminate them.

What is the difference between a microservice and a serverless function?

Typically, a microservice encapsulates a business capability, such as a shopping cart for an online eCommerce site. It exposes multiple operations that enable a user to manage their shopping experience. A function, however, is a small, lightweight block of code that executes a single-purpose operation in response to an event. Microservices are typically constructed to respond to requests, often from an interface. Requests can be HTTP Rest- or gRPC-based. Serverless services respond to events. Its event-driven architecture is ideal for processing short-running, background tasks.

What scenarios are appropriate for serverless?

Serverless exposes individual short-running functions that are invoked in response to a trigger. This makes them ideal for processing background tasks.

An application might need to send an email as a step in a workflow. Instead of sending the notification as part of a microservice request, place the message details onto a queue. An Azure Function can dequeue the message and asynchronously send the email. Doing so could improve the performance and scalability of the microservice. Queue-based load leveling can be implemented to avoid bottlenecks related to sending the emails. Additionally, this stand-alone service could be reused as a utility across many different applications.

Asynchronous messaging from queues and topics is a common pattern to trigger serverless functions. However, Azure Functions can be triggered by other events, such as changes to Azure Blob Storage. A service that supports image uploads could have an Azure Function responsible for optimizing the

image size. The function could be triggered directly by inserts into Azure Blob Storage, keeping complexity out of the microservice operations.

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

When should you avoid serverless?

Serverless solutions provision and scale on demand. When a new instance is invoked, cold starts are a common issue. A cold start is the period of time it takes to provision this instance. Normally, this delay might be a few seconds, but can be longer depending on various factors. Once provisioned, a single instance is kept alive as long as it receives periodic requests. But, if a service is called less frequently, Azure may remove it from memory and require a cold start when reinvoked. Cold starts are also required when a function scales out to a new instance.

Figure 3-10 shows a cold-start pattern. Note the extra steps required when the app is cold.

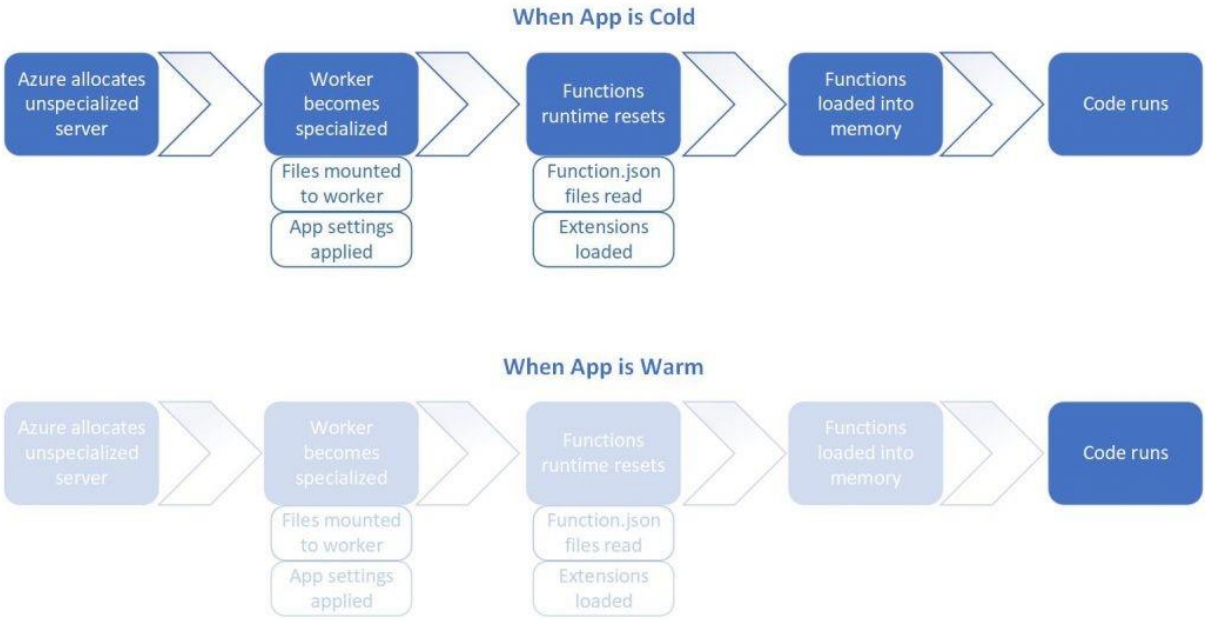


Figure 3-10. Cold start versus warm start.

To avoid cold starts entirely, you might switch from a consumption plan to a dedicated plan. You can also configure one or more pre-warmed instances with the premium plan upgrade. In these cases, when you need to add another instance, it's already up and ready to go. These options can help mitigate the cold start issue associated with serverless computing.

Cloud providers bill for serverless based on compute execution time and consumed memory. Long running operations or high memory consumption workloads aren't always the best candidates for serverless. Serverless functions favor small chunks of work that can complete quickly. Most serverless platforms require individual functions to complete within a few minutes. Azure Functions defaults to a

5-minute time-out duration, which can be configured up to 10 minutes. The Azure Functions premium plan can mitigate this issue as well, defaulting time-outs to 30 minutes with an unbounded higher limit that can be configured. Compute time isn't calendar time. More advanced functions using the Azure Durable Functions framework may pause execution over a course of several days. The billing is based on actual execution time - when the function wakes up and resumes processing.

Finally, leveraging Azure Functions for application tasks adds complexity. It's wise to first architect your application with a modular, loosely coupled design. Then, identify if there are benefits serverless would offer that justify the additional complexity.

Combining containers and serverless approaches

Cloud-native applications typically implement services leveraging containers and orchestration. There are often opportunities to expose some of the application's services as Azure Functions. However, with a cloud-native app deployed to Kubernetes, it would be nice to leverage Azure Functions within this same toolset. Fortunately, you can wrap Azure Functions inside 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?

Your Azure Function has no knowledge of the platform on which it's deployed. For some scenarios, you may have specific requirements and need to customize the environment on which your function code will run. You'll need a custom image that supports dependencies or a configuration not supported by the default image. In these cases, it makes sense to deploy your function in a custom Docker container.

When should you avoid using containers with Azure Functions?

If you want to use consumption billing, you won't be able to run your function in a container. What's more, if you deploy your function to a 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 earlier in this chapter.

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 --worker-runtime dotnet --docker
```

When the project is created, it will include a Dockerfile and the worker runtime configured to `dotnet`. 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

In this chapter, you've seen that the Azure Functions' platform automatically scales out to meet demand. When deploying containerized functions to AKS, however, you lose the built-in scaling functionality. To the rescue comes Kubernetes-based Event Driven (KEDA). It enables fine-grained autoscaling for `event-driven Kubernetes workloads,` including containerized functions.

KEDA provides event-driven scaling functionality to the Functions' runtime in a Docker container. KEDA can scale from zero instances (when no events are occurring) out to `n instances`, based on load. It enables autoscaling by exposing custom metrics to the Kubernetes autoscaler (Horizontal Pod Autoscaler). Using Functions containers with KEDA makes it possible to replicate serverless function capabilities in any Kubernetes cluster.

It is worth noting that the KEDA project is now managed by the Cloud Native Computing Foundation (CNCF).

Deploying containers in Azure

We've discussed containers in this chapter and in chapter 1. We've seen that containers provide many benefits to cloud-native applications, including portability. In the Azure cloud, you can deploy the same containerized services across staging and production environments. Azure provides several options for hosting these containerized workloads:

- Azure Kubernetes Services (AKS)
- Azure Container Instance (ACI)
- Azure Web Apps for Containers

Azure Container Registry

When containerizing a microservice, you first build a container "image." The image is a binary representation of the service code, dependencies, and runtime. While you can manually create an image using the `Docker Build` command from the Docker API, a better approach is to create it as part of an automated build process.

Once created, container images are stored in container registries. They enable you to build, store, and manage container images. There are many registries available, both public and private. Azure Container Registry (ACR) is a fully managed container registry service in the Azure cloud. It persists your images inside the Azure network, reducing the time to deploy them to Azure container hosts. You can also secure them using the same security and identity procedures that you use for other Azure resources.

You create an Azure Container Registry using the Azure portal, Azure CLI, or PowerShell tools. Creating a registry in Azure is simple. It requires an Azure subscription, 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`.

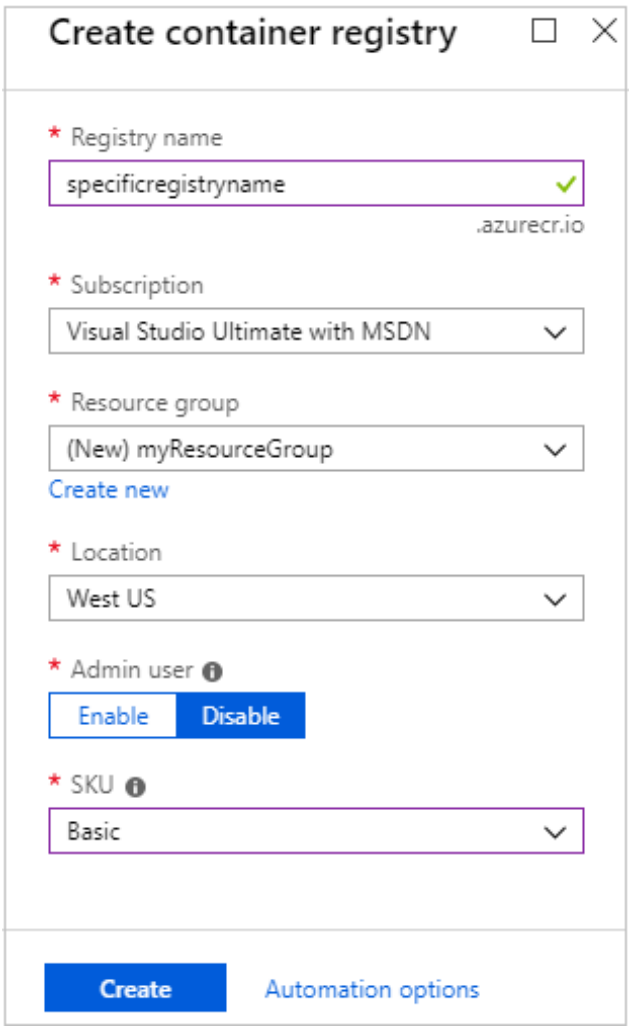


Figure 3-11. Create container registry

Once you've created the 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 authenticated, you can use docker commands to push container images to it. Before you can do so, however, you must tag your image with the fully qualified name (URL) of your ACR login server. It 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
```

As a best practice, developers shouldn't manually push images to a container registry. Instead, a build pipeline defined in a tool like GitHub or Azure DevOps should be responsible for this process. Learn more in the Cloud-Native DevOps chapter.

ACR Tasks

ACR Tasks is a set of features available from the Azure Container Registry. It extends your inner-loop development cycle by building and managing container images in the Azure cloud. Instead of invoking a `docker build` and `docker push` locally on your development machine, they're automatically handled by ACR Tasks in the cloud.

The following AZ CLI command both builds a container image and pushes it to ACR:

```
# create a container registry
az acr create --resource-group myResourceGroup --name myContainerRegistry008 --sku Basic

# build container image in ACR and push it into your container regsitry
az acr build --image sample/hello-world:v1  --registry myContainerRegistry008 --file
Dockerfile .
```

As you can see from the previous command block, there's no need to install Docker Desktop on your development machine. Additionally, you can configure ACR Task triggers to rebuild containers images on both source code and base image updates.

Azure Kubernetes Service

We discussed Azure Kubernetes Service (AKS) at length in this chapter. We've seen that it's the de facto container orchestrator managing containerized cloud-native applications.

Once you deploy an image to a registry, such as ACR, you can configure AKS to automatically pull and deploy it. With a CI/CD pipeline in place, you might 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. Then, the system will route a small percentage of users to the newly deployed version. As the team gains confidence in the new version, it can roll out more instances and retire the old. AKS easily supports this style of deployment.

As with most resources in Azure, you can create an Azure Kubernetes Service cluster using the portal, command-line, or 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

Cloud-native applications can quickly grow large and complex, requiring significant compute resources to run. In these scenarios, the entire application can't be hosted on a development machine (especially a laptop). Azure Dev Spaces is designed to address this problem using AKS. It enables developers to work with a local version of their services while hosting the rest of the application in an AKS development cluster.

Developers share a running (development) instance in an AKS cluster that contains the entire containerized application. But they use personal spaces set up on their machine to locally develop their services. When ready, they test from end-to-end in the AKS cluster - without replicating dependencies. Azure Dev Spaces merges code from the local machine with services in AKS. Developers can rapidly iterate and debug code directly in Kubernetes using Visual Studio or Visual Studio Code.

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're 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. The containers will be built and deployed according to your instructions. This task 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 reissuing 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.

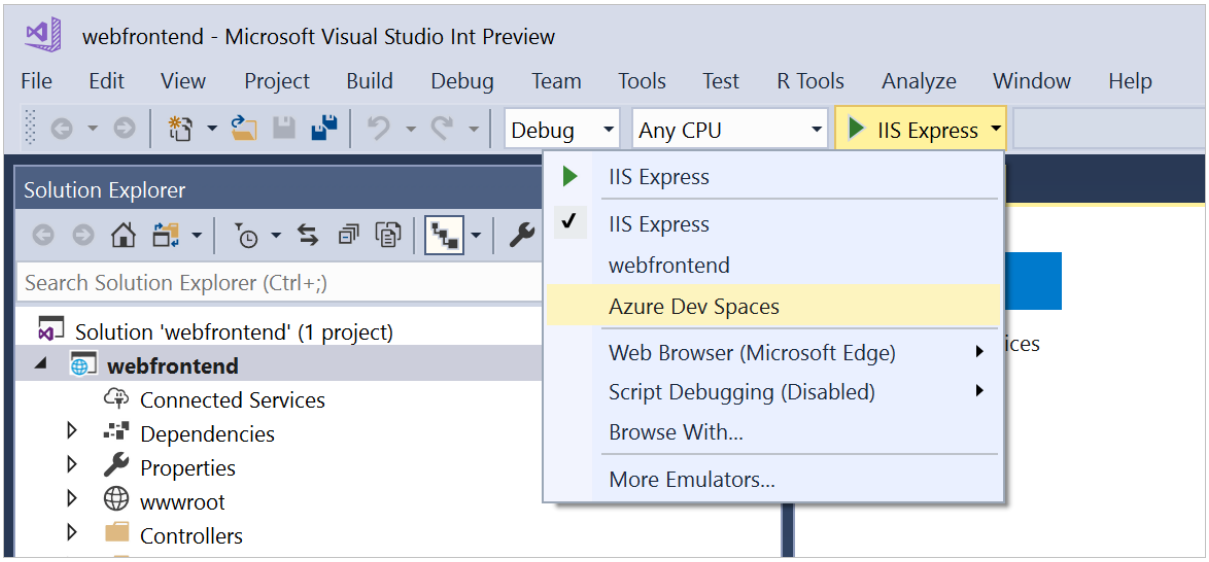


Figure 3-12. Connect to Azure Dev Spaces in Visual Studio

Scaling containers and serverless applications

There are two ways to scale an application: up or out. The former refers to adding capacity to a single resource, while the latter refers to adding more resources to increase capacity.

The simple solution: scaling up

Upgrading an existing host server with increased CPU, memory, disk I/O speed, and network I/O speed is known as *scaling up*. Scaling up a cloud-native application involves choosing more capable resources from the cloud vendor. For example, you can create a new node pool with larger VMs in your Kubernetes cluster. Then, migrate your containerized services to the new pool.

Serverless apps scale up by choosing the premium Functions plan or premium instance sizes from a dedicated app service plan.

Scaling out cloud-native apps

Cloud-native applications often experience large fluctuations in demand and require scale on a moment's notice. They favor scaling out. Scaling out is done horizontally by adding additional machines (called nodes) or application instances to an existing cluster. In Kubernetes, you can scale manually by adjusting configuration settings for the app (for example, scaling a node pool), or through autoscaling.

AKS clusters can autoscale in one of two ways:

First, the Horizontal Pod Autoscaler monitors resource demand and automatically scales your POD replicas to meet it. When traffic increases, additional replicas are automatically provisioned to scale out your services. Likewise, when demand decreases, they're removed to scale-in your services. You define the metric on which to scale, for example, CPU usage. You can also specify the minimum and maximum number of replicas to run. AKS monitors that metric and scales accordingly.

Next, the AKS Cluster Autoscaler feature enables you to automatically scale compute nodes across a Kubernetes cluster to meet demand. With it, you can automatically add new VMs to the underlying Azure Virtual Machine Scale Set whenever more compute capacity of is required. It also removes nodes when no longer required.

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

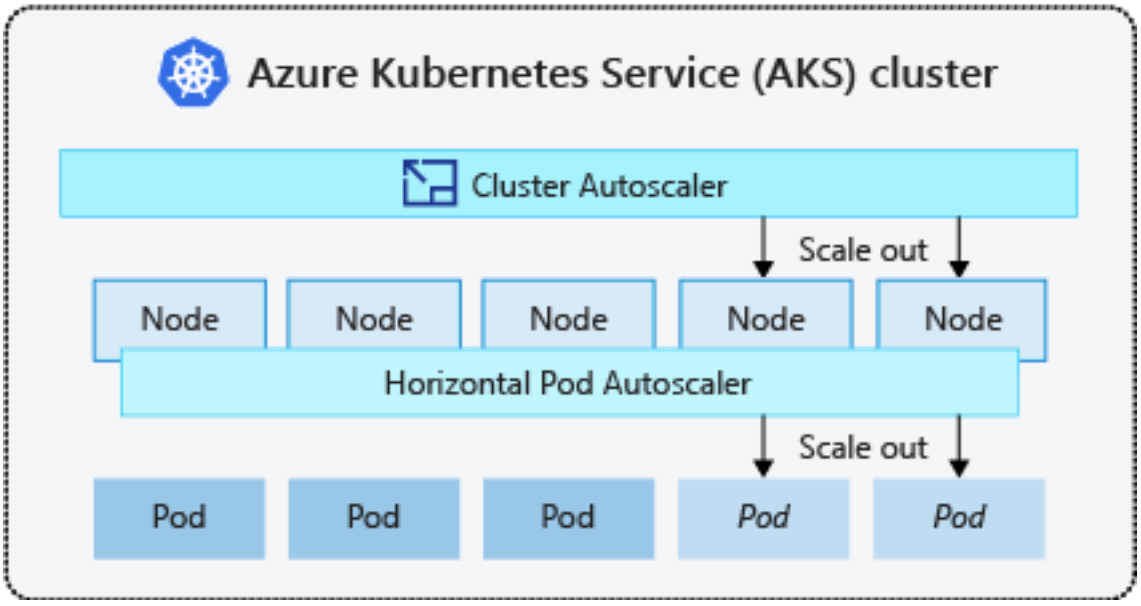


Figure 3-13. Scaling out an App Service plan.

Working together, both ensure an optimal number of container instances and compute nodes to support fluctuating demand. The horizontal pod autoscaler optimizes the number of pods required. The cluster autoscaler optimizes the number of nodes required.

Scaling Azure Functions

Azure Functions automatically scale out upon demand. Server resources are dynamically allocated and removed based on the number of triggered events. You're only charged for compute resources

consumed when your functions run. Billing is based upon the number of executions, execution time, and memory used.

While the default consumption plan provides an economical and scalable solution for most apps, the premium option allows developers flexibility for custom Azure Functions requirements. Upgrading to the premium plan provides control over instance sizes, pre-warmed instances (to avoid cold start delays), and dedicated VMs.

Other container deployment options

Aside from Azure Kubernetes Service (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'll need an Azure Container Registry (ACR) instance and credentials to access it. Push your container image to the ACR repository so it can pulled into your Azure App Service. Once complete, you can configure the app for Continuous Deployment. Doing so will automatically deploy updates whenever the image changes in ACR.

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

Azure Container Instances (ACI) enables you to run Docker containers in a managed, serverless cloud environment, without having to set up virtual machines or clusters. It's a great solution for short-running workloads that can run in an isolated container. Consider ACI for simple services, testing scenarios, task automation, and build jobs. ACI spins-up a container instance, performs the task, and then spins it down.

How to deploy an app to Azure Container Instances

To deploy to Azure Container Instances (ACI), you need an Azure Container Registry (ACR) and credentials for accessing it. Once you push your container image to the repository, it's available to pull into ACI. You can work with ACI using the Azure portal or command-line interface. ACR provides tight integration with ACI. Figure 3-14 shows how to push an individual container image to ACR.

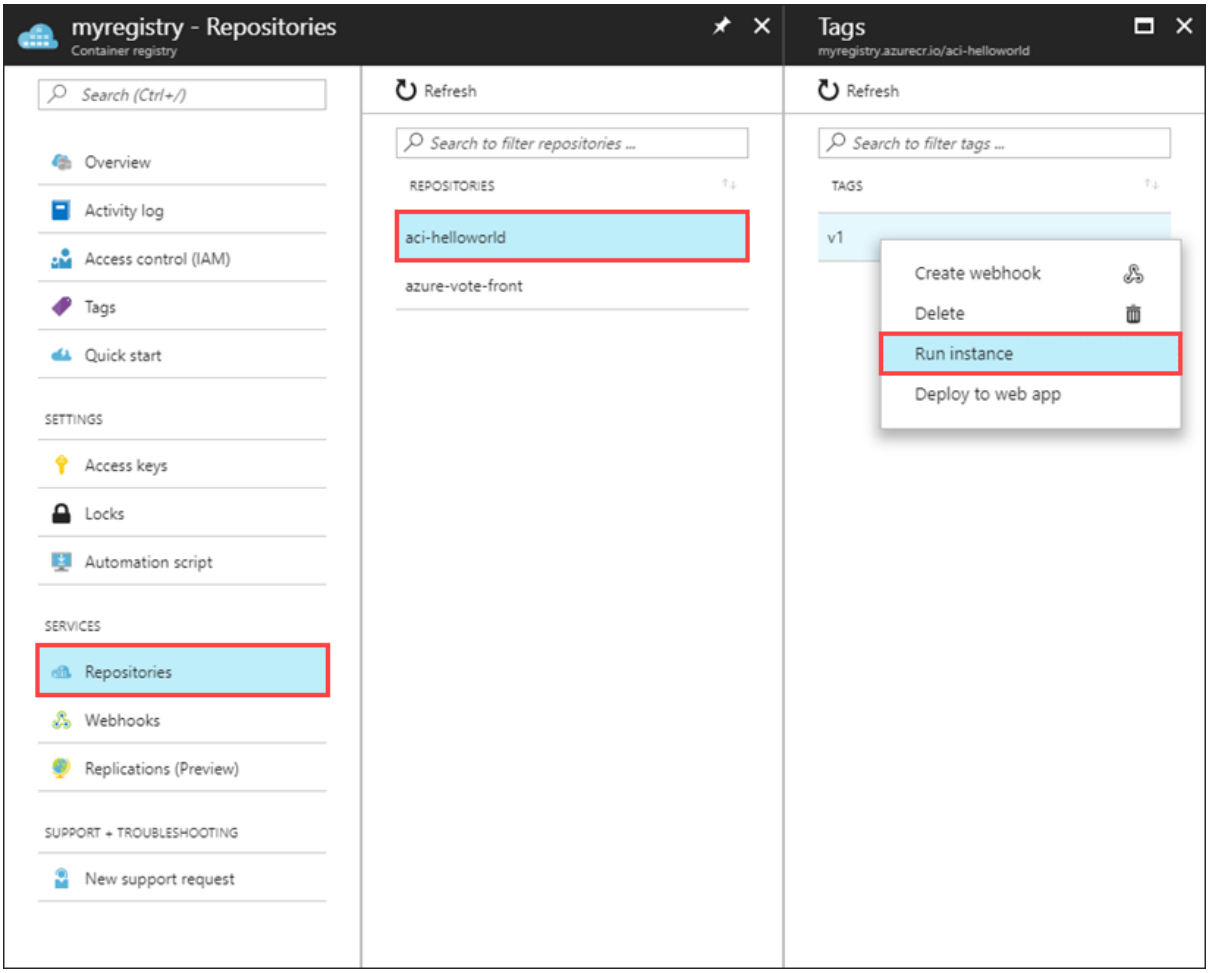


Figure 3-14. Azure Container Registry Run Instance

Creating an instance in ACI can be done quickly. Specify the image registry, Azure resource group information, the amount of memory to allocate, and the port on which to listen. 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 way to run simple container workloads in Azure. You don't need to configure an app service, orchestrator, or virtual machine. For scenarios where you require full container orchestration, service discovery, automatic scaling, or coordinated upgrades, we recommend Azure Kubernetes Service (AKS).

References

- What is Kubernetes?
- Installing Kubernetes with Minikube
- MiniKube vs Docker Desktop
- Visual Studio Tools for Docker
- Understanding serverless cold start
- Pre-warmed Azure Functions instances
- Create a function on Linux using a custom image
- Run Azure Functions in a Docker Container
- Create a function on Linux using a custom image
- Azure Functions with Kubernetes Event Driven Autoscaling
- Canary Release
- Azure Dev Spaces with VS Code
- Azure Dev Spaces with Visual Studio
- AKS Multiple Node Pools
- AKS Cluster Autoscaler
- Tutorial: Scale applications in AKS
- Azure Functions scale and hosting
- 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.

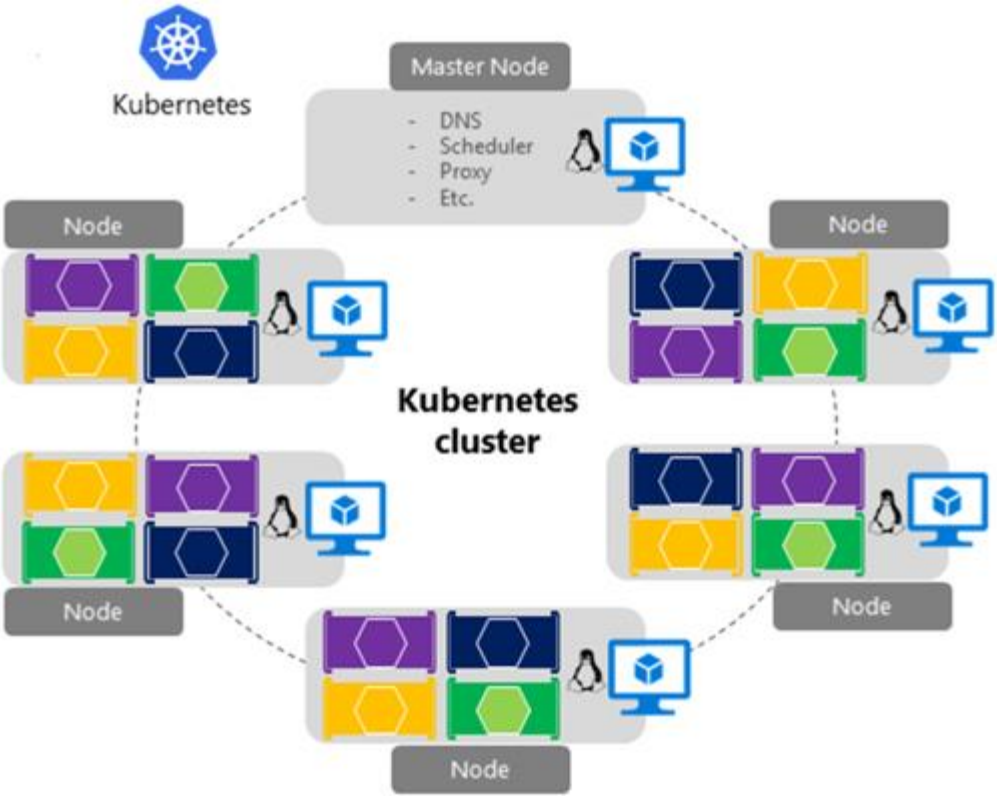


Figure 4-1. A Kubernetes cluster in Azure

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.

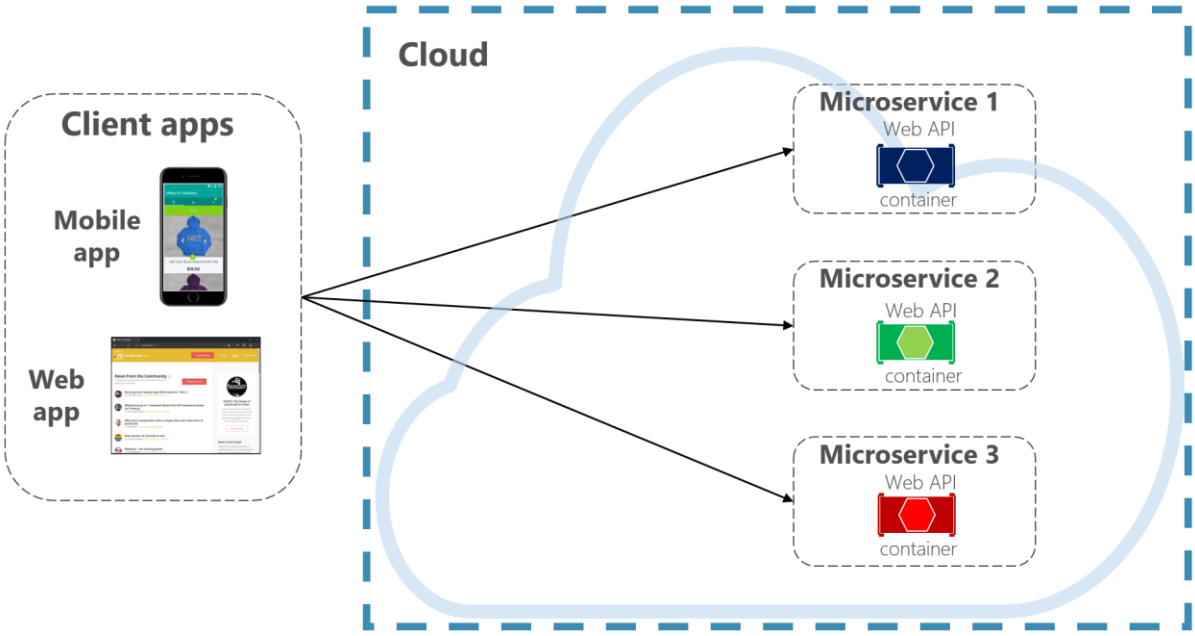


Figure 4-2. Direct client to service communication

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.

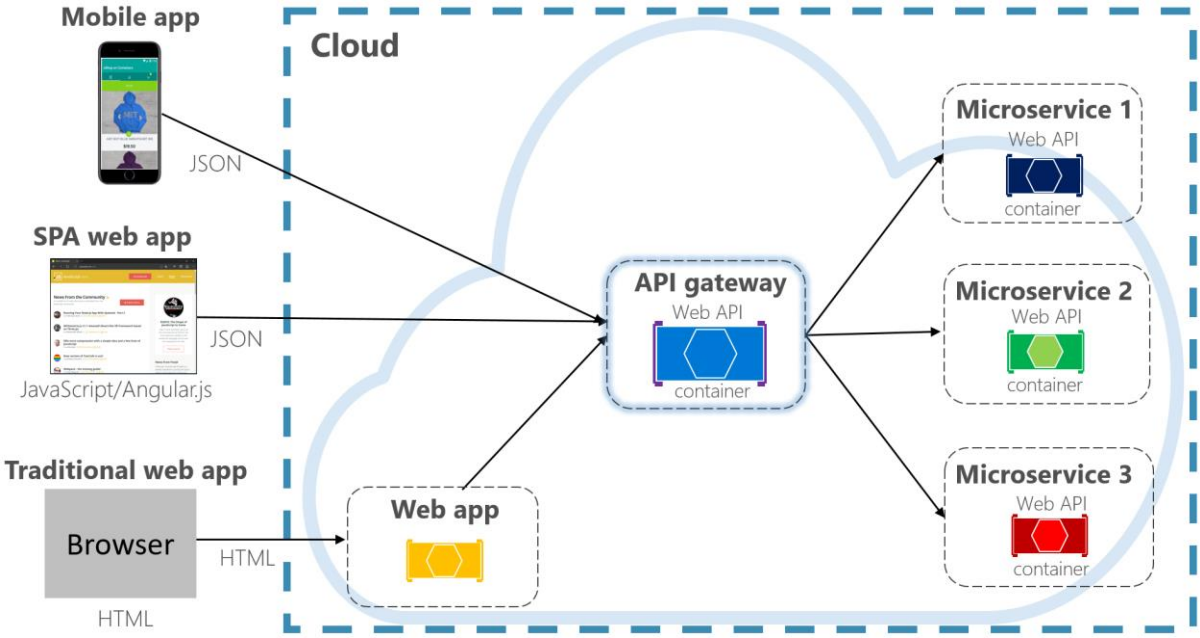


Figure 4-3. API gateway pattern

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.

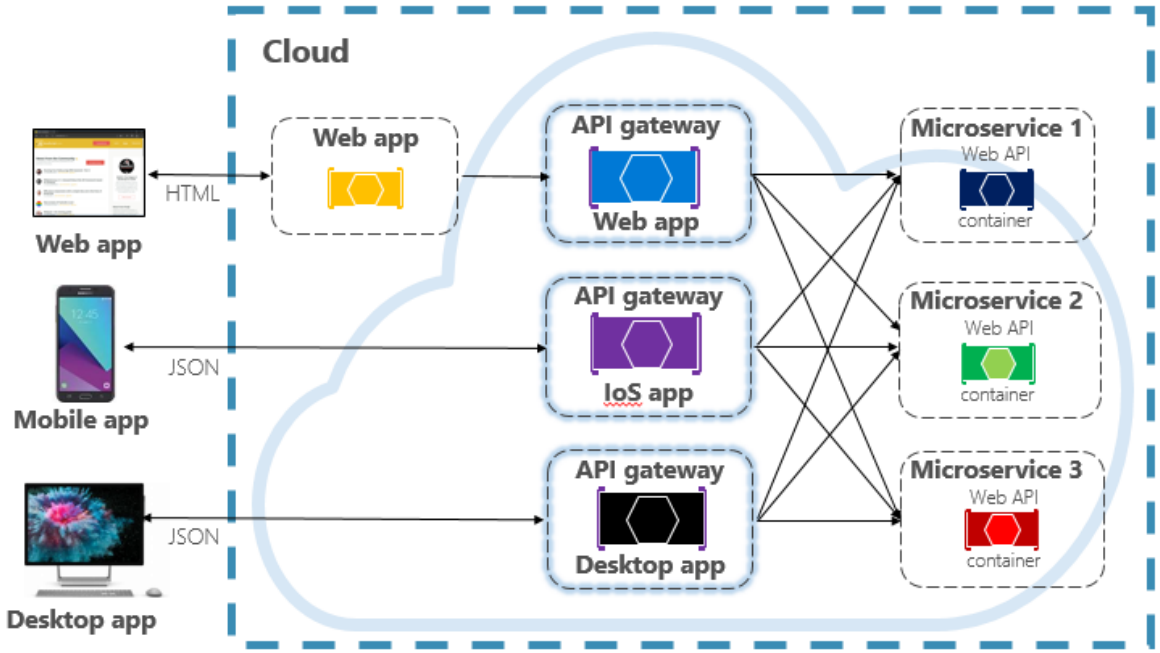


Figure 4-4. Backend for frontend pattern

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 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.

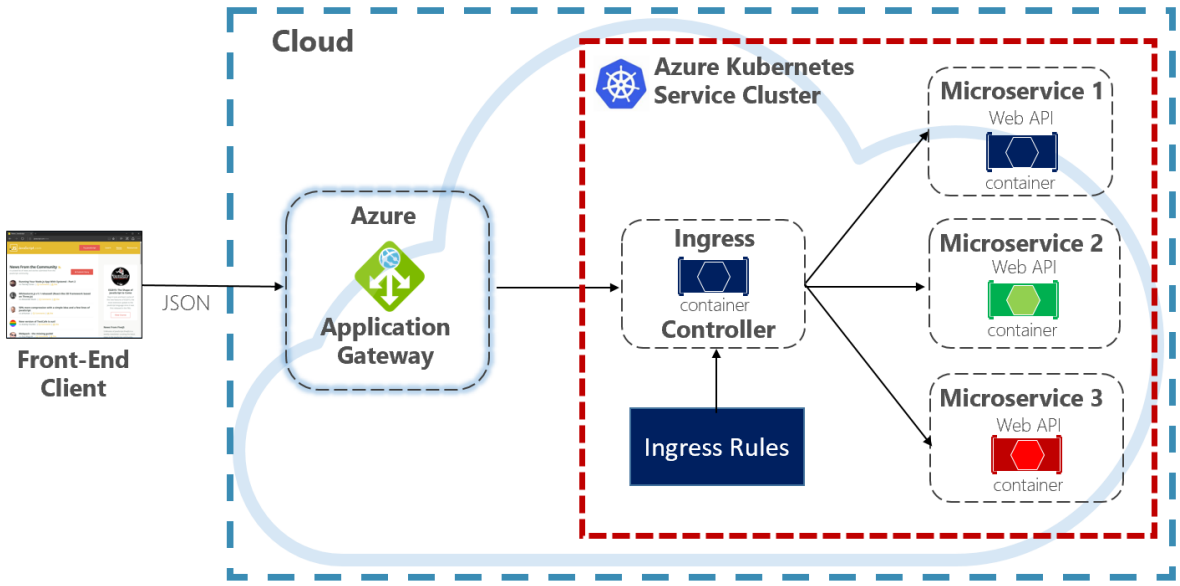


Figure 4-5. Application Gateway Ingress Controller

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.

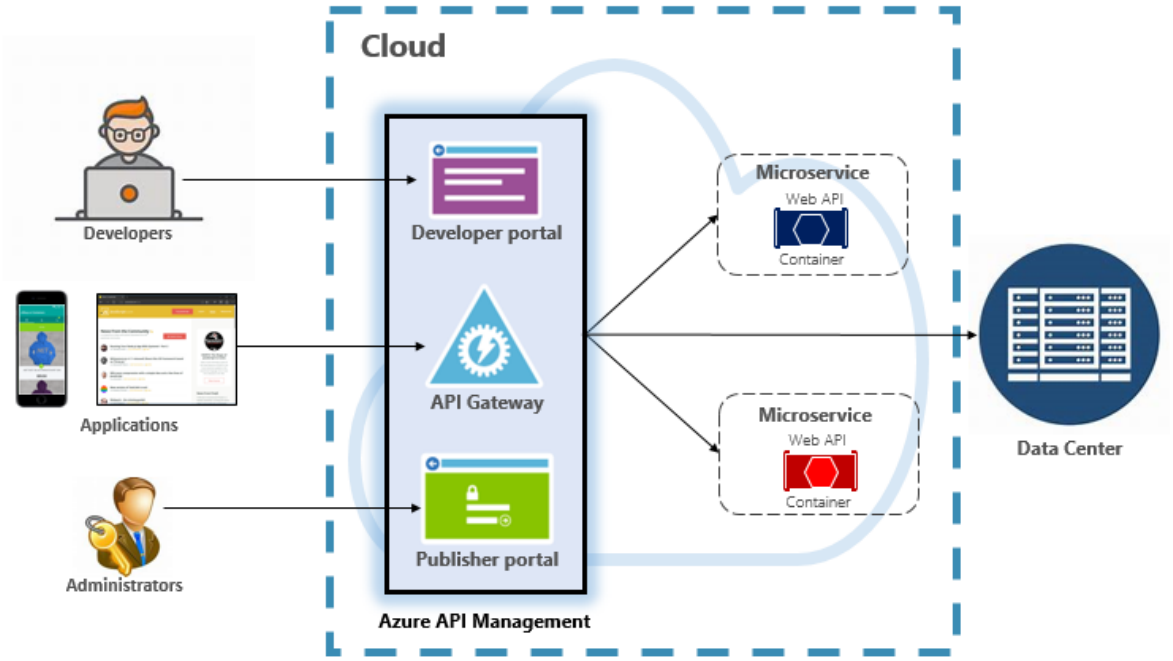


Figure 4-6. Azure API Management

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.

The Azure cloud also offers a 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 most 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.

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.

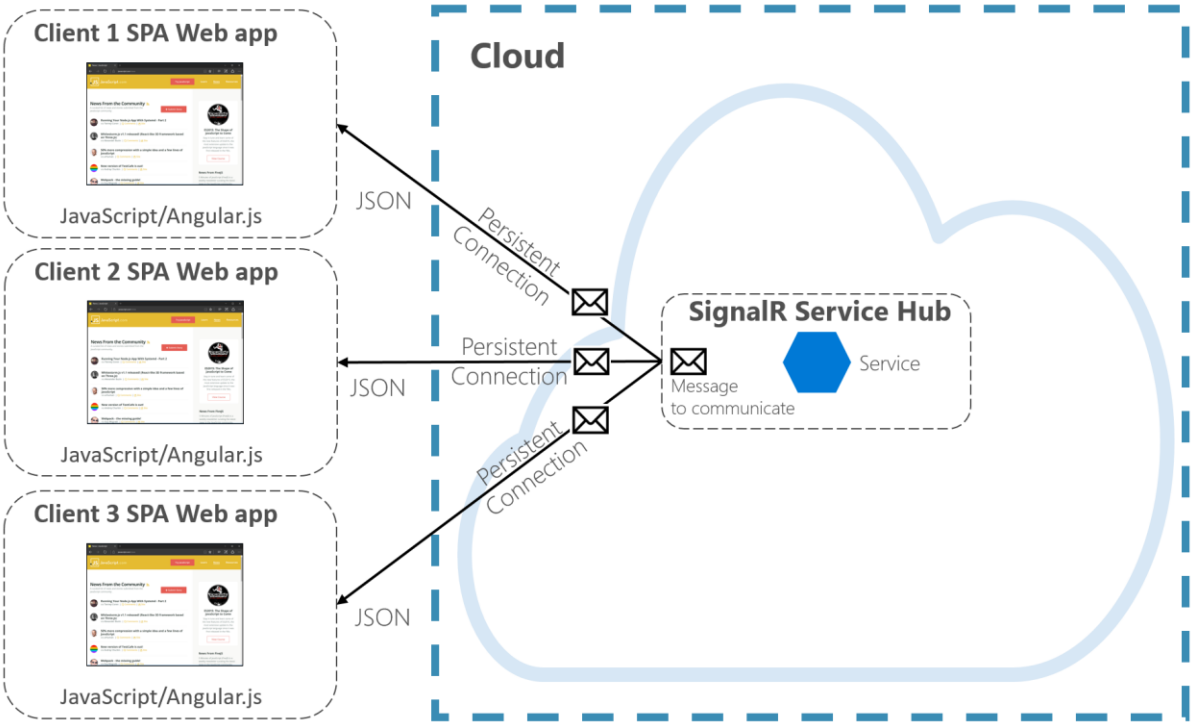


Figure 4-7. Azure SignalR

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 many 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.

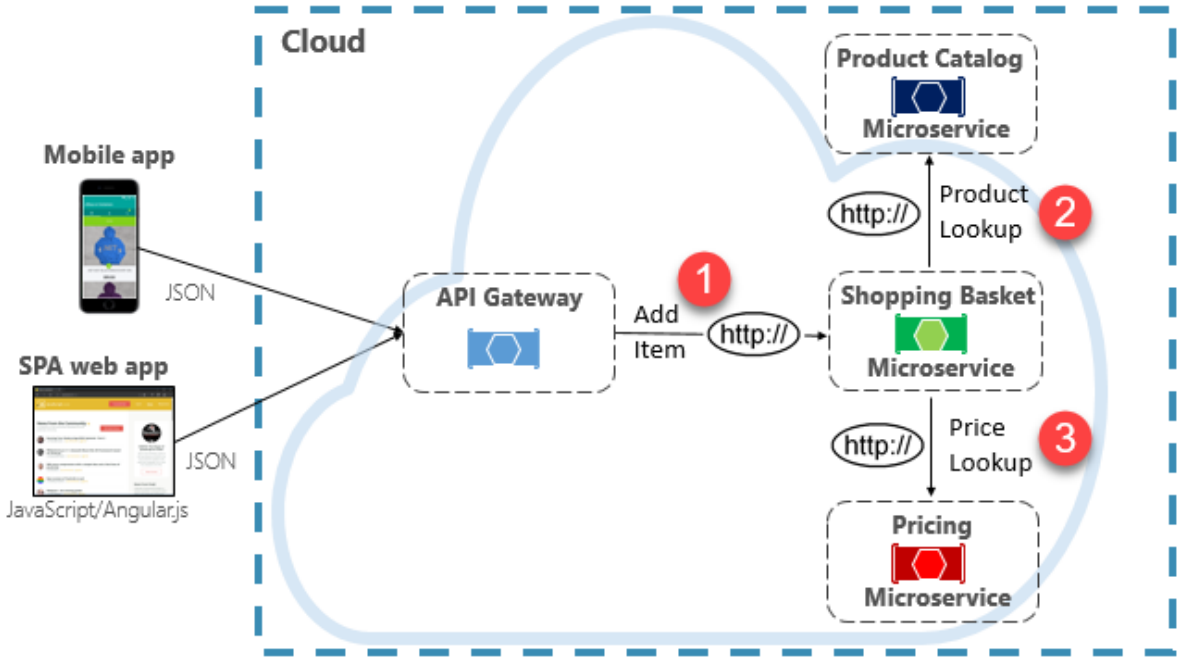


Figure 4-8. Direct HTTP communication

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:

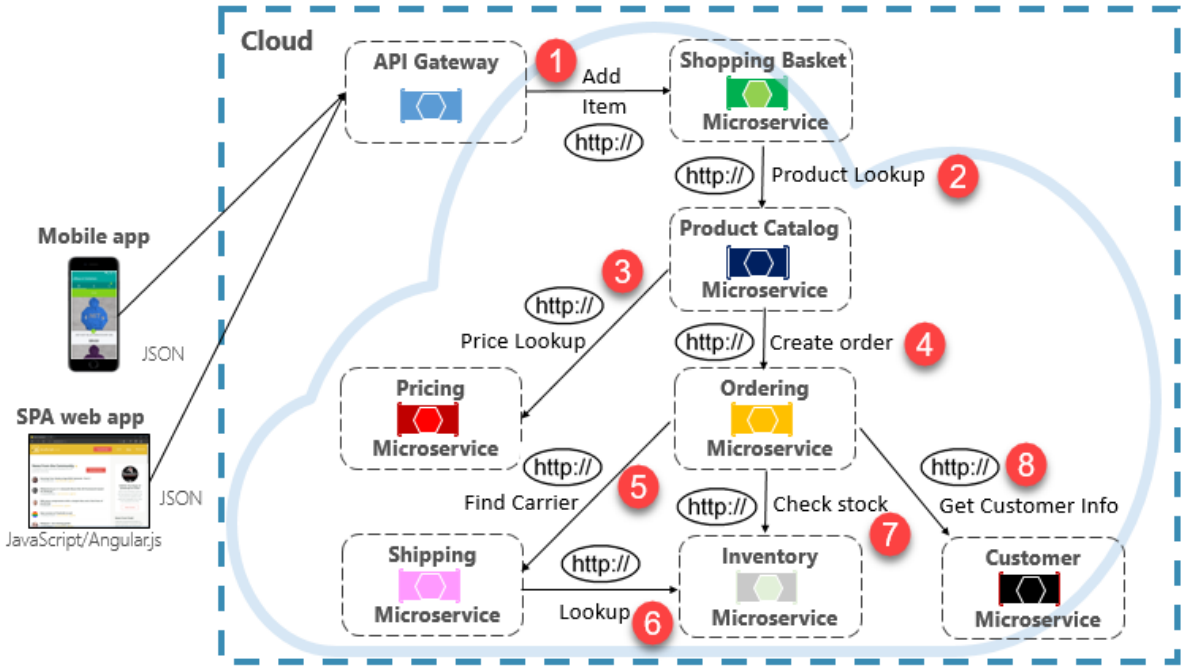


Figure 4-9. Chaining HTTP queries

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.

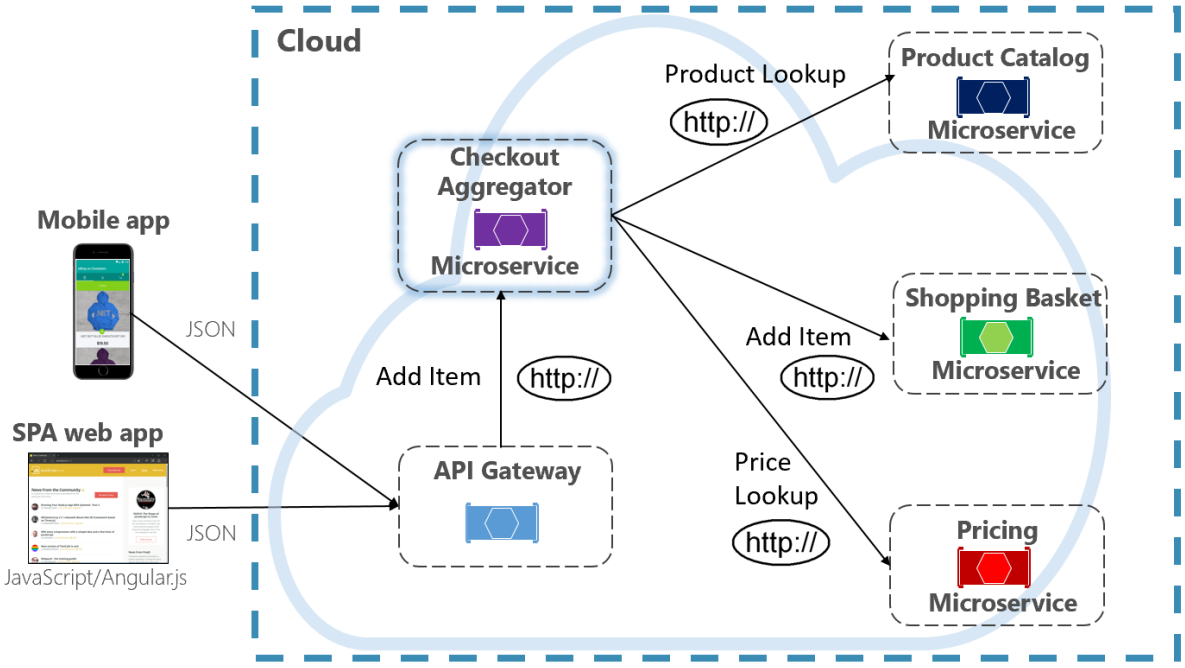


Figure 4-10. Aggregator microservice

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.

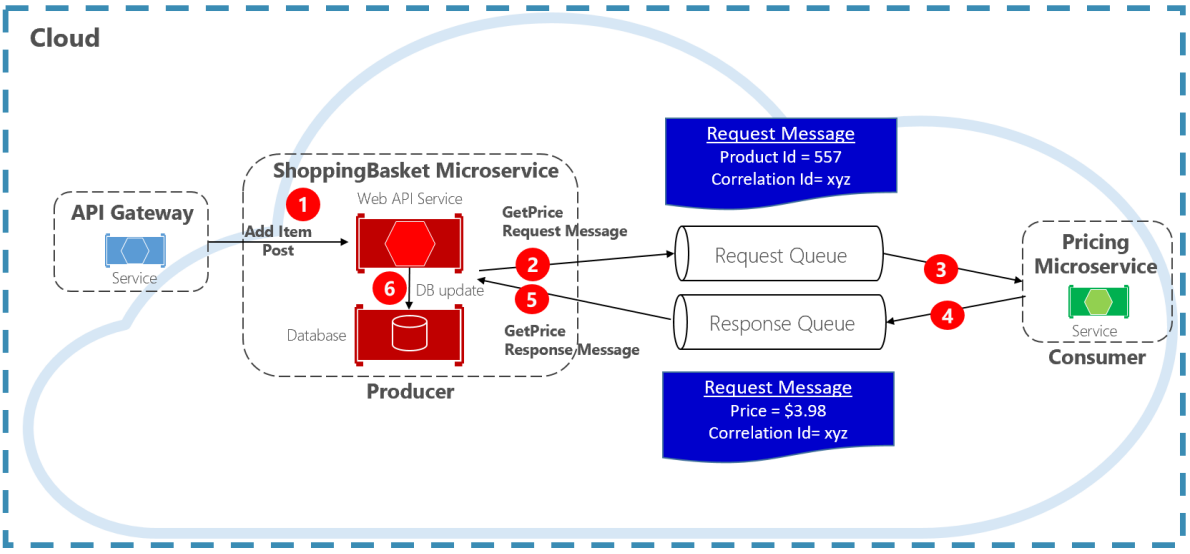


Figure 4-11. Request-reply pattern

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.

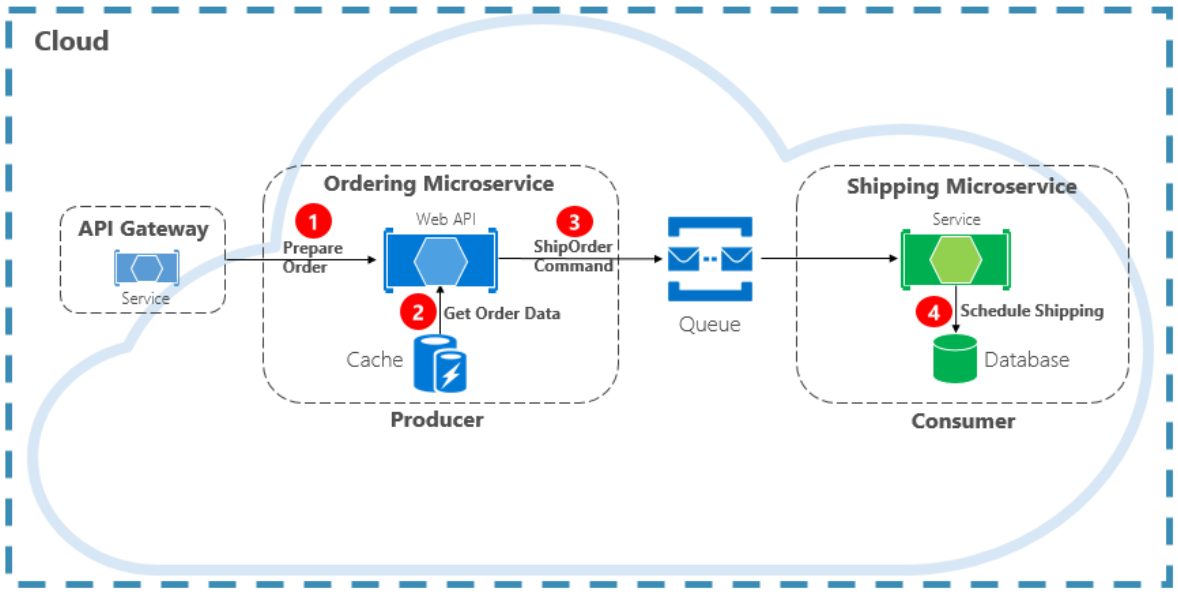


Figure 4-12. Command interaction with a queue

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.

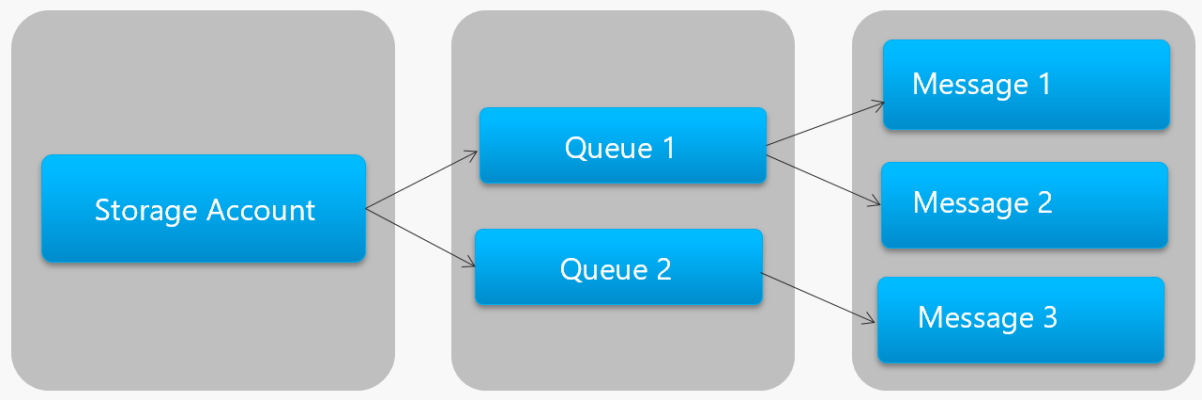


Figure 4-13. Storage queue hierarchy

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 AMQP protocol. AMQP 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.

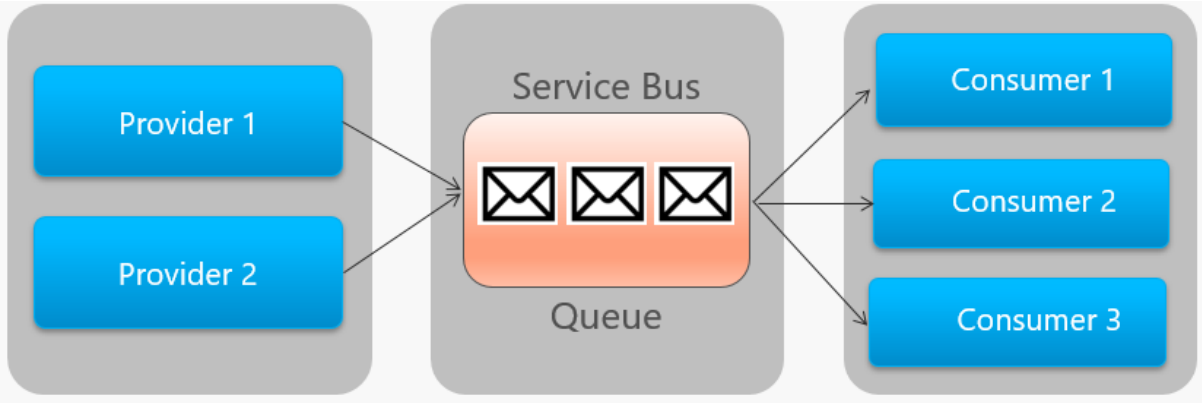


Figure 4-14. Service Bus queue

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.

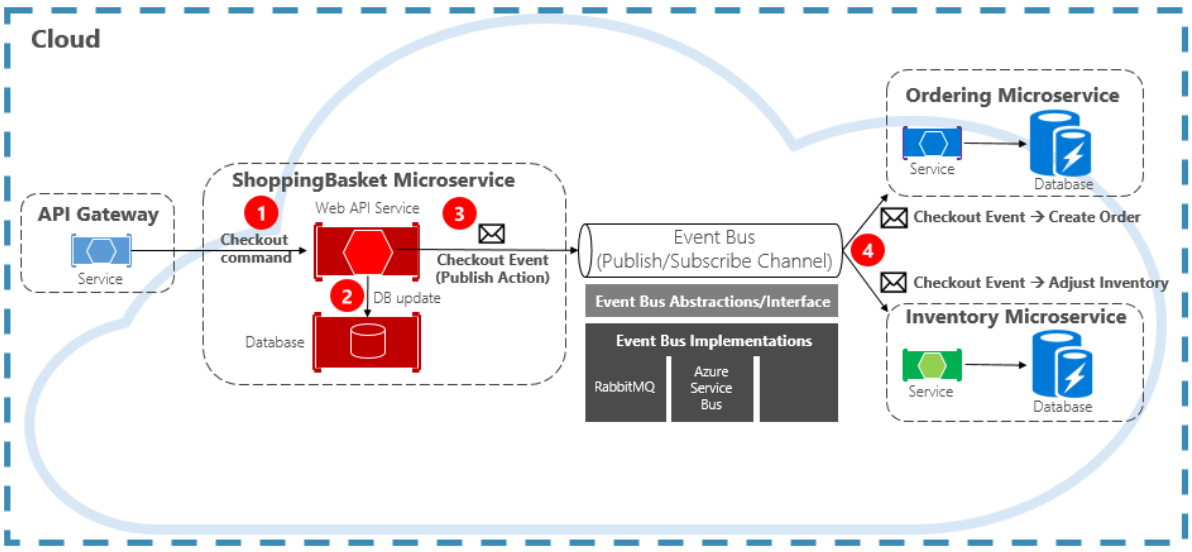


Figure 4-15. Event-Driven messaging

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.

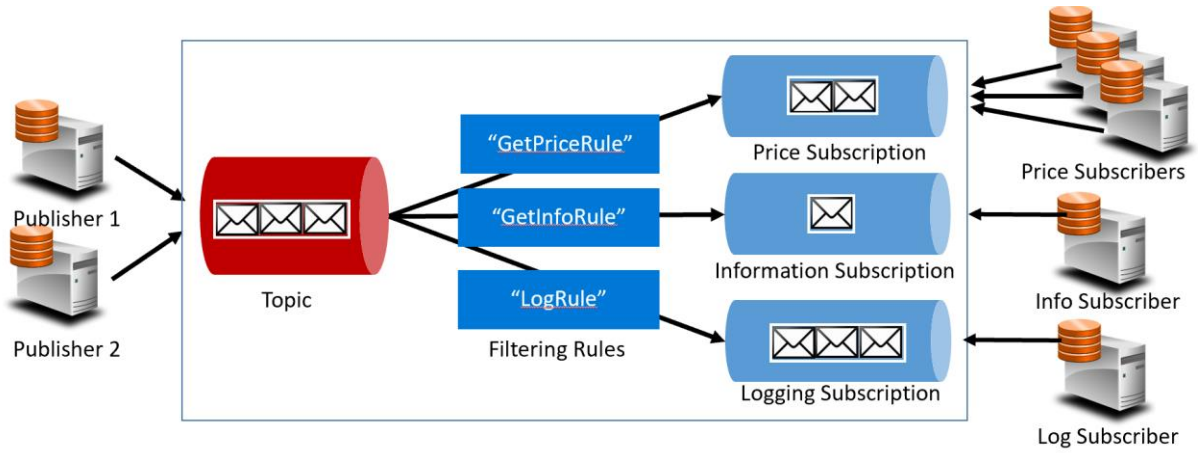


Figure 4-16. Topic architecture

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 "GetPrice" event would be sent to the price and logging subscriptions as the logging subscription has chosen to receive all messages. A "GetInformation" event would be sent to the information and logging subscriptions.

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.

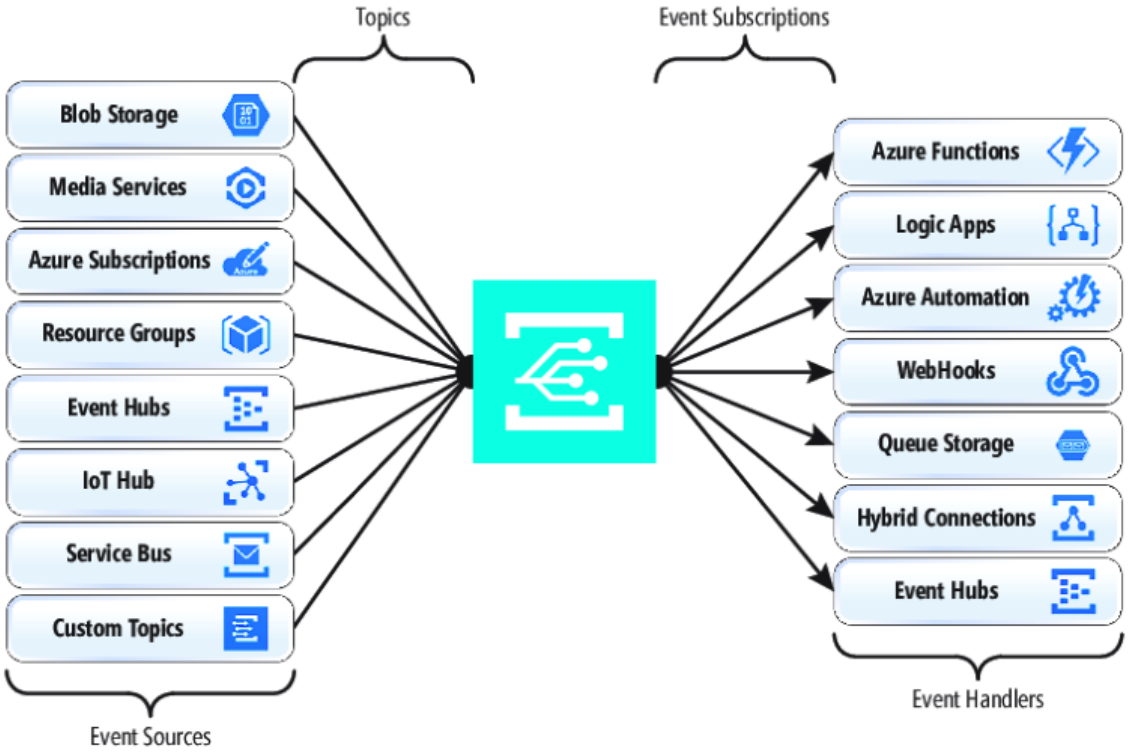


Figure 4-17. Event Grid anatomy

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 has 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.

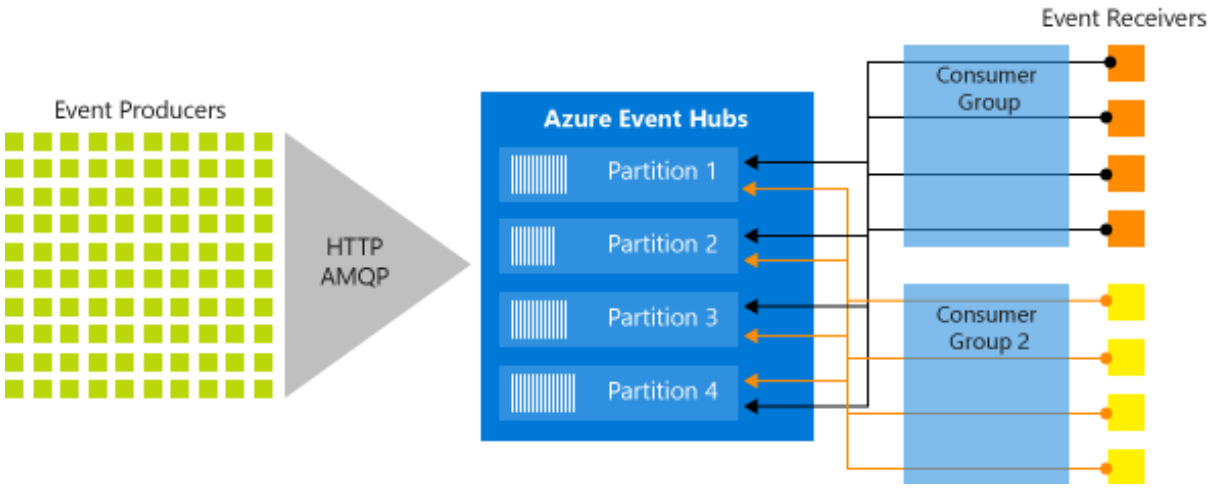


Figure 4-19. Event Hub partitioning

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.

gRPC

So far in this book, we've focused on REST-based communication. We've seen that REST is a flexible architectural style that defines CRUD-based operations against entity resources. Clients interact with resources across HTTP with a request/response communication model. While REST is widely implemented, a newer communication technology, gRPC, has gained tremendous momentum across the cloud-native community.

What is gRPC?

gRPC is a modern, high-performance framework that evolves the age-old remote procedure call (RPC) protocol. At the application level, gRPC streamlines messaging between clients and back-end services. Originating from Google, gRPC is open source and part of the Cloud Native Computing Foundation (CNCF) ecosystem of cloud-native offerings. CNCF considers gRPC an incubating project. Incubating means end users are using the technology in production applications, and the project has a healthy number of contributors.

A typical gRPC client app will expose a local, in-process function that implements a business operation. Under the covers, that local function invokes another function on a remote machine. What appears to be a local call essentially becomes a transparent out-of-process call to a remote service. The RPC plumbing abstracts the point-to-point networking communication, serialization, and execution between computers.

In cloud-native applications, developers often work across programming languages, frameworks, and technologies. This *interoperability* complicates message contracts and the plumbing required for cross-platform communication. gRPC provides a "uniform horizontal layer" that abstracts these concerns. Developers code in their native platform focused on business functionality, while gRPC handles communication plumbing.

gRPC offers comprehensive support across most popular development stacks, including Java, JavaScript, C#, Go, Swift, and NodeJS.

gRPC Benefits

gRPC uses HTTP/2 for its transport protocol. While compatible with HTTP 1.1, HTTP/2 features many advanced capabilities:

- A binary framing protocol for data transport - unlike HTTP 1.1, which is text based.
- Multiplexing support for sending multiple parallel requests over the same connection - HTTP 1.1 limits processing to one request/response message at a time.
- Bidirectional full-duplex communication for sending both client requests and server responses simultaneously.
- Built-in streaming enabling requests and responses to asynchronously stream large data sets.
- Header compression that reduces network usage.

gRPC is lightweight and highly performant. It can be up to 8x faster than JSON serialization with messages 60-80% smaller. In Microsoft Windows Communication Foundation (WCF) parlance, gRPC performance exceeds the speed and efficiency of the highly optimized NetTCP bindings. Unlike NetTCP, which favors the Microsoft stack, gRPC is cross-platform.

Protocol Buffers

gRPC embraces an open-source technology called Protocol Buffers. They provide a highly efficient and platform-neutral serialization format for serializing structured messages that services send to each other. Using a cross-platform Interface Definition Language (IDL), developers define a service contract for each microservice. The contract, implemented as a text-based `.proto` file, describes the methods, inputs, and outputs for each service. The same contract file can be used for gRPC clients and services built on different development platforms.

Using the proto file, the Protobuf compiler, `protoc`, generates both client and service code for your target platform. The code includes the following components:

- Strongly typed objects, shared by the client and service, that represent the service operations and data elements for a message.
- A strongly typed base class with the required network plumbing that the remote gRPC service can inherit and extend.
- A client stub that contains the required plumbing to invoke the remote gRPC service.

At runtime, each message is serialized as a standard Protobuf representation and exchanged between the client and remote service. Unlike JSON or XML, Protobuf messages are serialized as compiled binary bytes.

The book, gRPC for WCF Developers, available from the Microsoft Architecture site, provides in-depth coverage of gRPC and Protocol Buffers.

gRPC support in .NET

gRPC is integrated into .NET Core 3.0 SDK and later. The following tools support it:

- Visual Studio 2019, version 16.3 or later, with the web development workload installed.
- Visual Studio Code
- the dotnet CLI

The SDK includes tooling for endpoint routing, built-in IoC, and logging. The open-source Kestrel web server supports HTTP/2 connections. Figure 4-20 shows a Visual Studio 2019 template that scaffolds a skeleton project for a gRPC service. Note how .NET fully supports Windows, Linux, and macOS.

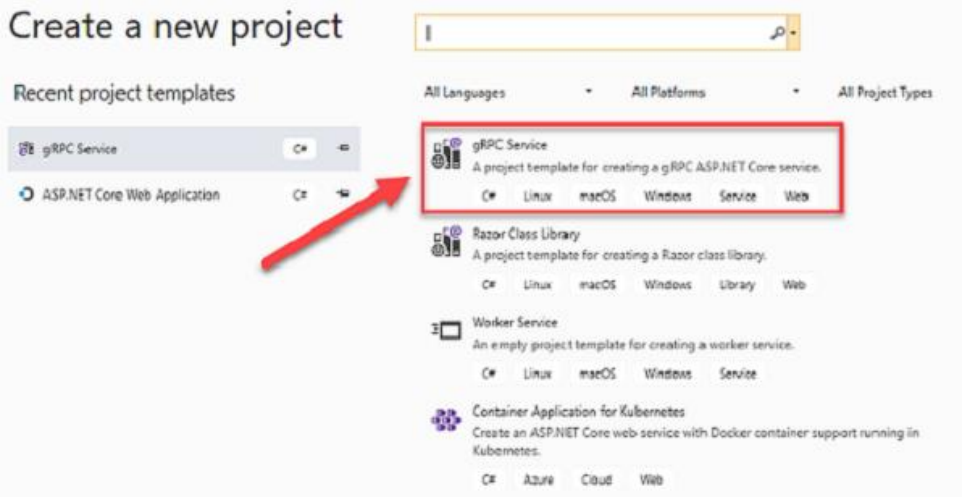


Figure 4-20. gRPC support in Visual Studio 2019

Figure 4-21 shows the skeleton gRPC service generated from the built-in scaffolding included in Visual Studio 2019.

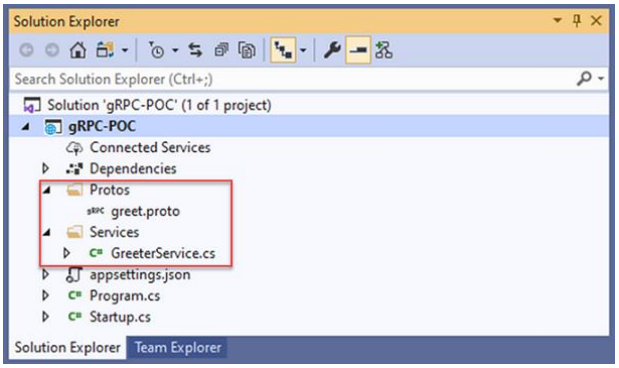


Figure 4-21. gRPC project in Visual Studio 2019

In the previous figure, note the proto description file and service code. As you'll see shortly, Visual Studio generates additional configuration in both the Startup class and underlying project file.

gRPC usage

Favor gRPC for the following scenarios:

- Synchronous backend microservice-to-microservice communication where an immediate response is required to continue processing.
- Polyglot environments that need to support mixed programming platforms.
- Low latency and high throughput communication where performance is critical.
- Point-to-point real-time communication - gRPC can push messages in real time without polling and has excellent support for bi-directional streaming.
- Network constrained environments – binary gRPC messages are always smaller than an equivalent text-based JSON message.

At the time, of this writing, gRPC is primarily used with backend services. Modern browsers can't provide the level of HTTP/2 control required to support a front-end gRPC client. That said, there's support for gRPC-Web with .NET that enables gRPC communication from browser-based apps built with JavaScript or Blazor WebAssembly technologies. gRPC-Web enables an ASP.NET Core gRPC app to support gRPC features in browser apps:

- Strongly typed, code-generated clients
- Compact Protobuf messages
- Server streaming

gRPC implementation

The microservice reference architecture, eShop on Containers, from Microsoft, shows how to implement gRPC services in .NET applications. Figure 4-22 presents the back-end architecture.

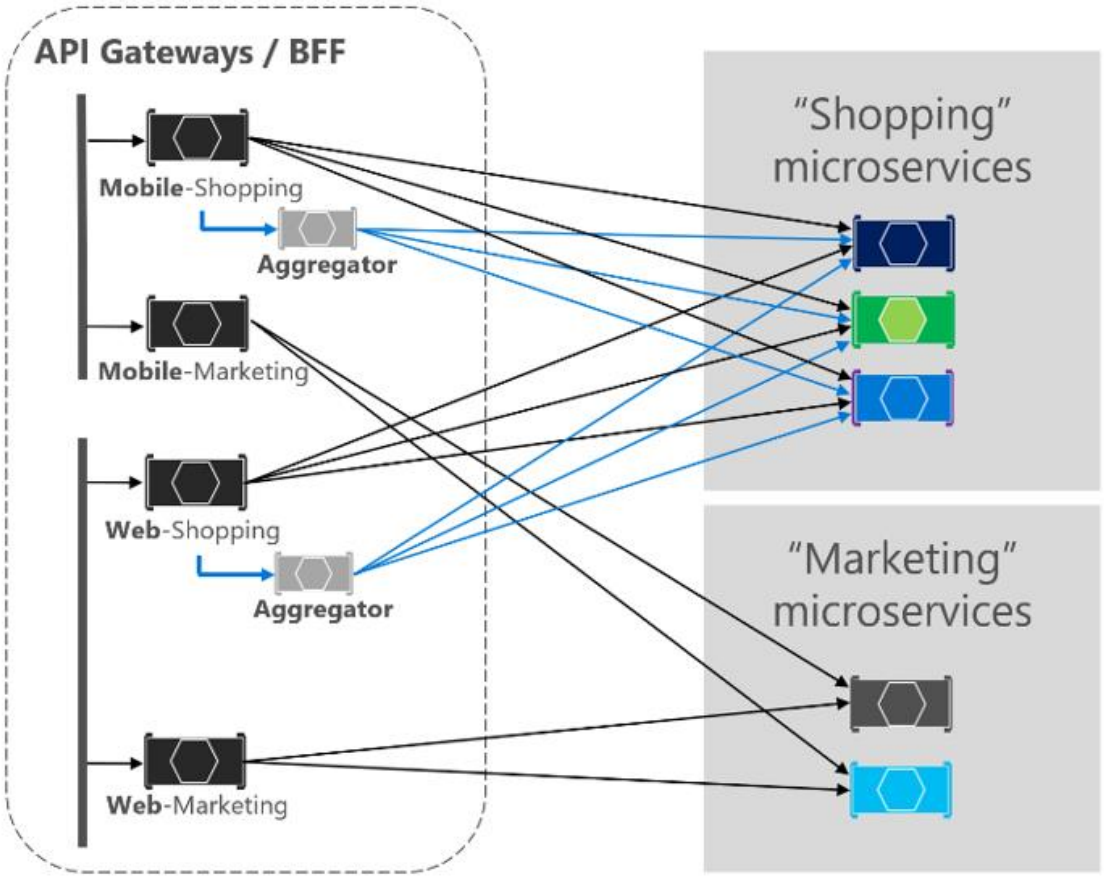


Figure 4-22. Backend architecture for eShop on Containers

In the previous figure, note how eShop embraces the Backend for Frontends pattern (BFF) by exposing multiple API gateways. We discussed the BFF pattern earlier in this chapter. Pay close attention to the Aggregator microservice (in gray) that sits between the Web-Shopping API Gateway and backend Shopping microservices. The Aggregator receives a single request from a client,

dispatches it to various microservices, aggregates the results, and sends them back to the requesting client. Such operations typically require synchronous communication as to produce an immediate response. In eShop, backend calls from the Aggregator are performed using gRPC as shown in Figure 4-23.

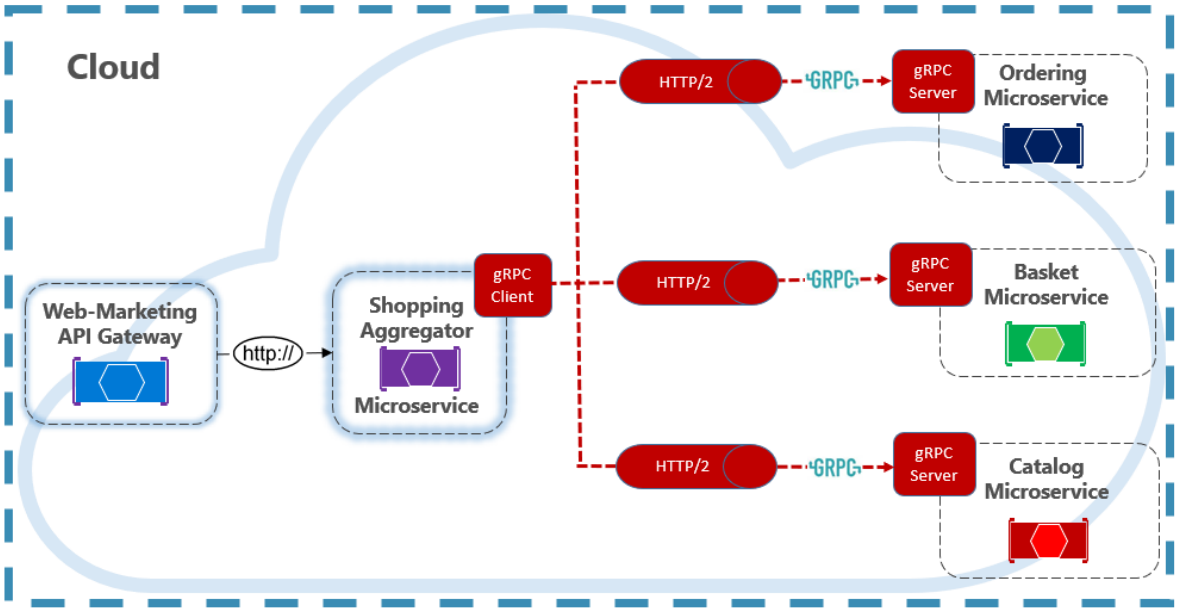


Figure 4-23. gRPC in eShop on Containers

gRPC communication requires both client and server components. In the previous figure, note how the Shopping Aggregator implements a gRPC client. The client makes synchronous gRPC calls (in red) to backend microservices, each of which implement a gRPC server. Both the client and server take advantage of the built-in gRPC plumbing from the .NET SDK. Client-side *stubs* provide the plumbing to invoke remote gRPC calls. Server-side components provide gRPC plumbing that custom service classes can inherit and consume.

Microservices that expose both a RESTful API and gRPC communication require multiple endpoints to manage traffic. You would open an endpoint that listens for HTTP traffic for the RESTful calls and another for gRPC calls. The gRPC endpoint must be configured for the HTTP/2 protocol that is required for gRPC communication.

While we strive to decouple microservices with asynchronous communication patterns, some operations require direct calls. gRPC should be the primary choice for direct synchronous communication between microservices. Its high-performance communication protocol, based on HTTP/2 and protocol buffers, make it a perfect choice.

Looking ahead

Looking ahead, gRPC will continue to gain traction for cloud-native systems. The performance benefits and ease of development are compelling. However, REST will likely be around for a long time. It 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 4-24.

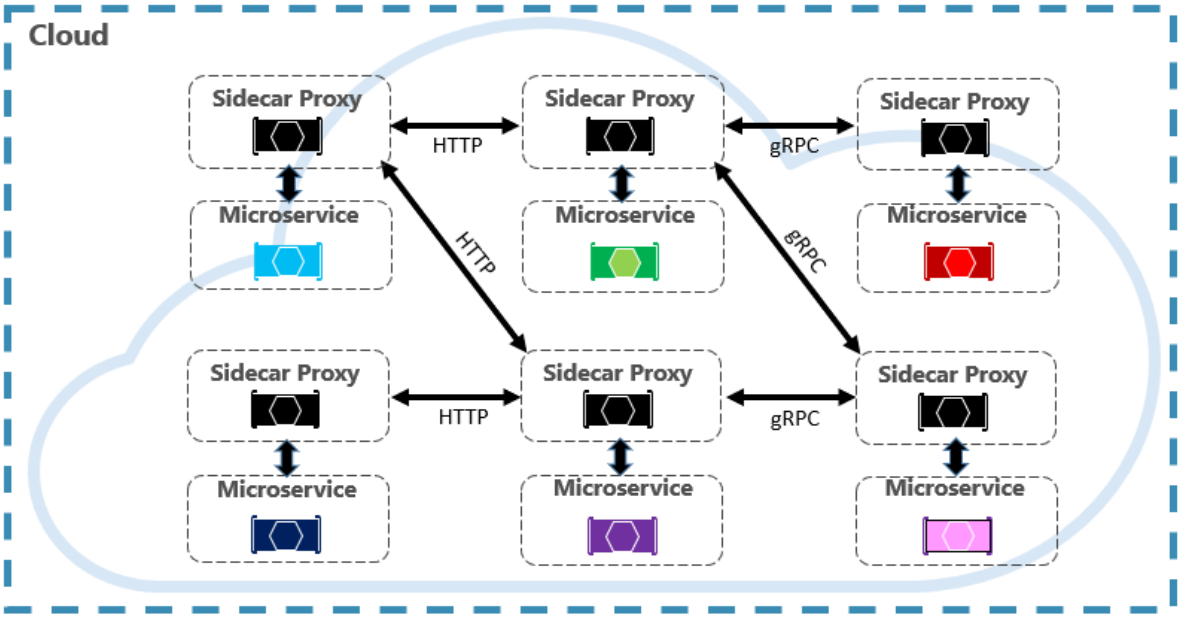


Figure 4-24. Service mesh with a side car

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)
- gRPC Documentation
- gRPC for WCF Developers
- Comparing gRPC Services with HTTP APIs
- Building gRPC Services with .NET video

Distributed data

As we've seen throughout this book, a cloud-native approach changes the way you design, deploy, and manage applications. It also changes the way you manage and store data.

Figure 5-1 contrasts the differences.

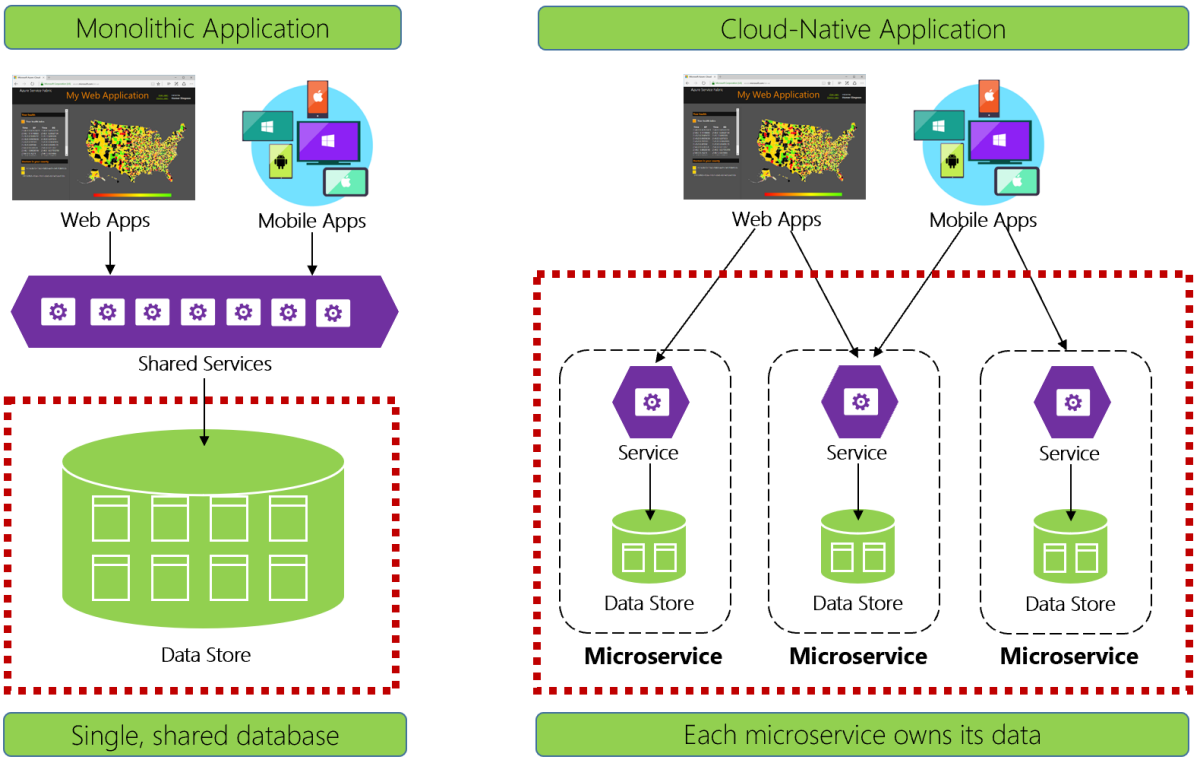


Figure 5-1. Data management in cloud-native applications

Experienced developers will easily recognize the architecture on the left-side of figure 5-1. In this *monolithic application*, business service components collocate together in a shared services tier, sharing data from a single relational database.

In many ways, a single database keeps data management simple. Querying data across multiple tables is straightforward. Changes to data update together or they all rollback. ACID transactions guarantee strong and immediate consistency.

Designing for cloud-native, we take a different approach. On the right-side of Figure 5-1, note how business functionality segregates into small, independent microservices. Each microservice encapsulates a specific business capability and its own data. The monolithic database decomposes

into a distributed data model with many smaller databases, each aligning with a microservice. When the smoke clears, we emerge with a design that exposes a *database per microservice*.

Database-per-microservice, why?

This database per microservice provides many benefits, especially for systems that must evolve rapidly and support massive scale. With this model…

- Domain data is encapsulated within the service
- Data schema can evolve without directly impacting other services
- Each data store can independently scale
- A data store failure in one service won't directly impact other services

Segregating data also enables each microservice to implement the data store type that is best optimized for its workload, storage needs, and read/write patterns. Choices include relational, document, key-value, and even graph-based data stores.

Figure 5-2 presents the principle of polyglot persistence in a cloud-native system.

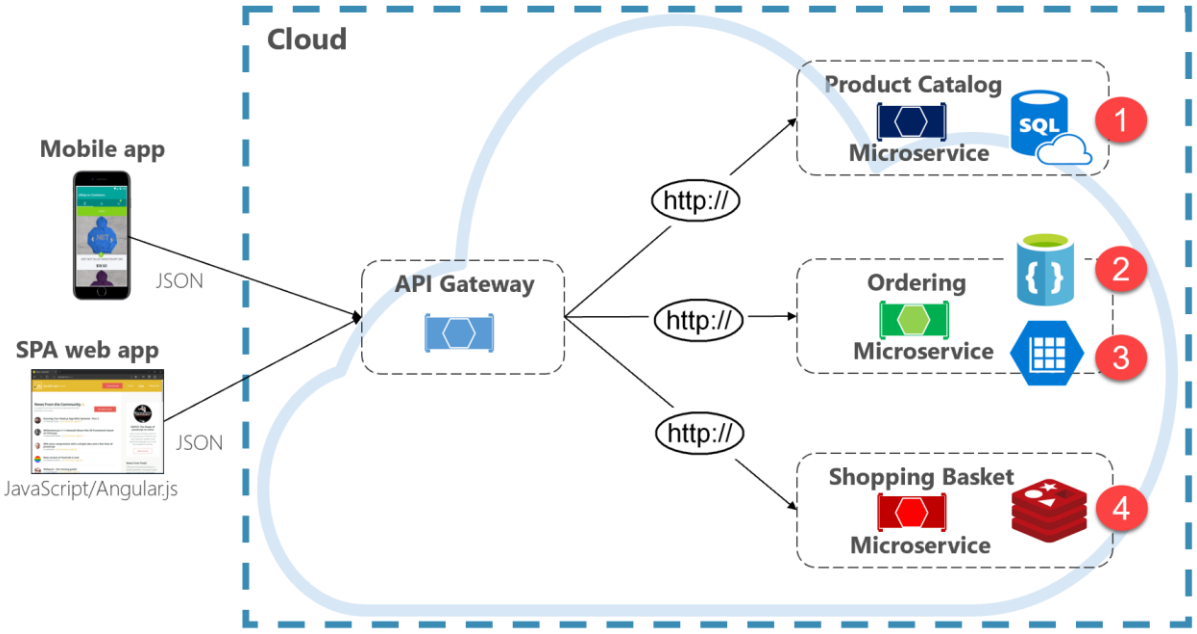


Figure 5-2. Polyglot data persistence

Note in the previous figure how each microservice supports a different type of data store.

- The product catalog microservice consumes a relational database to accommodate the rich relational structure of its underlying data.
- The shopping cart microservice consumes a distributed cache that supports its simple, key-value data store.
- The ordering microservice consumes both a NoSql document database for write operations along with a highly denormalized key/value store to accommodate high-volumes of read operations.

While relational databases remain relevant for microservices with complex data, NoSQL databases have gained considerable popularity. They provide massive scale 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. We cover NoSQL databases later in this chapter.

While encapsulating data into separate microservices can increase agility, performance, and scalability, it also presents many challenges. In the next section, we discuss these challenges along with patterns and practices to help overcome them.

Cross-service queries

While microservices are independent and focus on specific functional capabilities, like inventory, shipping, or ordering, they frequently require integration with other microservices. Often the integration involves one microservice *querying* another for data. Figure 5-3 shows the scenario.

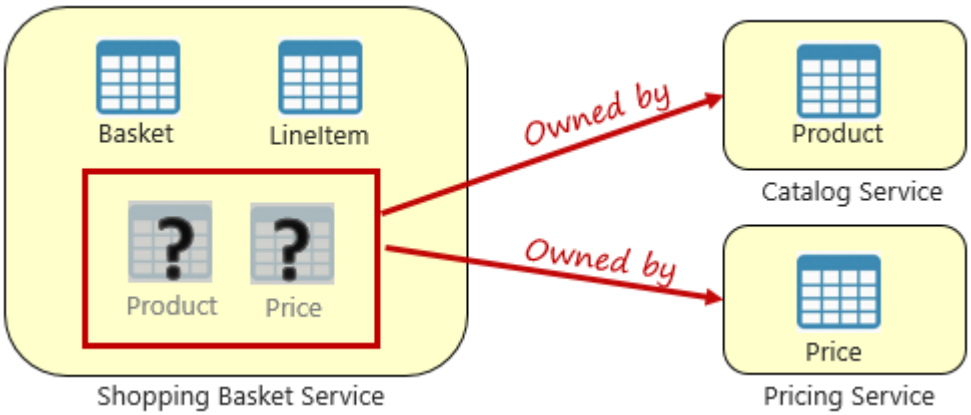


Figure 5-3. Querying across microservices

In the preceding figure, we see a shopping basket microservice that adds an item to a user's shopping basket. While the data store for this microservice contains basket and line item data, it doesn't maintain product or pricing data. Instead, those data items are owned by the catalog and pricing microservices. This aspect presents a problem. How can the shopping basket microservice add a product to the user's shopping basket when it doesn't have product nor pricing data in its database?

One option discussed in Chapter 4 is a direct HTTP call from the shopping basket to the catalog and pricing microservices. However, in chapter 4, we said synchronous HTTP calls *couple* microservices together, reducing their autonomy and diminishing their architectural benefits.

We could also implement a request-reply pattern with separate inbound and outbound queues for each service. However, this pattern is complicated and requires plumbing to correlate request and response messages. While it does decouple the backend microservice calls, the calling service must still synchronously wait for the call to complete. Network congestion, transient faults, or an overloaded microservice and can result in long-running and even failed operations.

Instead, a widely accepted pattern for removing cross-service dependencies is the Materialized View Pattern, shown in Figure 5-4.

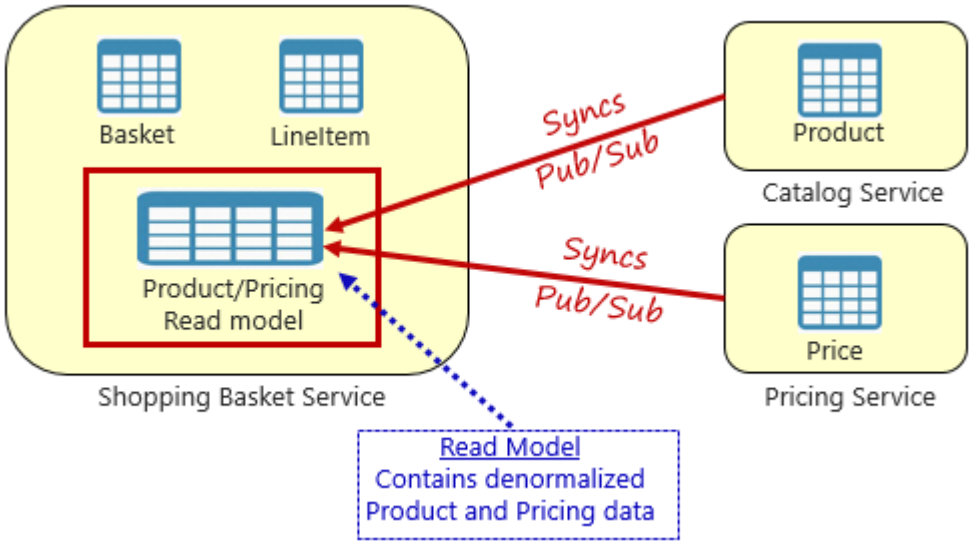


Figure 5-4. Materialized View Pattern

With this pattern, you place a local data table (known as a *read model*) in the shopping basket service. This table contains a denormalized copy of the data needed from the product and pricing microservices. Copying the data directly into the shopping basket microservice eliminates the need for expensive cross-service calls. With the data local to the service, you improve the service's response time and reliability. Additionally, having its own copy of the data makes the shopping basket service more resilient. If the catalog service should become unavailable, it wouldn't directly impact the shopping basket service. The shopping basket can continue operating with the data from its own store.

The catch with this approach is that you now have duplicate data in your system. However, *strategically* duplicating data in cloud-native systems is an established practice and not considered an anti-pattern, or bad practice. Keep in mind that *one and only one service* can own a data set and have authority over it. You'll need to synchronize the read models when the system of record is updated. Synchronization is typically implemented via asynchronous messaging with a publish/subscribe pattern, as shown in Figure 5.4.

Distributed transactions

While querying data across microservices is difficult, implementing a transaction across several microservices is even more complex. The inherent challenge of maintaining data consistency across independent data sources in different microservices can't be understated. The lack of distributed transactions in cloud-native applications means that you must manage distributed transactions programmatically. You move from a world of *immediate consistency* to that of *eventual consistency*.

Figure 5-5 shows the problem.

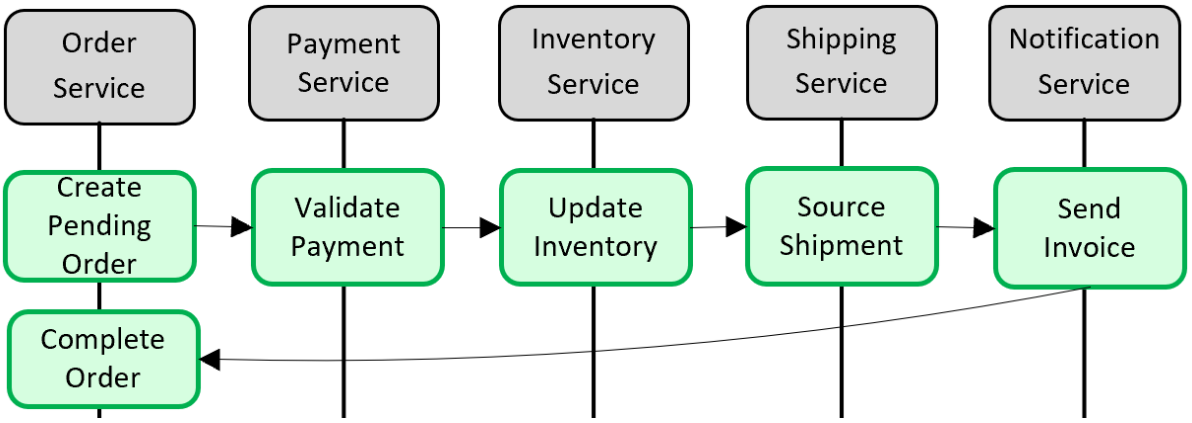


Figure 5-5. Implementing a transaction across microservices

In the preceding figure, five independent microservices participate in a distributed transaction that creates an order. Each microservice maintains its own data store and implements a local transaction for its store. To create the order, the local transaction for *each* individual microservice 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 that would span across all five services to keep data consistent.

Instead, you must construct this distributed transaction *programmatically*.

A popular pattern for adding distributed transactional support is the Saga pattern. It's implemented by grouping local transactions together programmatically and sequentially invoking each one. If any of the local transactions fail, the Saga aborts the operation and invokes a set of compensating transactions. The compensating transactions undo the changes made by the preceding local transactions and restore data consistency. Figure 5-6 shows a failed transaction with the Saga pattern.

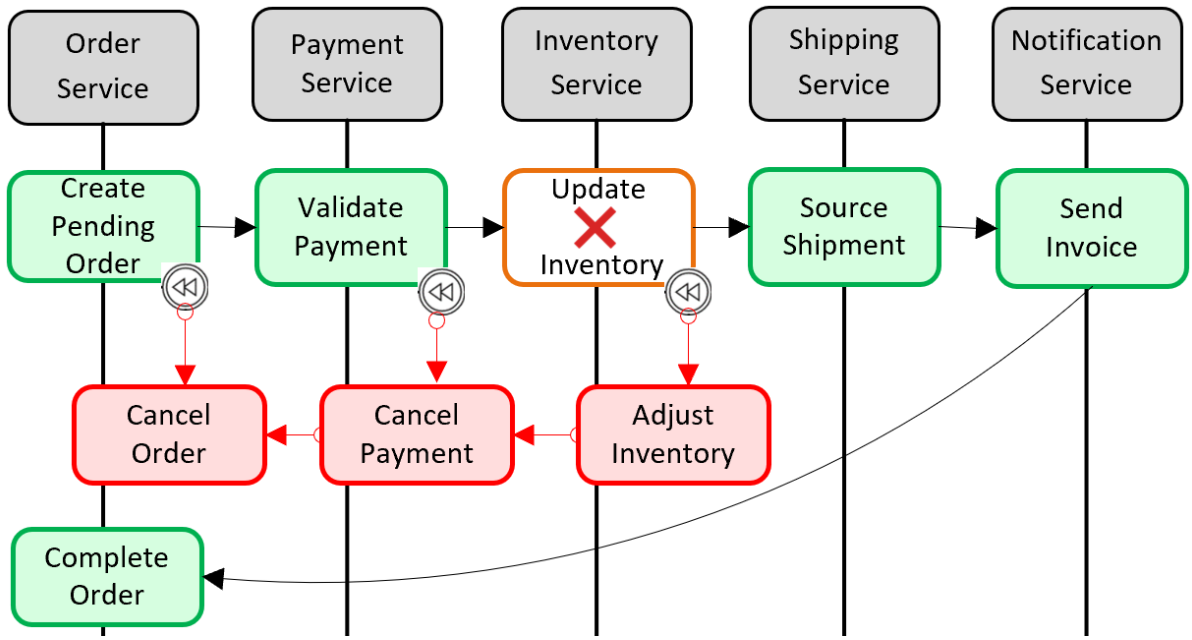


Figure 5-6. Rolling back a transaction

In the previous figure, the *Update Inventory* operation has failed in the Inventory microservice. The Saga invokes a set of compensating transactions (in red) to adjust the inventory counts, cancel the payment and the order, and return 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. In Chapter 4, we discussed the service aggregator pattern that would be the foundation for an orchestrated saga implementation. We also discussed eventing along with Azure Service Bus and Azure Event Grid topics that would be a foundation for a choreographed saga implementation.

High volume data

Large cloud-native applications often support high-volume data requirements. In these scenarios, traditional data storage techniques can cause bottlenecks. For complex systems that deploy on a large scale, both Command and Query Responsibility Segregation (CQRS) and Event Sourcing may improve application performance.

CQRS

CQRS, is an architectural pattern that can help maximize performance, scalability, and security. The pattern separates operations that read data from those operations that write data.

For normal scenarios, the same entity model and data repository object are used for *both* read and write operations.

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

Figure 5-7 shows an implementation of the CQRS pattern.

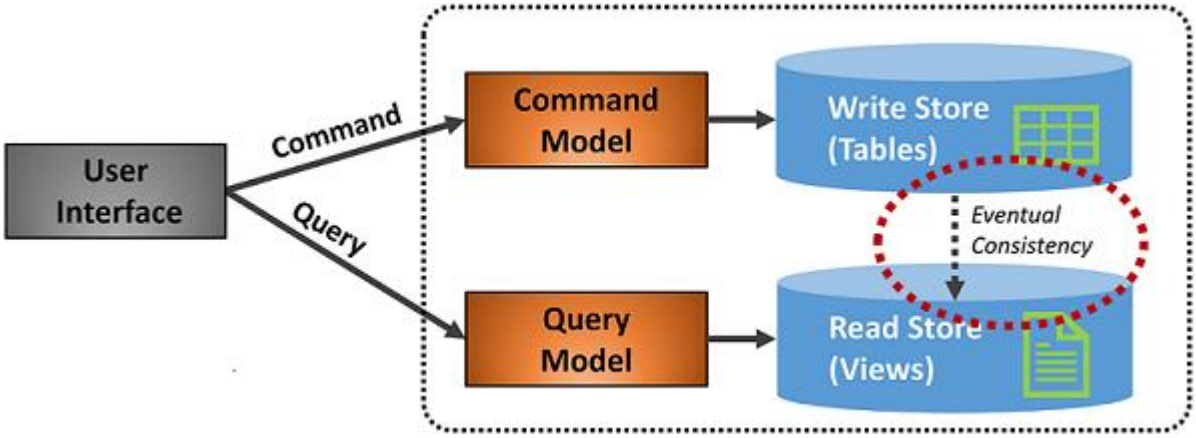


Figure 5-7. CQRS implementation

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

This separation enables reads and writes to scale independently. Read operations use a schema optimized for queries, while the writes use a schema optimized for updates. Read queries go against denormalized data, while complex business logic can be applied to the write model. As well, you might impose tighter security on write operations than those exposing reads.

Implementing CQRS can improve application performance for cloud-native services. However, it does result in a more complex design. Apply this principle carefully and strategically to those sections of your cloud-native application that will benefit from it. For more on CQRS, see the Microsoft book .NET Microservices: Architecture for Containerized .NET Applications.

Event sourcing

Another approach to optimizing high volume data scenarios involves Event Sourcing.

A system typically stores the current state of a data entity. If a user changes their phone number, for example, the customer record is updated with the new number. We always know the current state of a data entity, but each update overwrites the previous state.

In most cases, this model works fine. In high volume systems, however, overhead from transactional locking and frequent update operations can impact database performance, responsiveness, and limit scalability.

Event Sourcing takes a different approach to capturing data. Each operation that affects data is persisted to an event store. Instead of updating the state of a data record, we append each change to a sequential list of past events - similar to an accountant's ledger. The Event Store becomes the system of record for the data. It's used to propagate various materialized views within the bounded context of a microservice. Figure 5.8 shows the pattern.

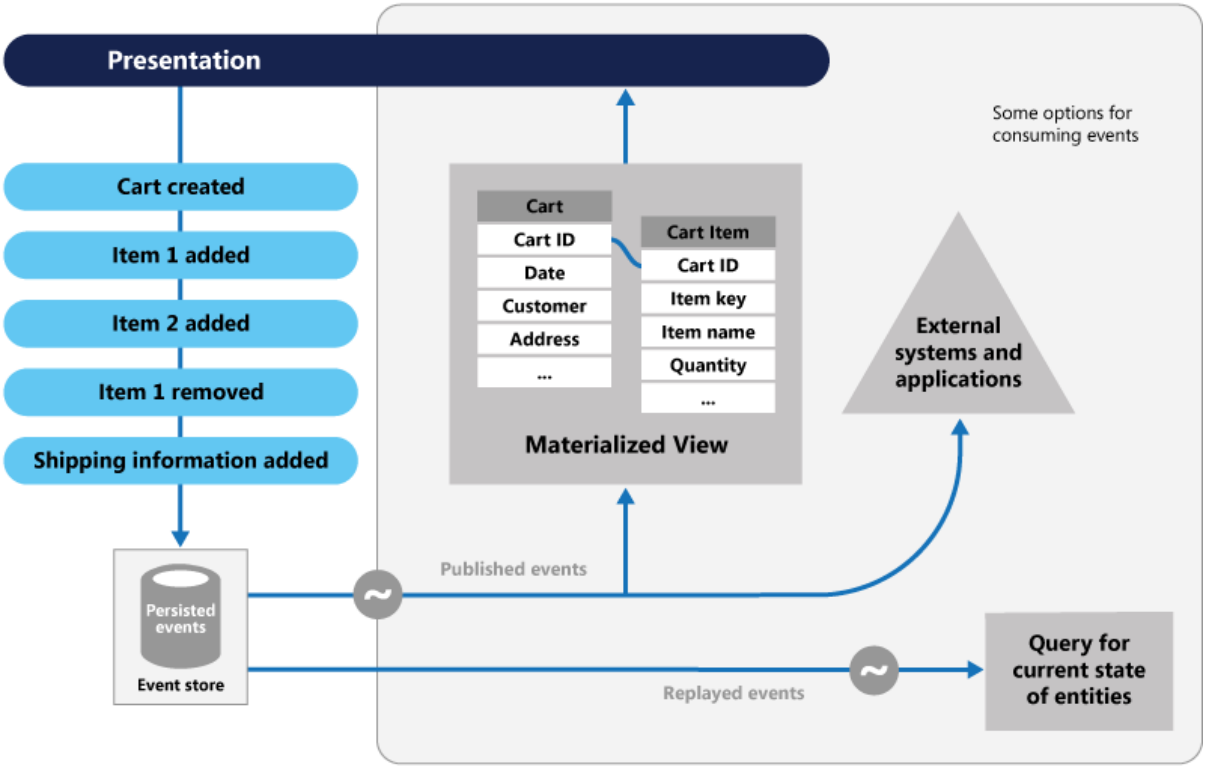


Figure 5-8. Event Sourcing

In the previous figure, note how each entry (in blue) for a user's shopping cart is appended to an underlying event store. In the adjoining materialized view, the system projects the current state by replaying all the events associated with each shopping cart. This view, or read model, is then exposed back to the UI. Events can also be integrated with external systems and applications or queried to determine the current state of an entity. With this approach, you maintain history. You know not only the current state of an entity, but also how you reached this state.

Mechanically speaking, event sourcing simplifies the write model. There are no updates or deletes. Appending each data entry as an immutable event minimizes contention, locking, and concurrency conflicts associated with relational databases. Building read models with the materialized view pattern enables you to decouple the view from the write model and choose the best data store to optimize the needs of your application UI.

For this pattern, consider a data store that directly supports event sourcing. Azure Cosmos DB, MongoDB, Cassandra, CouchDB, and RavenDB are good candidates.

As with all patterns and technologies, implement strategically and when needed. While event sourcing can provide increased performance and scalability, it comes at the expense of complexity and a learning curve.

Relational vs. NoSQL data

Relational and NoSQL are two types of database systems commonly implemented in cloud-native apps. They're built differently, store data differently, and accessed differently. In this section, we'll look at both. Later in this chapter, we'll look at an emerging database technology called *NewSQL*.

Relational databases have been a prevalent technology for decades. They're mature, proven, and widely implemented. Competing database products, tooling, and expertise abound. Relational databases provide a store of related data tables. These tables have a fixed schema, use SQL (Structured Query Language) to manage data, and support ACID guarantees.

No-SQL databases 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 unstructured or semi-structured data, often in key-value pairs or JSON documents. No-SQL databases typically don't provide ACID guarantees beyond the scope of a single database partition. High volume services that require sub second response time favor NoSQL datastores.

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

NoSQL databases include several different models for accessing and managing data, each suited to specific use cases. Figure 5-9 presents four common models.

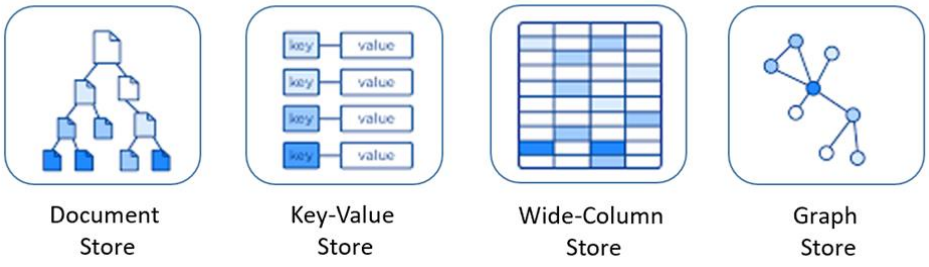


Figure 5-9: Data models for NoSQL databases

Model	Characteristics
Document Store	Data and metadata are stored hierarchically in JSON-based documents inside the database.
Key Value Store	The simplest of the NoSQL databases, data is represented as a collection of key-value pairs.
Wide-Column Store	Related data is stored as a set of nested-key/value pairs within a single column.
Graph Store	Data is stored in a graph structure as node, edge, and data properties.

The CAP theorem

As a way to understand the differences between these types of databases, consider the CAP theorem, a set of principles applied to distributed systems that store state. Figure 5-10 shows the three properties of the CAP theorem.

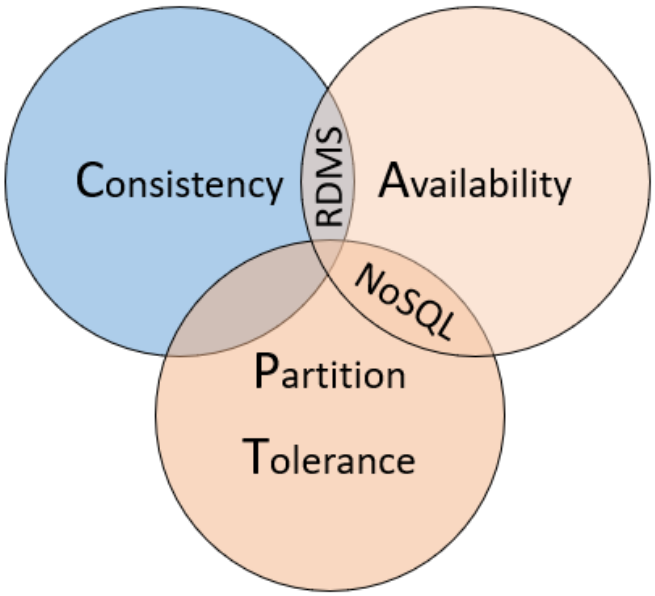


Figure 5-10. The CAP theorem

The theorem states that distributed data systems 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 responds with the most recent data, even if the system must block the request until all replicas update. If you query a "consistent system" for an item that is currently updating, you'll wait for that response until all replicas successfully update. However, you'll receive the most current data.
- *Availability*. Every node returns an immediate response, even if that response isn't the most recent data. If you query an "available system" for an item that is updating, you'll get the best possible answer the service can provide at that moment.
- *Partition Tolerance*. Guarantees the system continues to operate even if a replicated data node fails or loses connectivity with other replicated data nodes.

CAP theorem explains the tradeoffs associated with managing consistency and availability during a network partition; however tradeoffs with respect to consistency and performance also exist with the absence of a network partition. CAP theorem is often further extended to PACELC to explain the tradeoffs more comprehensively.

Relational databases typically provide consistency and availability, but not partition tolerance. They're typically provisioned to a single server and scale vertically by adding more resources to the machine.

Many relational database systems support built-in replication features where copies of the primary database can be made to other secondary server instances. Write operations are made to the primary instance and replicated to each of the secondaries. Upon a failure, the primary instance can fail over to a secondary to provide high availability. Secondaries can also be used to distribute read operations. While writes operations always go against the primary replica, read operations can be routed to any of the secondaries to reduce system load.

Data can also be horizontally partitioned across multiple nodes, such as with sharding. But, sharding dramatically increases operational overhead by spitting data across many pieces that cannot easily communicate. It can be costly and time consuming to manage. Relational features that include table joins, transactions, and referential integrity require steep performance penalties in sharded deployments.

Replication consistency and recovery point objectives can be tuned by configuring whether replication occurs synchronously or asynchronously. If data replicas were to lose network connectivity in a "highly consistent" or synchronous relational database cluster, you wouldn't be able to write to the database. The system would reject the write operation as it can't replicate that change to the other data replica. Every data replica has to update before the transaction can complete.

NoSQL databases typically support high availability and partition tolerance. They scale out horizontally, often across commodity servers. This approach provides tremendous availability, both within and across geographical regions at a reduced cost. You partition and replicate data across these machines, or nodes, providing redundancy and fault tolerance. Consistency is typically tuned through consensus protocols or quorum mechanisms. They provide more control when navigating tradeoffs between tuning synchronous versus asynchronous replication in relational systems.

If data replicas were to lose connectivity in a "highly available" NoSQL database cluster, you could still complete a write operation to the database. The database cluster would allow the write operation and update each data replica as it becomes available. NoSQL databases that support multiple writable replicas can further strengthen high availability by avoiding the need for failover when optimizing recovery time objective.

Modern NoSQL databases typically implement partitioning capabilities as a feature of their system design. Partition management is often built-in to the database, and routing is achieved through placement hints - often called partition keys. A flexible data models enables the NoSQL databases to lower the burden of schema management and improve availablity when deploying application updates that require data model changes.

High availability and massive scalability are often more critical to the business than relational table joins and referential integrity. Developers can implement techniques and patterns such as Sagas, CQRS, and asynchronous messaging to embrace eventual consistency.

Nowadays, care must be taken when considering the CAP theorem constraints. A new type of database, called NewSQL, has emerged which extends the relational database engine to support both horizontal scalability and the scalable performance of NoSQL systems.

Considerations for relational vs. NoSQL systems

Based upon specific data requirements, a cloud-native-based microservice can implement a relational, NoSQL datastore or both.

Consider a NoSQL datastore when:	Consider a relational database when:
You have high volume workloads that require predictable latency at large scale (e.g. latency measured in milliseconds while performing millions of transactions per second)	Your workload volume generally fits within thousands of transactions per second
Your data is dynamic and frequently changes	Your data is highly structured and requires referential integrity
Relationships can be de-normalized data models	Relationships are expressed through table joins on normalized data models
Data retrieval is simple and expressed without table joins	You work with complex queries and reports
Data is typically replicated across geographies and requires finer control over consistency, availablity, and performance	Data is typically centralized, or can be replicated regions asynchronously
Your application will be deployed to commodity hardware, such as with public clouds	Your application will be deployed to large, high-end hardware

In the next sections, we'll explore the options available in the Azure cloud for storing and managing your cloud-native data.

Database as a Service

To start, you could provision an Azure virtual machine and install your database of choice for each service. While you'd have full control over the environment, you'd forgo many built-in features of the cloud platform. You'd also be responsible for managing the virtual machine and database for each service. This approach could quickly become time-consuming and expensive.

Instead, cloud-native applications favor data services exposed as a Database as a Service (DBaaS). Fully managed by a cloud vendor, these services provide built-in security, scalability, and monitoring. Instead of owning the service, you simply consume it as a backing service. The provider operates the resource at scale and bears the responsibility for performance and maintenance.

They can be configured across cloud availability zones and regions to achieve high availability. They all support just-in-time capacity and a pay-as-you-go model. Azure features different kinds of managed data service options, each with specific benefits.

We'll first look at relational DBaaS services available in Azure. You'll see that Microsoft's flagship SQL Server database is available along with several open-source options. Then, we'll talk about the NoSQL data services in Azure.

Azure relational databases

For cloud-native microservices that require relational data, Azure offers four managed relational databases as a service (DBaaS) offerings, shown in Figure 5-11.

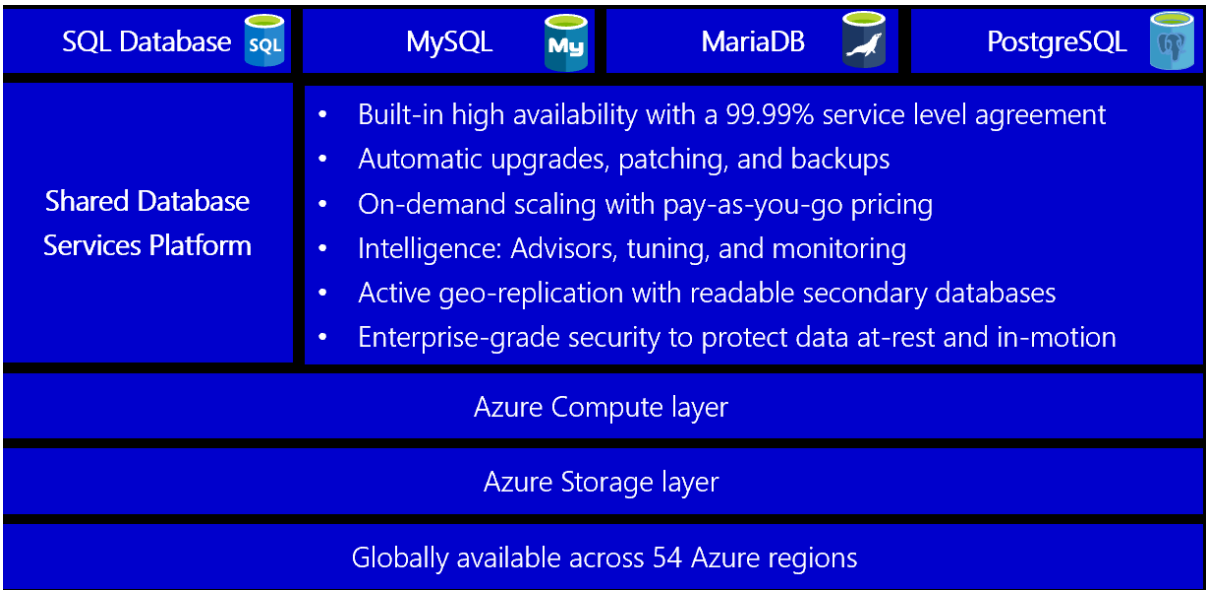


Figure 5-11. Managed relational databases available in Azure

In the previous figure, note how each sits upon a common DBaaS infrastructure which features key capabilities at no additional cost.

These features are especially important to organizations who provision large numbers of databases, but have limited resources to administer them. You can provision an Azure database in minutes by selecting the amount of processing cores, memory, and underlying storage. You can scale the database on-the-fly and dynamically adjust resources with little to no downtime.

Azure SQL Database

Development teams with expertise in Microsoft SQL Server should consider Azure SQL Database. It's a fully managed relational database-as-a-service (DBaaS) based on the Microsoft SQL Server Database Engine. The service shares many features found in the on-premises version of SQL Server and runs the latest stable version of the SQL Server Database Engine.

For use with a cloud-native microservice, Azure SQL Database is available with three deployment options:

- A Single Database represents a fully managed SQL Database running on an Azure SQL Database server in the Azure cloud. The database is considered *contained* as it has no configuration dependencies on the underlying database server.
- A Managed Instance is a fully managed instance of the Microsoft SQL Server Database Engine that provides 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 serverless is a compute tier for a single database that automatically scales based on workload demand. It bills only for the amount of compute used per second. The service is well suited for workloads with intermittent, unpredictable usage patterns, interspersed with periods of inactivity. The serverless compute tier also automatically pauses databases during inactive periods so that only storage charges are billed. It automatically resumes when activity returns.

Beyond the traditional Microsoft SQL Server stack, Azure also features managed versions of three popular open-source databases.

Open-source databases in Azure

Open-source relational databases have become a popular choice for cloud-native applications. Many enterprises favor them over commercial database products, especially for cost savings. Many development teams enjoy their flexibility, community-backed development, and ecosystem of tools and extensions. Open-source databases can be deployed across multiple cloud providers, helping minimize the concern of "vendor lock-in."

Developers can easily self-host any open-source database on an Azure VM. While providing full control, this approach puts you on the hook for the management, monitoring, and maintenance of the database and VM.

However, Microsoft continues its commitment to keeping Azure an "open platform" by offering several popular open-source databases as *fully managed* DBaaS services.

Azure Database for MySQL

MySQL is an open-source relational database and a pillar for applications built on the LAMP software stack. Widely chosen for *read heavy* workloads, it's used by many large organizations, including Facebook, Twitter, and YouTube. The community edition is available for free, while the enterprise edition requires a license purchase. Originally created in 1995, the product was purchased by Sun Microsystems in 2008. Oracle acquired Sun and MySQL in 2010.

Azure Database for MySQL is a managed relational database service based on the open-source MySQL Server engine. It uses the MySQL Community edition. The Azure MySQL server is the administrative point for the service. It's the same MySQL server engine used for on-premises deployments. The engine can create a single database per server or multiple databases per server that share resources. You can continue to manage data using the same open-source tools without having to learn new skills or manage virtual machines.

Azure Database for MariaDB

MariaDB Server is another popular open-source database server. It was created as a *fork* of MySQL when Oracle purchased Sun Microsystems, who owned MySQL. The intent was to ensure that MariaDB remained open-source. As MariaDB is a fork of MySQL, the data and table definitions are compatible, and the client protocols, structures, and APIs, are close-knit.

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

Azure Database for MariaDB is a fully managed relational database as a service in the Azure cloud. The service is based on the MariaDB community edition server engine. It can handle mission-critical workloads with predictable performance and dynamic scalability.

Azure Database for PostgreSQL

PostgreSQL is an open-source relational database with over 30 years of active development. PostgresSQL has a strong reputation for reliability and data integrity. It's feature rich, SQL compliant, and considered more performant than MySQL - especially for workloads with complex queries and heavy writes. 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. The service supports many development platforms, including C++, Java, Python, Node, C#, and PHP. You can migrate PostgreSQL databases to it using the command-line interface tool or Azure Data Migration Service.

Azure Database for PostgreSQL is available with two deployment options:

- The Single Server deployment option is a central administrative point for multiple databases to which you can deploy many databases. 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 by *horizontally scaling* a single database across hundreds of nodes to deliver 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.

NoSQL data in Azure

Cosmos DB is a fully managed, globally distributed NoSQL database service in the Azure cloud. It has been adopted by many large companies across the world, including Coca-Cola, Skype, ExxonMobil, and Liberty Mutual.

If your services require fast response from anywhere in the world, high availability, or elastic scalability, Cosmos DB is a great choice. Figure 5-12 shows Cosmos DB.

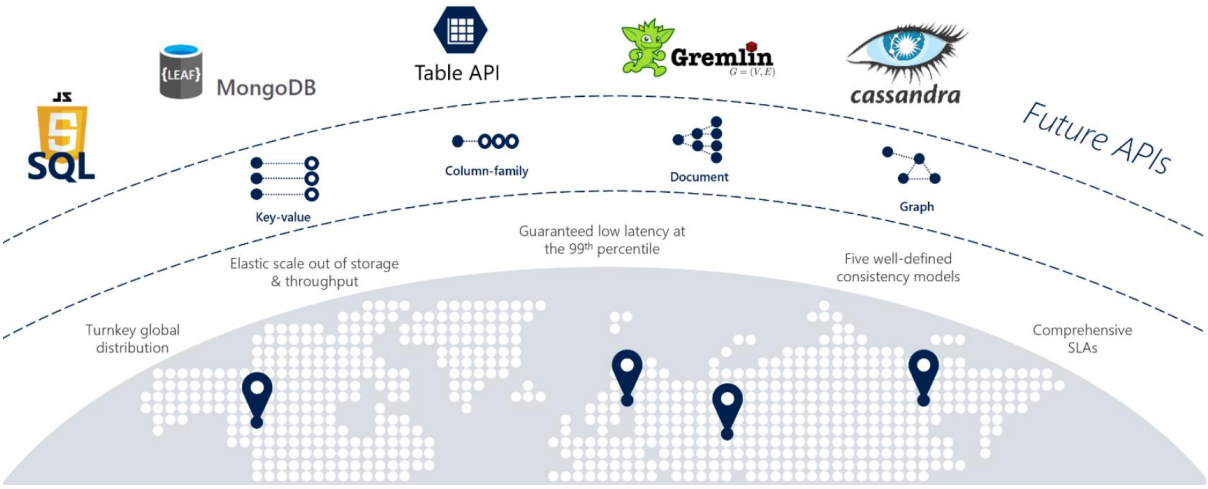


Figure 5-12: Overview of Azure Cosmos DB

The previous figure presents many of the built-in cloud-native capabilities available in Cosmos DB. In this section, we'll take a closer look at them.

Global support

Cloud-native applications often have a global audience and require global scale.

You can distribute Cosmos databases across regions or around 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 services. In the background, Cosmos DB transparently replicates the data to each of the configured regions.

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

The Multi-Master protocol is an important feature in Cosmos DB that 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.

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

Multi-model support

When replatforming monolithic applications to a cloud-native architecture, development teams sometimes have to migrate open-source, NoSQL data stores. Cosmos DB can help you preserve your

investment in these NoSQL datastores with its *multi-model* data platform. The following table shows the supported NoSQL compatibility APIs.

Provider	Description
SQL API	Proprietary API that supports JSON documents and SQL-based queries
Mongo DB API	Supports Mongo DB APIs and JSON documents
Gremlin API	Supports Gremlin API with graph-based nodes and edge data representations
Cassandra API	Supports Casandra API for wide-column data representations
Table API	Supports Azure Table Storage with premium enhancements
etcd API	Enables Cosmos DB as a backing store for Azure Kubernetes Service clusters

Development teams can migrate existing Mongo, Gremlin, or Cassandra databases into Cosmos DB with minimal changes to data or code. For new apps, development teams can choose among open-source options or the built-in SQL API model.

Internally, Cosmos stores the data in a simple struct format made up of primitive data types. For each request, the database engine translates the primitive data into the model representation you've selected.

In the previous table, note the Table API option. This API is an evolution of Azure Table Storage. Both share the same underlying table model, but the Cosmos DB Table API adds premium enhancements not available in the Azure Storage API. The following table contrasts the features.

	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	Unlimited operations per table
Global Distribution	Single region with optional single secondary read region	Turnkey distributions to all regions with automatic failover
Indexing	Available for partition and row key properties only	Automatic indexing of all properties
Pricing	Optimized for cold workloads (low throughput : storage ratio)	Optimizied for hot workloads (high throughput : storage ratio)

Microservices that consume Azure Table storage can easily migrate to the Cosmos DB Table API. No code changes are required.

Tunable consistency

Earlier in the *Relational vs. NoSQL* section, we discussed the subject of *data consistency*. Data consistency refers to the *integrity* of your data. Cloud-native services with distributed data rely on replication and 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 will always return the most current data - even if the system must incur latency waiting for an update to replicate across all database copies. While a database configured for *eventual consistency* will return data immediately, even if that data isn't the most current copy. The latter option enables higher availability, greater scale, and increased performance.

Azure Cosmos DB offers five well-defined consistency models shown in Figure 5-13.

Figure 5-13: Cosmos DB Consistency Levels

These options enable you to make precise choices and granular tradeoffs for consistency, availability, and the performance for your data. The levels are presented in the following table.

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

In the article Getting Behind the 9-Ball: Cosmos DB Consistency Levels Explained, Microsoft Program Manager Jeremy Likness provides an excellent explanation of the five models.

Partitioning

Azure Cosmos DB embraces automatic partitioning to scale a database to meet the performance needs of your cloud-native services.

You manage data in Cosmos DB data by creating databases, containers, and items.

Containers live in a Cosmos DB database and represent a schema-agnostic grouping of items. Items are the data that you add to the container. They're represented as documents, rows, nodes, or edges. All items added to a container are automatically indexed.

To partition the container, items are divided into distinct subsets called logical partitions. Logical partitions are populated based on the value of a partition key that is associated with each item in a container. Figure 5-14 shows two containers each with a logical partition based on a partition key value.

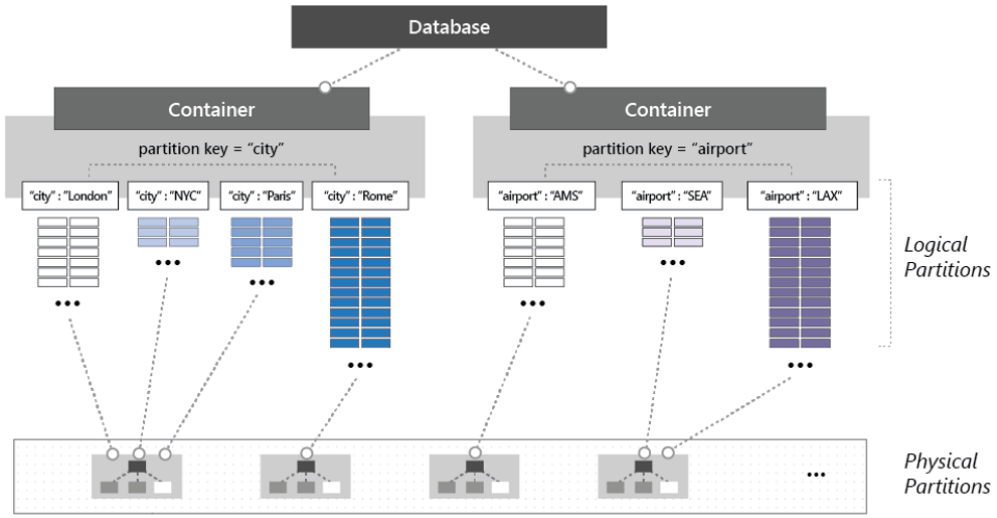


Figure 5-14: Cosmos DB partitioning mechanics

Note in the previous figure how each item includes a partition key of either 'city' or 'airport'. The key determines the item's logical partition. Items with a city code are assigned to the container on the left, and items with an airport code, to the container on the right. Combining the partition key value with the ID value creates an item's index, which uniquely identifies the item.

Internally, Cosmos DB automatically manages the placement of logical partitions on physical partitions to satisfy the scalability and performance needs of the container. As application throughput and storage requirements increase, Azure Cosmos DB redistributes logical partitions across a greater number of servers. Redistribution operations are managed by Cosmos DB and invoked without interruption or downtime.

NewSQL databases

NewSQL is an emerging database technology that combines the distributed scalability of NoSQL with the ACID guarantees of a relational database. NewSQL databases are important for business systems that must process high-volumes of data, across distributed environments, with full transactional support and ACID compliance. While a NoSQL database can provide massive scalability, it does not guarantee data consistency. Intermittent problems from inconsistent data can place a burden on the development team. Developers must construct safeguards into their microservice code to manage problems caused by inconsistent data.

The Cloud Native Computing Foundation (CNCF) features several NewSQL database projects.

Project	Characteristics
Cockroach DB	An ACID-compliant, relational database that scales globally. Add a new node to a cluster and CockroachDB takes care of balancing the data across instances and geographies. It creates, manages, and distributes replicas to ensure reliability. It's open source and freely available.

Project	Characteristics
TiDB	An open-source database that supports Hybrid Transactional and Analytical Processing (HTAP) workloads. It is MySQL-compatible and features horizontal scalability, strong consistency, and high availability. TiDB acts like a MySQL server. You can continue to use existing MySQL client libraries, without requiring extensive code changes to your application.
YugabyteDB	An open source, high-performance, distributed SQL database. It supports low query latency, resilience against failures, and global data distribution. YugabyteDB is PostgressSQL-compatible and handles scale-out RDBMS and internet-scale OLTP workloads. The product also supports NoSQL and is compatible with Cassandra.
Vitess	Vitess is a database solution for deploying, scaling, and managing large clusters of MySQL instances. It can run in a public or private cloud architecture. Vitess combines and extends many important MySQL features and features both vertical and horizontal sharding support. Originated by YouTube, Vitess has been serving all YouTube database traffic since 2011.

The open-source projects in the previous figure are available from the Cloud Native Computing Foundation. Three of the offerings are full database products, which include .NET support. The other, Vitess, is a database clustering system that horizontally scales large clusters of MySQL instances.

A key design goal for NewSQL databases is to work natively in Kubernetes, taking advantage of the platform's resiliency and scalability.

NewSQL databases are designed to thrive in ephemeral cloud environments where underlying virtual machines can be restarted or rescheduled at a moment's notice. The databases are designed to survive node failures without data loss nor downtime. CockroachDB, for example, is able to survive a machine loss by maintaining three consistent replicas of any data across the nodes in a cluster.

Kubernetes uses a *Services construct* to allow a client to address a group of identical NewSQL databases processes from a single DNS entry. By decoupling the database instances from the address of the service with which it's associated, we can scale without disrupting existing application instances. Sending a request to any service at a given time will always yield the same result.

In this scenario, all database instances are equal. There are no primary or secondary relationships. Techniques like *consensus replication* found in CockroachDB allow any database node to handle any request. If the node that receives a load-balanced request has the data it needs locally, it responds immediately. If not, the node becomes a gateway and forwards the request to the appropriate nodes to get the correct answer. From the client's perspective, every database node is the same: They appear as a single *logical* database with the consistency guarantees of a single-machine system, despite having dozens or even hundreds of nodes that are working behind the scenes.

For a detailed look at the mechanics behind NewSQL databases, see the DASH: Four Properties of Kubernetes-Native Databases article.

Data migration to the cloud

One of the more time-consuming tasks is migrating data from one data platform to another. The Azure Data Migration Service can help expedite such efforts. It can migrate data from several external

database sources into Azure Data platforms with minimal downtime. Target platforms include the following services:

- Azure SQL Database
- Azure Database for MySQL
- Azure Database for MariaDB
- Azure Database for PostgreSQL
- Azure Cosmos DB

The service provides recommendations to guide you through the changes required to execute a migration, both small or large.

Caching in a cloud-native app

The benefits of caching are well understood. The technique works by temporarily copying frequently accessed data from a backend data store to *fast storage* that's located closer to the application. Caching is often implemented where…

- Data remains relatively static.
- Data access is slow, especially compared to the speed of the cache.
- Data is subject to high levels of contention.

Why?

As discussed in the Microsoft caching guidance, caching can increase performance, scalability, and availability for individual microservices and the system as a whole. It reduces the latency and contention of handling large volumes of concurrent requests to a data store. As data volume and the number of users increase, the greater the benefits of caching become.

Caching is most effective when a client repeatedly reads data that is immutable or that changes infrequently. Examples include reference information such as product and pricing information, or shared static resources that are costly to construct.

While microservices should be stateless, a distributed cache can support concurrent access to session state data when absolutely required.

Also consider caching to avoid repetitive computations. If an operation transforms data or performs a complicated calculation, cache the result for subsequent requests.

Caching architecture

Cloud native applications typically implement a distributed caching architecture. The cache is hosted as a cloud-based backing service, separate from the microservices. Figure 5-15 shows the architecture.

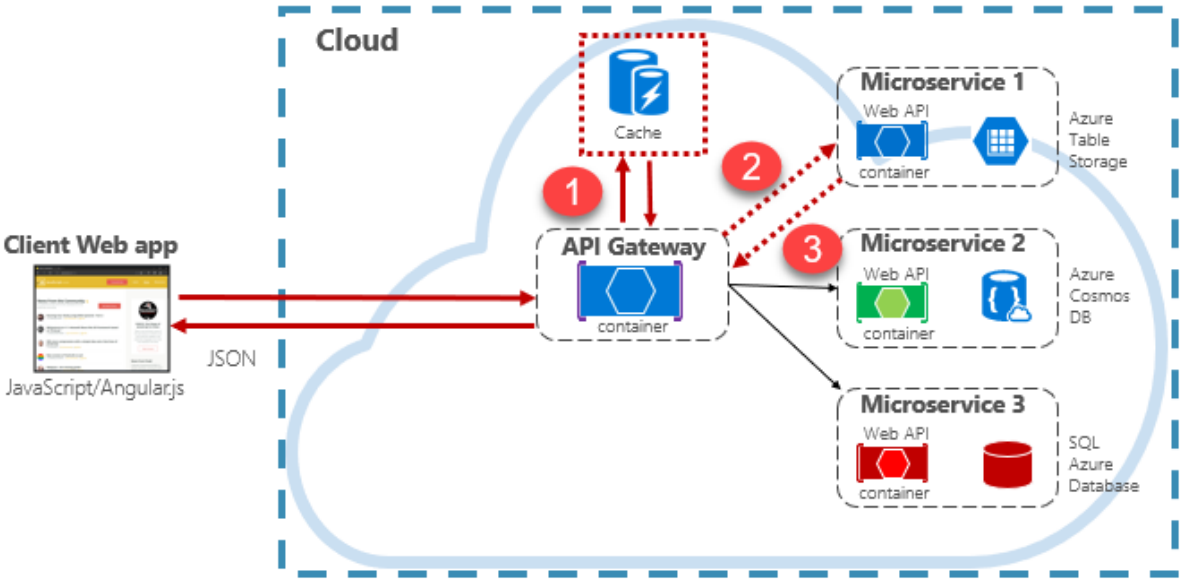


Figure 5-15: Caching in a cloud native app

In the previous figure, note how the cache is independent of and shared by the microservices. In this scenario, the cache is invoked by the API Gateway. As discussed in chapter 4, the gateway serves as a front end for all incoming requests. The distributed cache increases system responsiveness by returning cached data whenever possible. Additionally, separating the cache from the services allows the cache to scale up or out independently to meet increased traffic demands.

The previous figure presents a common caching pattern known as the cache-aside pattern. For an incoming request, you first query the cache (step #1) for a response. 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 a local database in a downstream service (step #2). It's then 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 timely and consistent.

As a shared cache grows, it might prove beneficial to partition its data across multiple nodes. Doing so can help minimize contention and improve scalability. Many Caching services support the ability to dynamically add and remove nodes and rebalance data across partitions. This approach typically involves clustering. Clustering exposes a collection of federated nodes as a seamless, single cache. Internally, however, the data is dispersed across the nodes following a predefined distribution strategy that balances the load evenly.

Azure Cache for Redis

Azure Cache for Redis is a secure data caching and messaging broker service, fully managed by Microsoft. Consumed as a Platform as a Service (PaaS) offering, it provides high throughput and low-latency access to data. The service is accessible to any application within or outside of Azure.

The Azure Cache for Redis service manages access to open-source Redis servers hosted across Azure data centers. The service acts as a facade providing management, access control, and security. The service natively supports a rich set of data structures, including strings, hashes, lists, and sets. If your application already uses Redis, it will work as-is with Azure Cache for Redis.

Azure Cache for Redis is more than a simple cache server. It can support a number of scenarios to enhance a microservices architecture:

- An in-memory data store
- A distributed non-relational database
- A message broker
- A configuration or discovery server

For advanced scenarios, a copy of the cached data can be persisted to disk. If a catastrophic event disables both the primary and replica caches, the cache is reconstructed from the most recent snapshot.

Azure Redis Cache is available across a number of predefined configurations and pricing tiers. The Premium tier features many enterprise-level features such as clustering, data persistence, geo-replication, and virtual-network isolation.

Elasticsearch in a cloud-native app

Elasticsearch is a distributed search and analytics system that enables complex search capabilities across diverse types of data. It's open source and widely popular. Consider how the following companies integrate Elasticsearch into their application:

- Wikipedia for full-text and incremental (search as you type) searching.
- GitHub to index and expose over 8 million code repositories.
- Docker for making its container library discoverable.

Elasticsearch is built on top of the Apache Lucene full-text search engine. Lucene provides high-performance document indexing and querying. It indexes data with an inverted indexing scheme – instead of mapping pages to keywords, it maps keywords to pages just like a glossary at the end of a book. Lucene has powerful query syntax capabilities and can query data by:

- Term (a full word)
- Prefix (starts-with word)
- Wildcard (using "*" or "?" filters)
- Phrase (a sequence of text in a document)

- Boolean value (complex searches combining queries)

While Lucene provides low-level plumbing for searching, Elasticsearch provides the server that sits on top of Lucene. Elasticsearch adds higher-level functionality to simplify working Lucene, including a RESTful API to access Lucene's indexing and searching functionality. It also provides a distributed infrastructure capable of massive scalability, fault tolerance, and high availability.

For larger cloud-native applications with complex search requirements, Elasticsearch is available as managed service in Azure. The Microsoft Azure Marketplace features preconfigured templates which developers can use to deploy an Elasticsearch cluster on Azure.

From the Microsoft Azure Marketplace, developers can use preconfigured templates built to quickly deploy an Elasticsearch cluster on Azure. Using the Azure-managed offering, you can deploy up to 50 data nodes, 20 coordinating nodes, and three dedicated master nodes.

Summary

This chapter presented a detailed look at data in cloud-native systems. We started by contrasting data storage in monolithic applications with data storage patterns in cloud-native systems. We looked at data patterns implemented in cloud-native systems, including cross-service queries, distributed transactions, and patterns to deal with high-volume systems. We contrasted SQL with NoSQL data. We looked at data storage options available in Azure that include both Microsoft-centric and open-source options. Finally, we discussed caching and Elasticsearch in a cloud-native application.

References

- Command and Query Responsibility Segregation (CQRS) pattern
- Event Sourcing pattern
- Why isn't RDBMS Partition Tolerant in CAP Theorem and why is it Available?
- Materialized View
- All you really need to know about open source databases
- Compensating Transaction pattern
- Saga Pattern
- Saga Patterns | How to implement business transactions using microservices
- Compensating Transaction pattern
- Getting Behind the 9-Ball: Cosmos DB Consistency Levels Explained
- Exploring the different types of NoSQL Databases Part II
- On RDBMS, NoSQL and NewSQL databases. Interview with John Ryan
- SQL vs NoSQL vs NewSQL: The Full Comparison
- DASH: Four Properties of Kubernetes-Native Databases

- CockroachDB
- TiDB
- YugabyteDB
- Vitess
- Elasticsearch: The Definitive Guide
- Introduction to Apache Lucene

Cloud-native resiliency

Resiliency is the ability of your system to react to failure and still remain functional. It's not about avoiding failure, but accepting failure and constructing your cloud-native services to respond to it. You want to return to a fully functioning state quickly as possible.

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

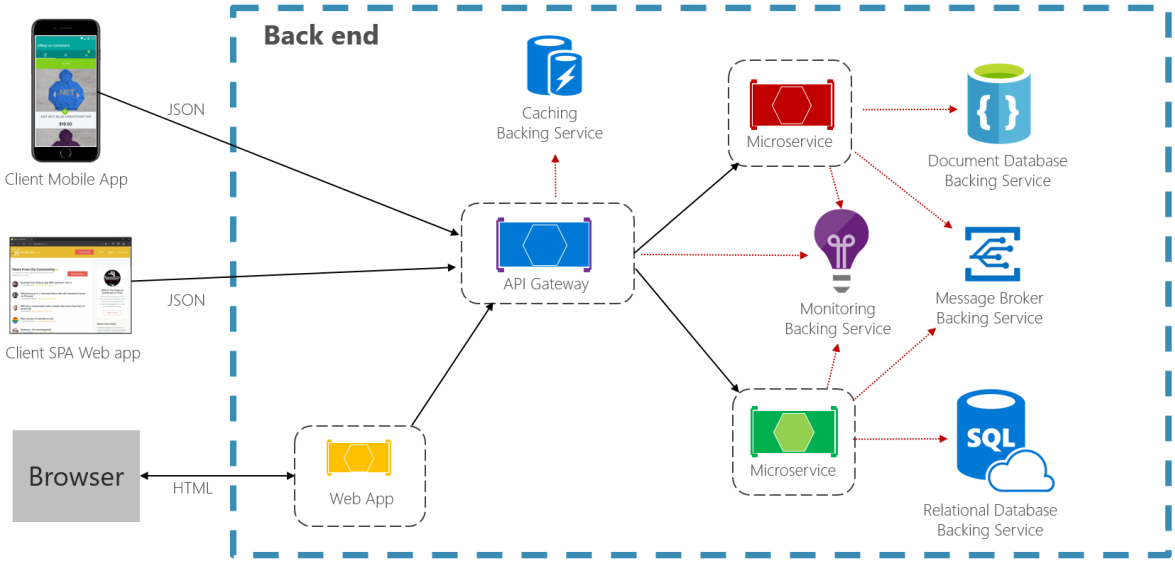


Figure 6-1. Distributed cloud-native environment

In the previous figure, each microservice and cloud-based backing service execute in a separate process, across server infrastructure, communicating via network-based calls.

Operating in this environment, a service must be sensitive to many different challenges:

- Unexpected network latency - the time for a service request to travel to the receiver and back.
- Transient faults - short-lived network connectivity errors.
- Blockage 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 orchestrator operation such as a rolling upgrade or moving a service from one node to another.

- Hardware failures.

Cloud platforms can detect and mitigate many of these infrastructure issues. It may restart, scale out, and even redistribute your service to a different node. However, to take full advantage of this built-in protection, you must design your services to react to it and thrive in this dynamic environment.

In the following sections, we'll explore defensive techniques that your service and managed cloud resources can leverage to minimize downtime and disruption.

Application resiliency patterns

The first line of defense is application resiliency.

While you could invest considerable time writing your own resiliency framework, such products already exist. 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 .NET Framework or .NET 5. The following table describes the resiliency features, called `policies`, available in the Polly Library. They can be applied individually or grouped together.

Policy	Experience
Retry	Configures retry operations on designated operations.
Circuit Breaker	Blocks requested operations for a predefined period when faults exceed a configured threshold
Timeout	Places limit on the duration for which a caller can wait for a response.
Bulkhead	Constrains actions to fixed-size resource pool to prevent failing calls from swamping a resource.
Cache	Stores responses automatically.
Fallback	Defines structured behavior upon a failure.

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

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

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 the .NET platform. 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 mitigate 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 6-2 shows a retry in action.

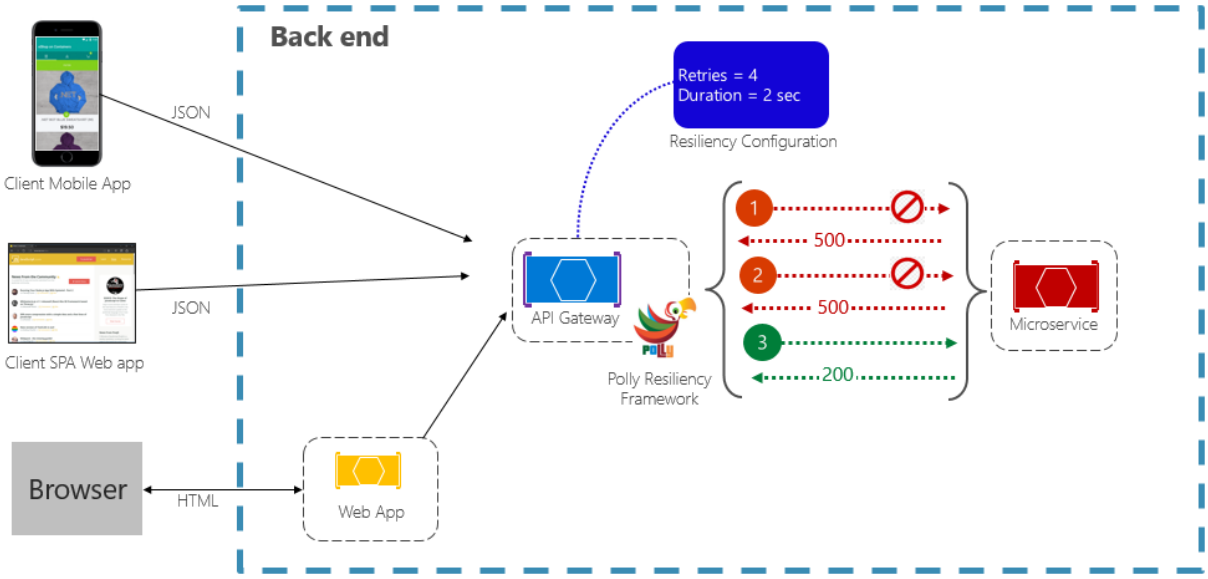


Figure 6-2. Retry pattern in action

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 retries 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.
- Had the 4th retry attempt failed, a fallback policy would be invoked to gracefully handle the problem.

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. After a pre-defined number of failed calls, it blocks all traffic to the service. Periodically, it will allow a trial call to determine whether the fault has resolved. Figure 6-3 shows the Circuit Breaker pattern in action.

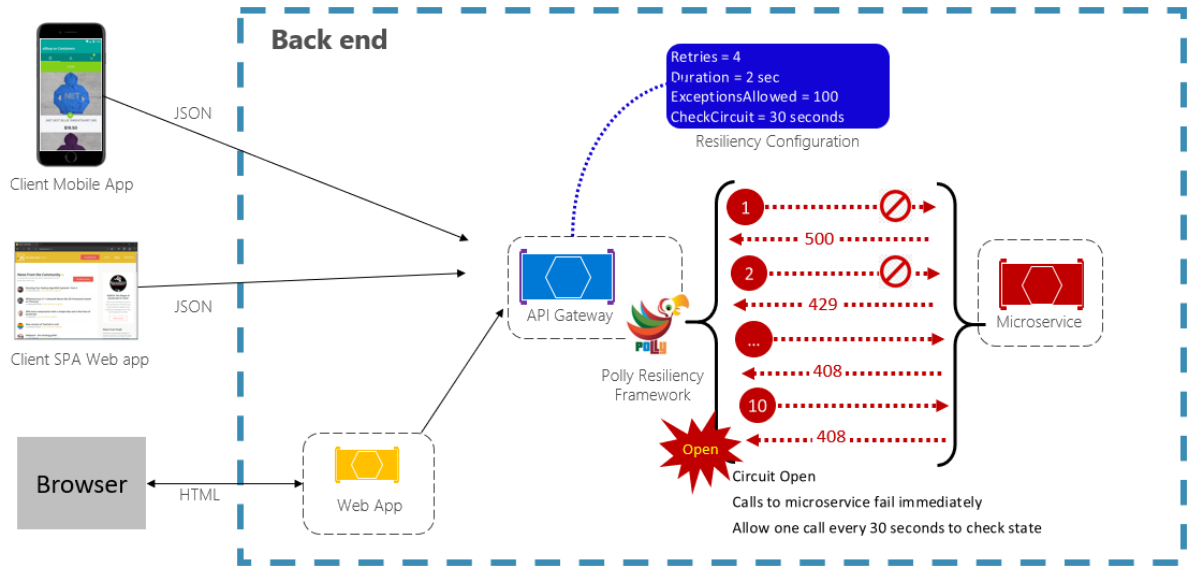


Figure 6-3. Circuit breaker pattern in action

In the previous figure, a Circuit Breaker pattern has been added to the original retry pattern. Note how after 100 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. Typically, an application will *combine* these two patterns by using the Retry pattern to invoke an operation through a circuit breaker.

Testing for resiliency

Testing for resiliency cannot always be done the same way that you test application functionality (by running unit tests, integration tests, and so on). Instead, you must test how the end-to-end workload performs under failure conditions, which only occur intermittently. For example: inject failures by crashing processes, expired certificates, make dependent services unavailable etc. Frameworks like chaos-monkey can be used for such chaos testing.

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 resiliency

We've said resiliency enables your application to react to failure and still remain functional. The whitepaper, Resilience in Azure whitepaper, provides guidance for achieving resilience in the Azure platform. Here are some key recommendations:

- *Hardware failure.* Build redundancy into the application by deploying components across different fault domains. For example, ensure that Azure VMs are placed in different racks by using Availability Sets.

- *Datacenter failure.* Build redundancy into the application with fault isolation zones across datacenters. For example, ensure that Azure VMs are placed in different fault-isolated datacenters by using Azure Availability Zones.
- *Regional failure.* Replicate the data and components into another region so that applications can be quickly recovered. For example, use Azure Site Recovery to replicate Azure VMs to another Azure region.
- *Heavy load.* Load balance across instances to handle spikes in usage. For example, put two or more Azure VMs behind a load balancer to distribute traffic to all VMs.
- *Accidental data deletion or corruption.* Back up data so it can be restored if there's any deletion or corruption. For example, use Azure Backup to periodically back up your Azure VMs.

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 upon your business requirements and will affect both the 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, but 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.
- *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 you deploy your application to a single region, and that 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 6-4:

Figure 6-4. 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 6-5 shows this scenario.

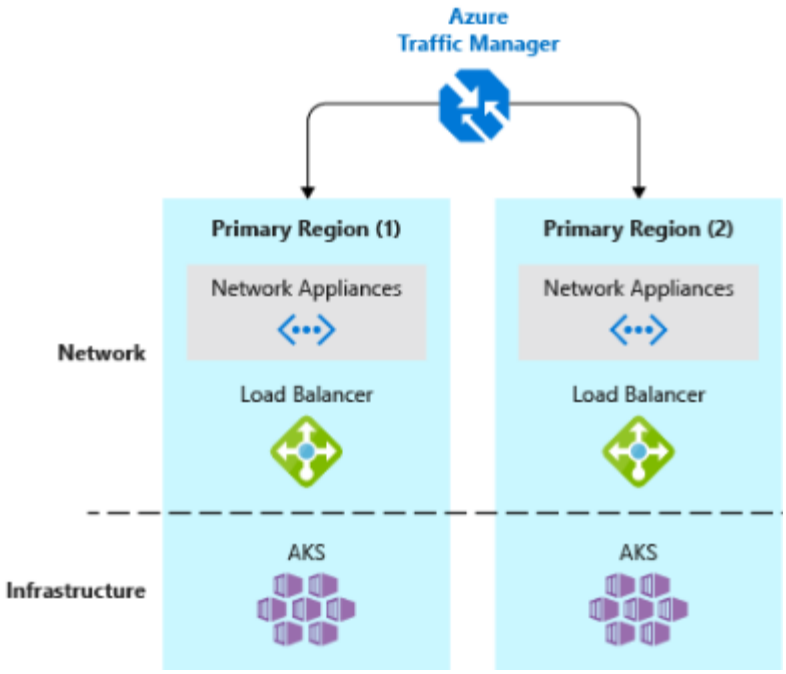


Figure 6-5. AKS and Azure Traffic Manager

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 6-6 shows the two approaches:

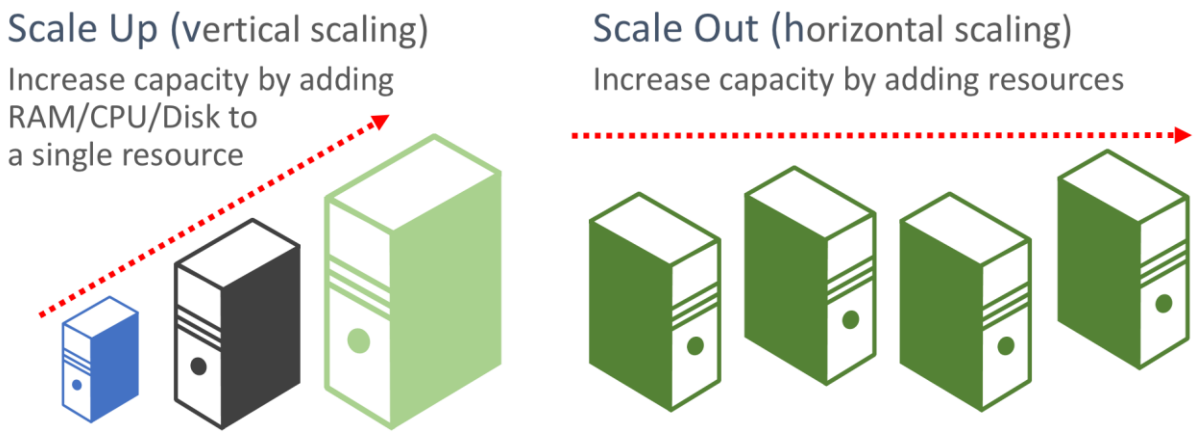


Figure 6-6. Scale up versus scale out

- *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 out aggressively.* A final practice would be to scale out aggressively so that you can quickly meet immediate spikes in traffic without losing business. And, then scale in (that is, remove unneeded instances) 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 embraced a microservice-based architectural approach. While such an architecture provides important benefits, it presents many challenges:

- *Out-of-process network communication.* Each microservice communicates over a network protocol that introduces network congestion, latency, and transient faults.
- *Service discovery.* How do microservices discover and communicate with each other when running across a cluster of machines with their own IP addresses and ports?
- *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 microservice?
- *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 microservices?

You can address these concerns with different libraries and frameworks, but the implementation can be expensive, complex, and time-consuming. You also end up with infrastructure concerns coupled to business logic.

Service mesh

A better approach is an evolving technology entitled *Service Mesh*. A service mesh is a configurable infrastructure layer with built-in capabilities to handle service communication and the other challenges mentioned above. It decouples these concerns by moving them into a service proxy. The

proxy is deployed into a separate process (called a sidecar) to provide isolation from business code. However, the sidecar is linked to the service - it's created with it and shares its lifecycle. Figure 6-7 shows this scenario.

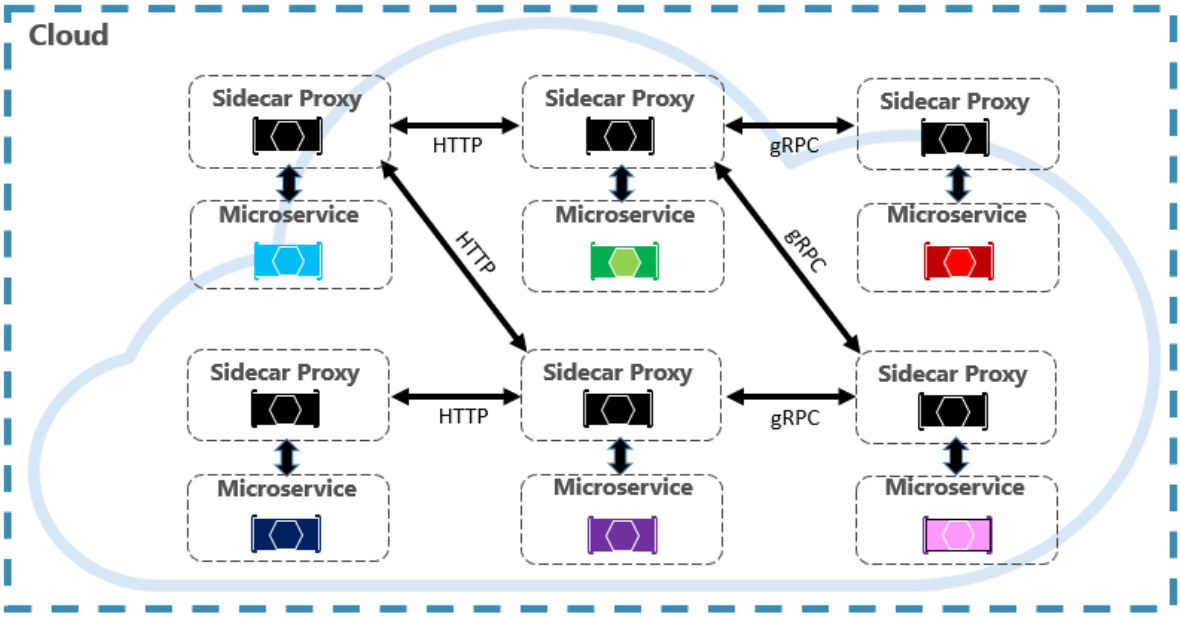


Figure 6-7. Service mesh with a side car

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 6-8 shows these components and their responsibilities.

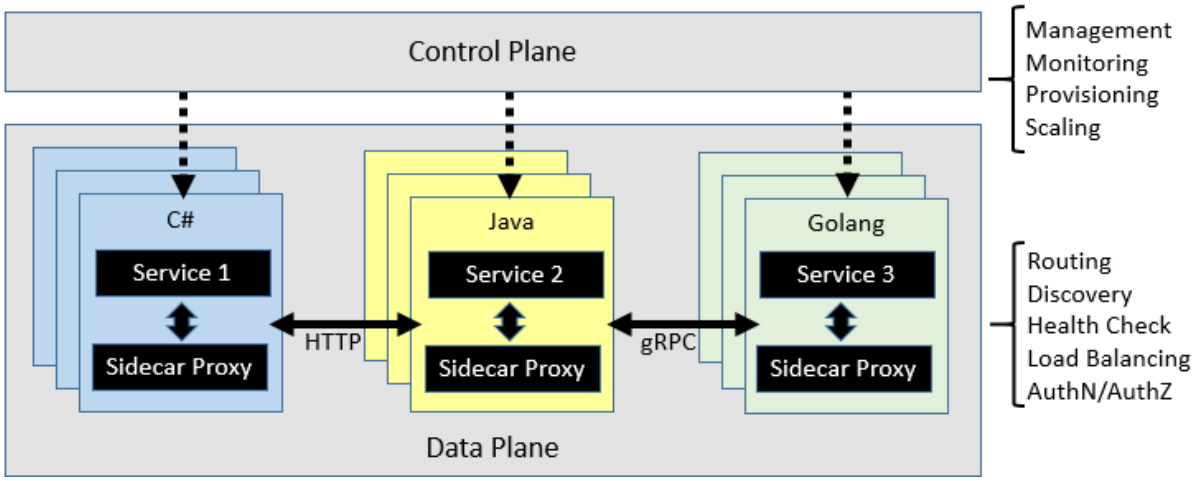


Figure 6-8. Service mesh control and data plane

Once configured, a service mesh is highly functional. It can retrieve a corresponding pool of instances from a service discovery endpoint. The mesh 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 will retry the request on another instance. If it returns errors, a mesh will evict the instance from the load-balancing pool and restate it after it heals. If a request times out, a mesh can fail and then retry the request. A mesh captures and emits metrics and distributed tracing to a centralized metrics system.

Istio and Envoy

While a few service mesh options currently exist, Istio is the most popular at the time of this writing. Istio is 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. The technology 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. It 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

References

- Polly
- Retry pattern
- Circuit Breaker pattern
- Resilience in Azure whitepaper
- network latency
- Redundancy
- geo-replication
- Azure Traffic Manager
- Autoscaling guidance
- Istio
- Envoy proxy

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 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.

Challenges when logging with cloud-native applications

In traditional applications, log files are 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`.

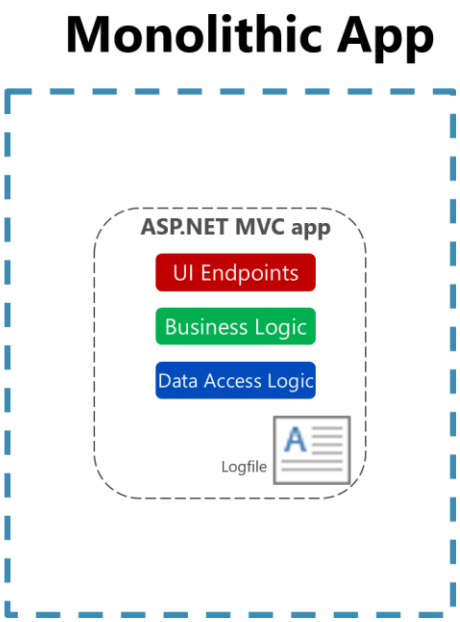


Figure 7-1. Logging to a file in a monolithic app.

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. Even simple scaling up of monolithic applications across multiple nodes can make it challenging to locate the appropriate file-based log file.

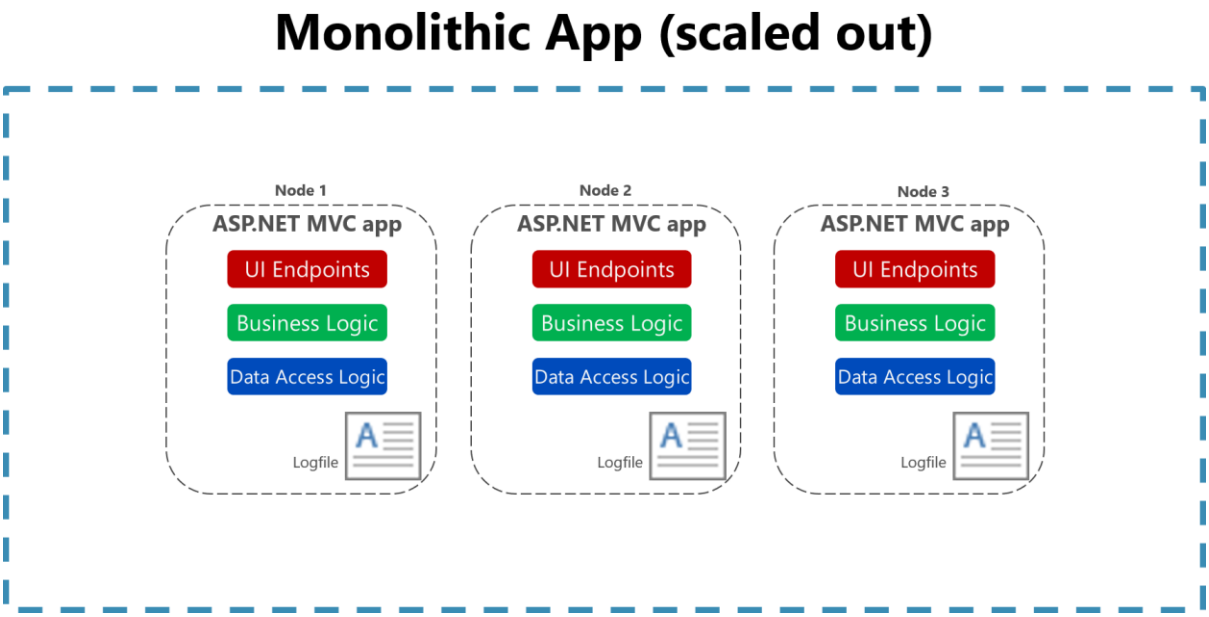


Figure 7-2. Logging to files in a scaled monolithic app.

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.

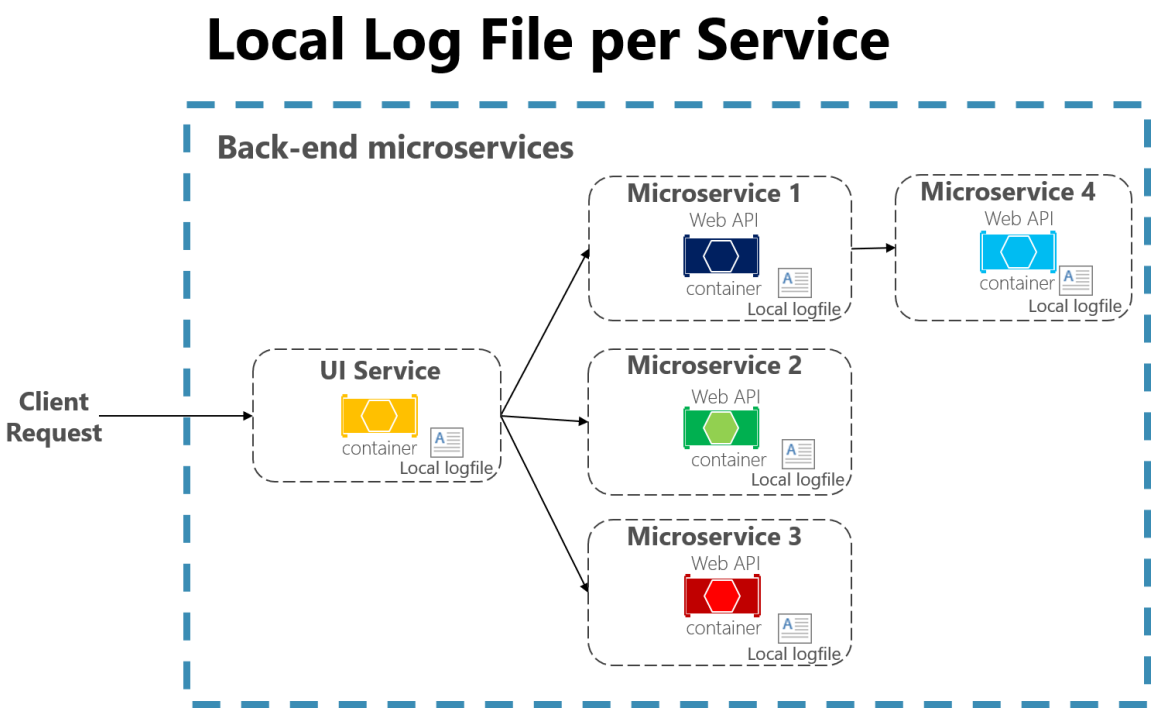


Figure 7-3. Logging to local files in a microservices app.

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 enough that specialized tools are needed to support effective use of the logs.

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. Some tools can be used to automatically provide this kind of logging.

Because of the challenges associated with using file-based logs in cloud-native apps, centralized logs are preferred. 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 7-4 demonstrates how a microservices architecture can leverage centralized logging as part of its workflow.

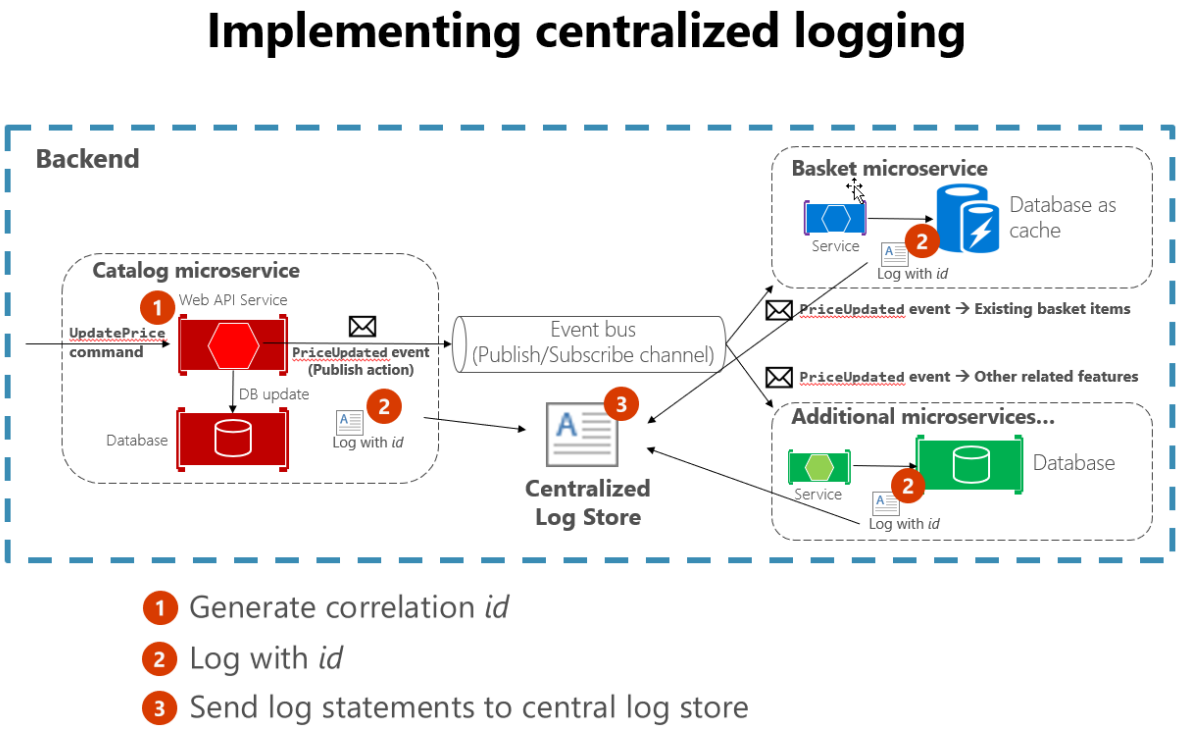


Figure 7-4. Logs from various sources are ingested into a centralized log store.

Challenges with detecting and responding to potential app health issues

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. You should know when problems occur with your application *before* users do, or before users notify you. Otherwise, the first you know about a problem may be when you notice an angry deluge of social media posts deriding your application or even your organization.

Some scenarios you may need to consider include:

- One service in your application keeps failing and restarting, resulting in intermittent slow responses.
- At some times of the day, your application's response time is slow.
- After a recent deployment, load on the database has tripled.

Implemented properly, monitoring can let you know about conditions that will lead to problems, letting you address underlying conditions before they result in any significant user impact.

Monitoring cloud-native apps

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 aspect 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.

Cloud-native monitoring tools provide real-time telemetry and insight into apps regardless of whether they're single-process monolithic applications or distributed microservice architectures. They include tools that allow collection of data from the app as well as tools for querying and displaying information about the app's health.

Challenges with reacting to critical problems in cloud-native apps

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.

Generally, alerts are layered on top of monitoring such that certain conditions trigger appropriate alerts to notify team members of urgent problems. Some scenarios that may require alerts include:

- One of your application's services is not responding after 1 minute of downtime.
- Your application is returning unsuccessful HTTP responses to more than 1% of requests.
- Your application's average response time for key endpoints exceeds 2000 ms.

Alerts in cloud-native apps

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 enough 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. Then, when true errors occur, 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.

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

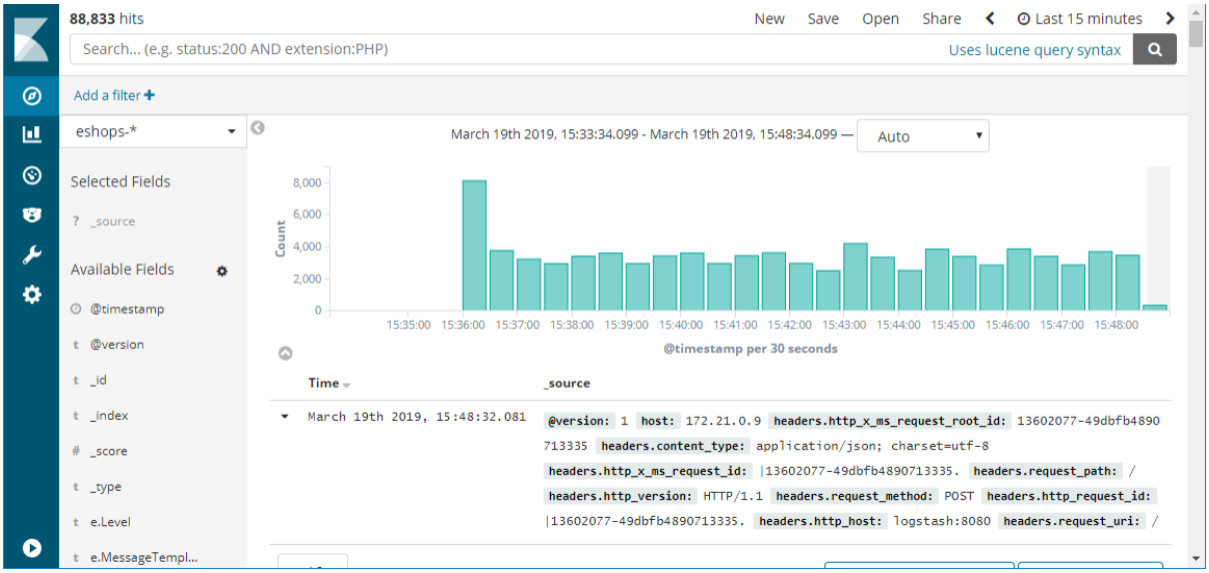


Figure 7-6. An example of a Kibana dashboard showing the results of a query against logs that are ingested from Kubernetes

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 even for 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 library allows for more intelligent filtering and searching. A sample Serilog configuration for writing to Logstash appears in Figure 7-7.

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

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

Logstash would use a configuration like the one shown in Figure 7-8.

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

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

Figure 7-8. 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 7-9.

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

Figure 7-9. 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 many 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.

Azure Monitor for Containers

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

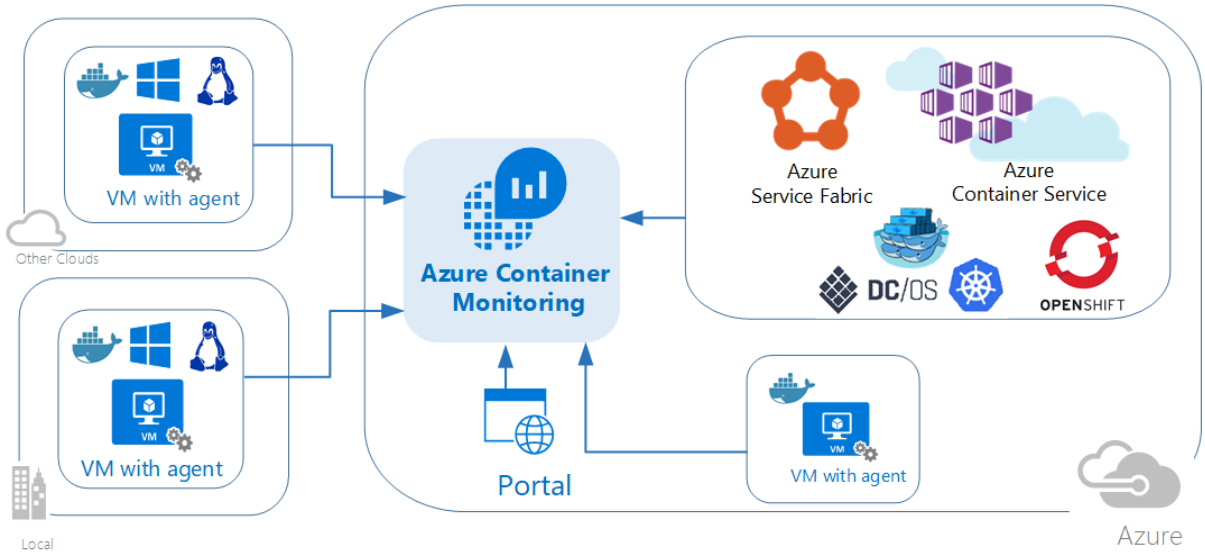


Figure 7-10. Consuming logs from various containers

Prometheus is a popular open source metric monitoring solution. It is part of the Cloud Native Compute Foundation. Typically, using Prometheus requires managing a Prometheus server with its own store. However, Azure Monitor for Containers provides direct integration with Prometheus metrics endpoints, so a separate server is not required.

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 approach ensures a parallel experience in environments that mix different log sources such as a hybrid Kubernetes/Azure Functions environment.

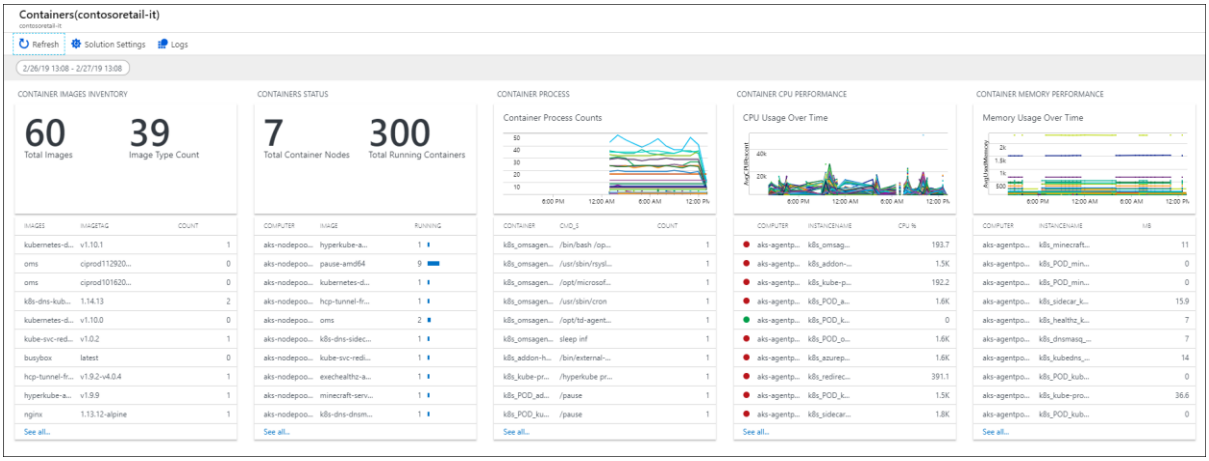


Figure 7-11. A sample dashboard showing logging and metric information from many running containers.

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 than 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. It helps you understand how your cloud-native services are performing and proactively identifies issues affecting them. Figure 7-12 presents a high level of view of Azure Monitor.

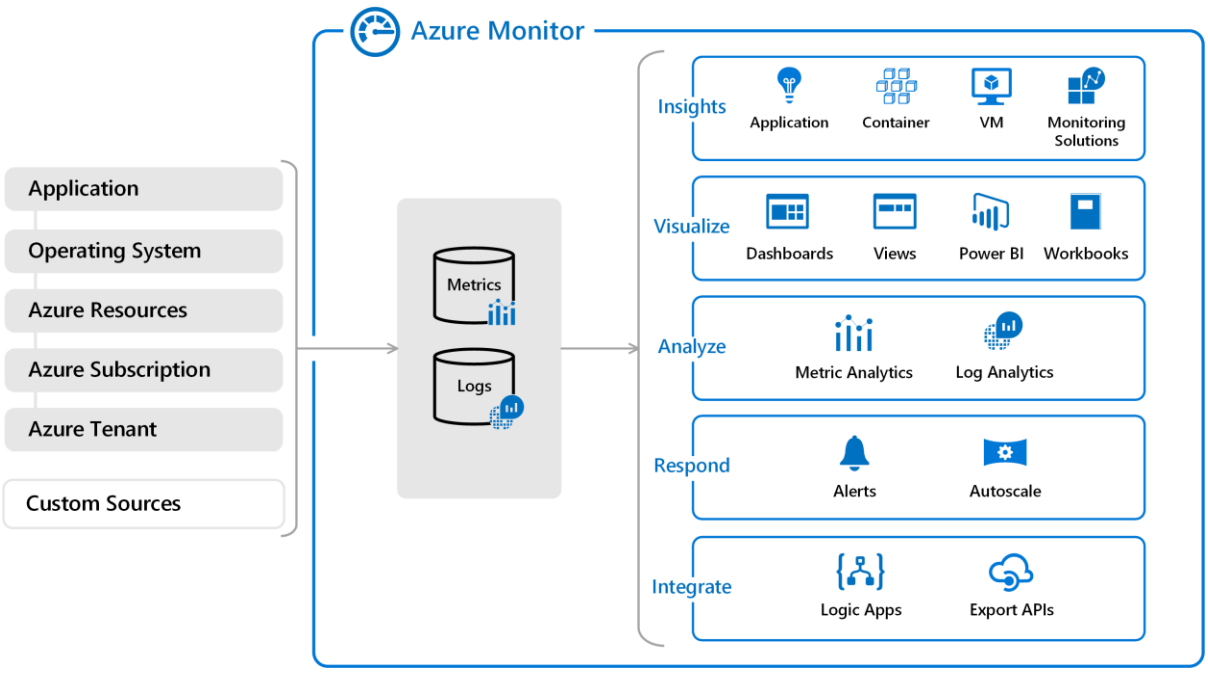


Figure 7-12. High-level view of Azure Monitor.

Gathering logs and metrics

The first step in any monitoring solution is to gather as much data as possible. The more data gathered, the deeper the insights. 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 configuration. 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 specific 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 simulate user 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 (SQL-like) query language called *Kusto* that can query records, summarize them, and even plot charts. For example, the following query will locate all 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 7-13 shows the results of this Application Insights Query.

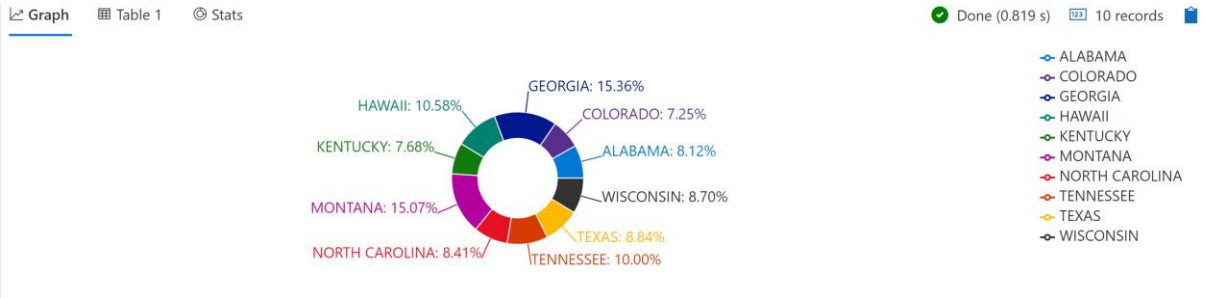


Figure 7-13. Application Insights query results.

There is a playground for experimenting with Kusto queries. 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.

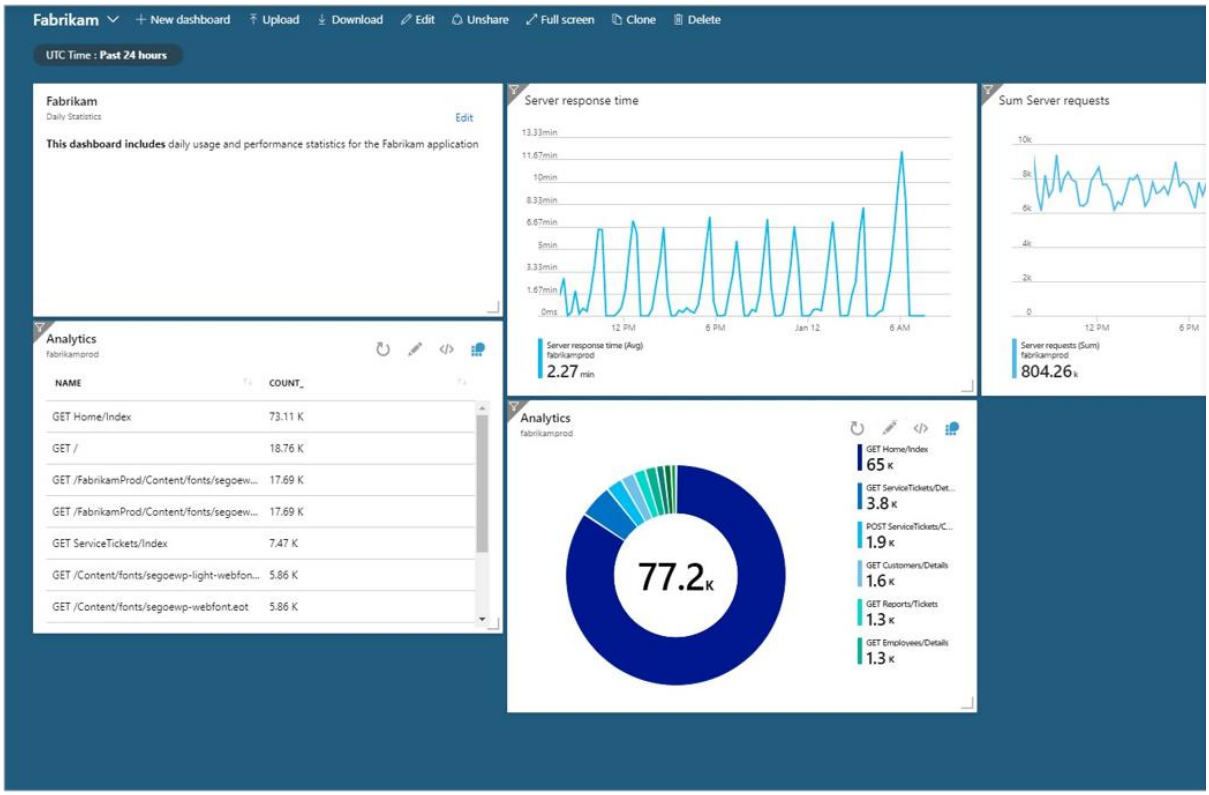


Figure 7-14. An example of Application Insights charts embedded in the main Azure Dashboard.

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 7-15. 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 approach 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.

References

- Azure Monitor

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 use 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 use 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 other 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 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.

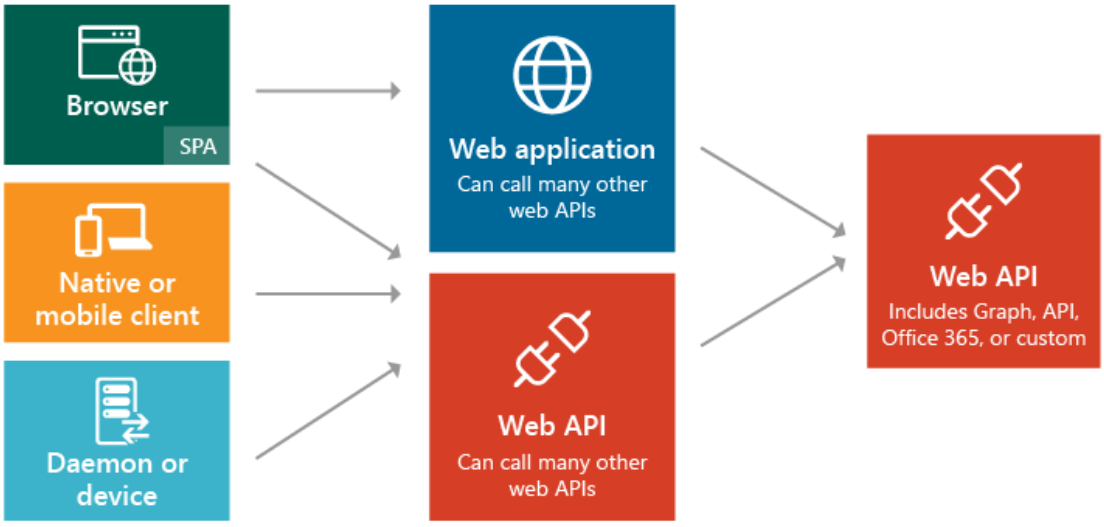


Figure 8-1. Application types and scenarios.

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 available under dual license:

- RPL - let's you use the IdentityServer4 free if used in open source work
- Paid - let's you use the IdentityServer4 in a commercial scenario

Please reach out to official Product's pricing page.

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 = "<insert 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 gets exposed, 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 process 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 writes 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 vulnerability 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 scenario 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

.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 most 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 fine-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.

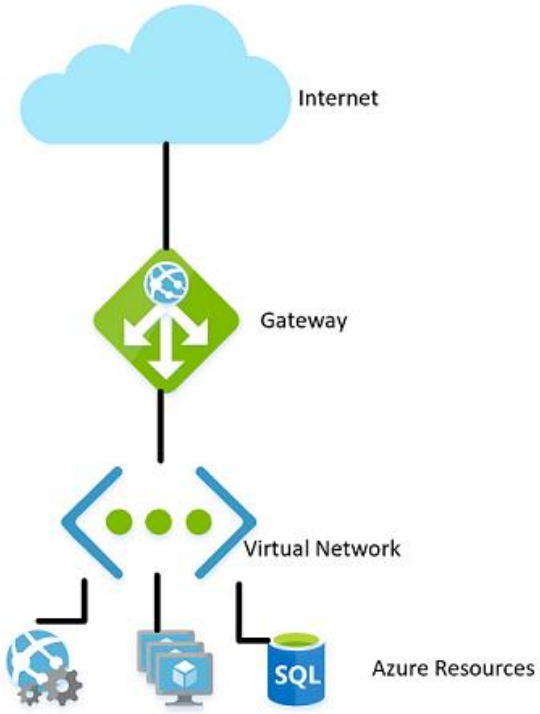


Figure 9-1. A virtual network in Azure.

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 scenario 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, a 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 approach 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.

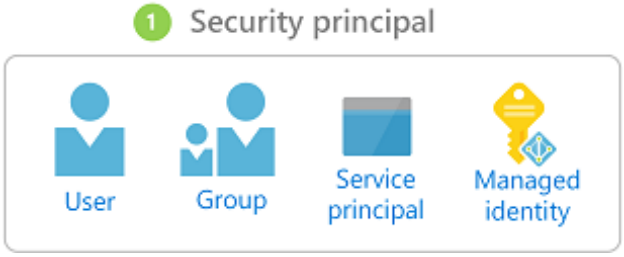


Figure 9-2. Different types of security principals.

- 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 aspect 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 a security principal has. Azure has a large number of built-in roles and users can define their own roles.

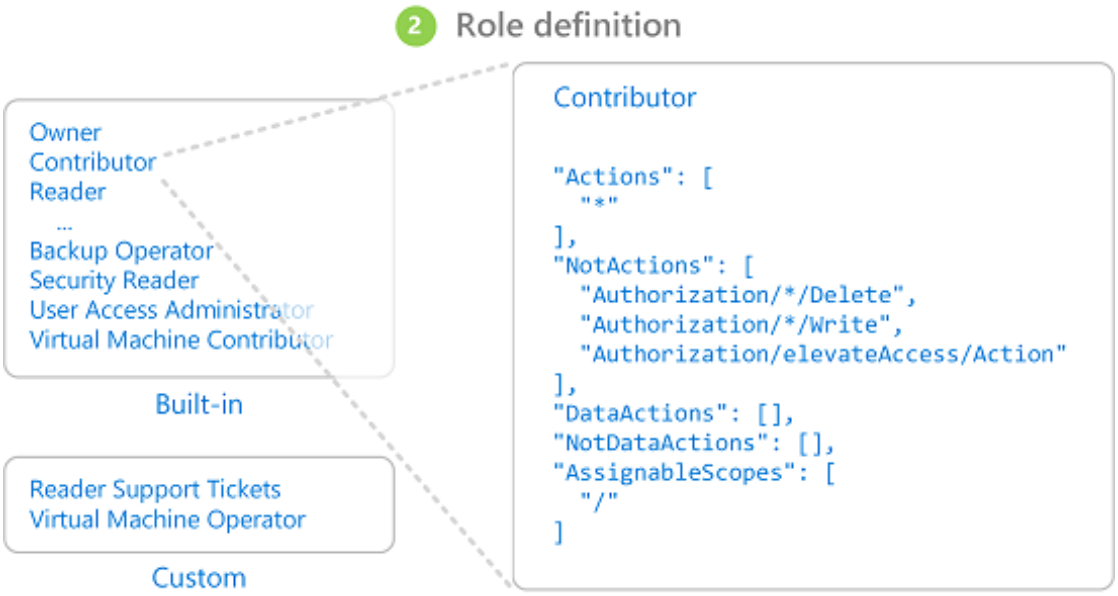


Figure 9-3. RBAC role definitions.

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 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 9-3.

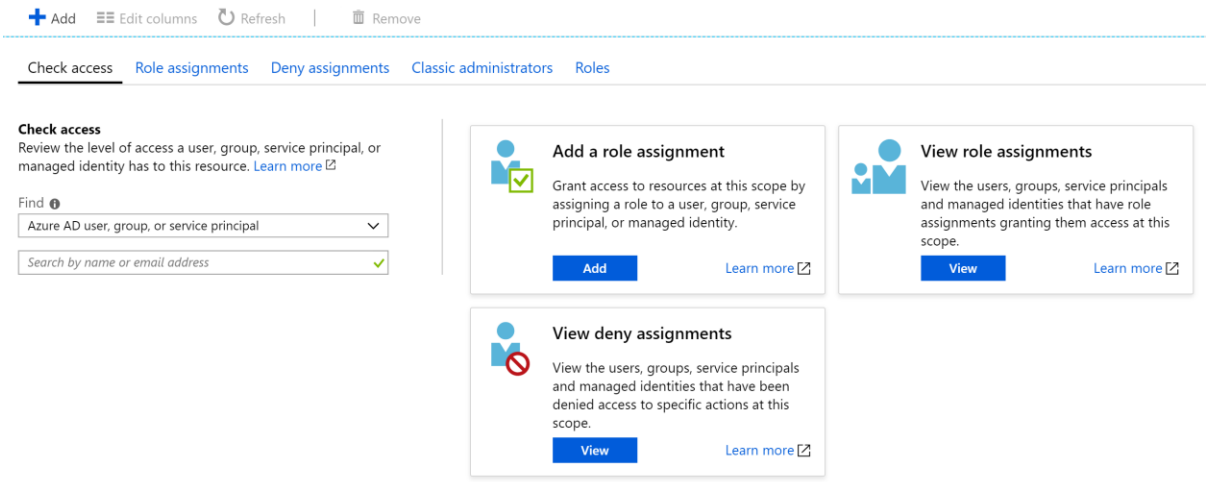


Figure 9-4. Permission calculator for an app service.

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. 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.

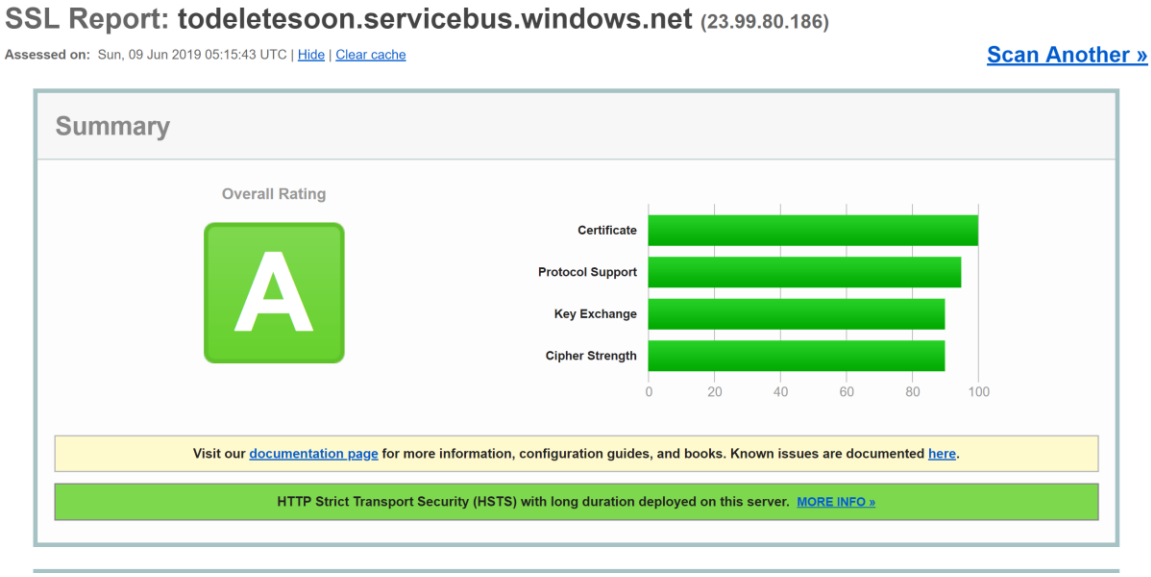


Figure 9-5. SSL labs report showing a score of A for a Service Bus endpoint.

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 the disk. The application code itself is loaded from some storage mechanism. Most applications also use some kind of a 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 Service runs 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.

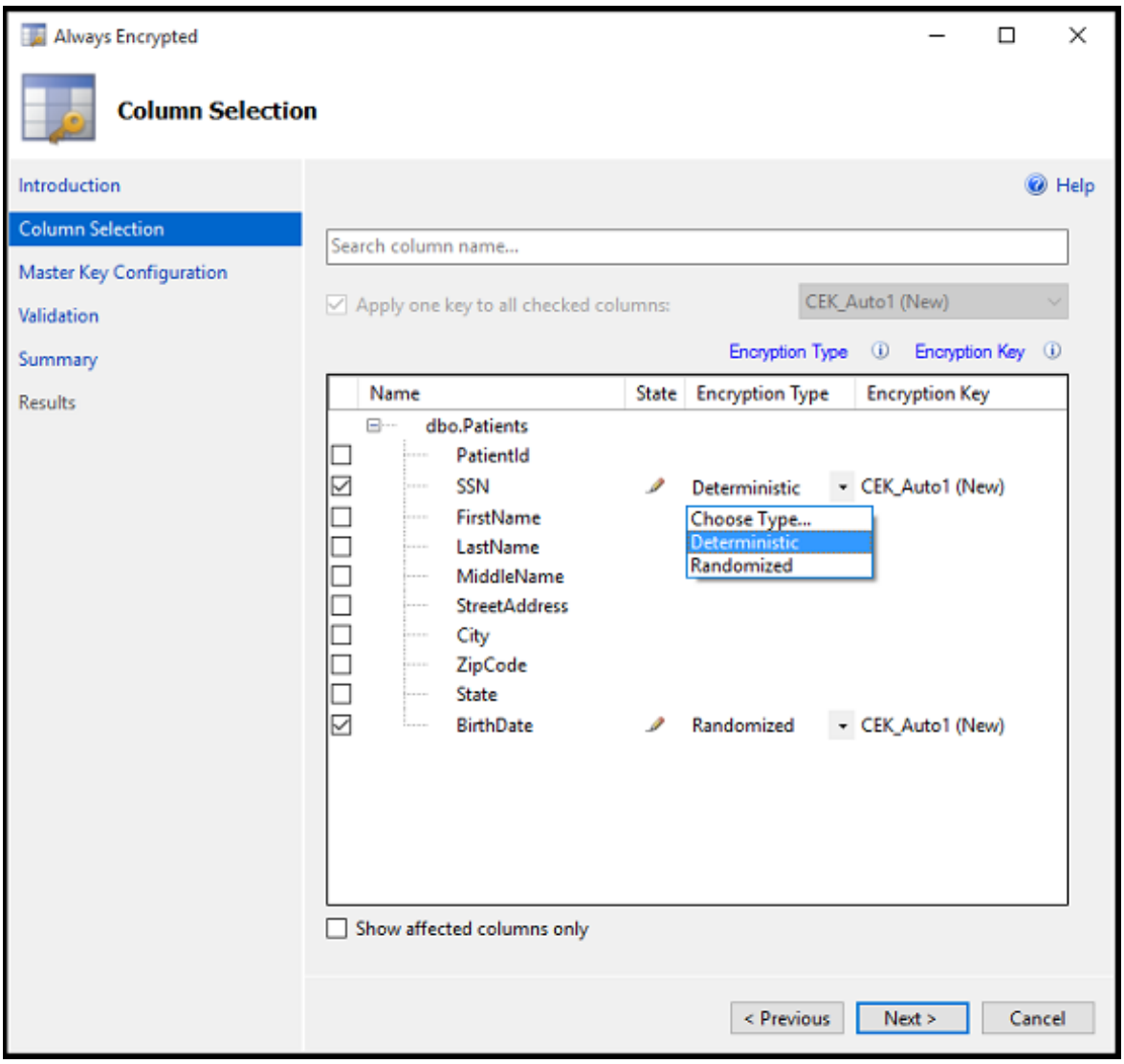


Figure 9-6. Selecting columns in a table to be encrypted using Always Encrypted.

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.

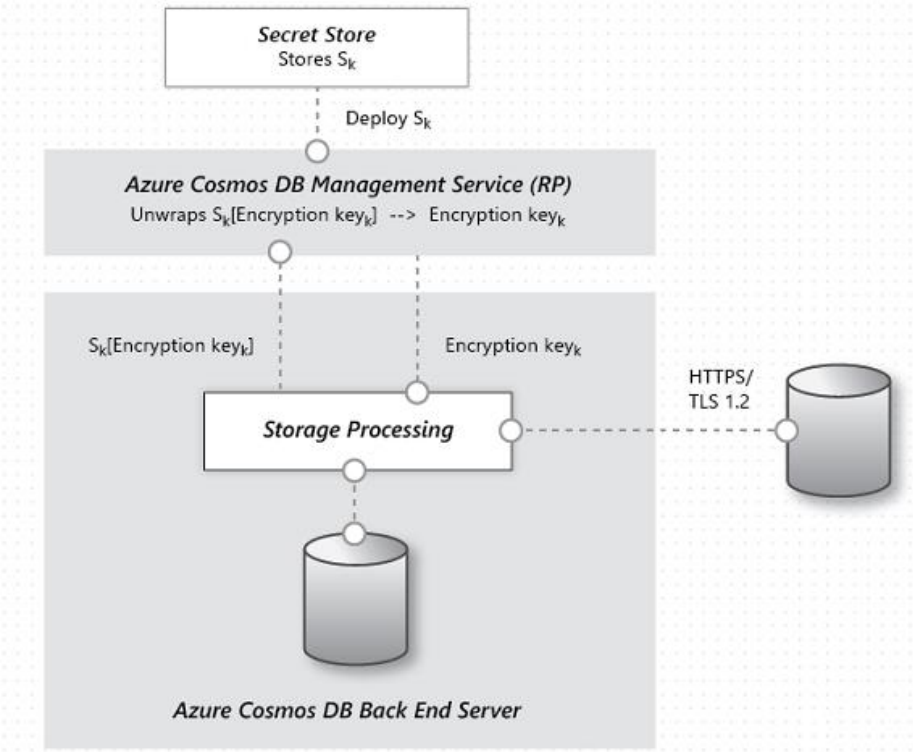


Figure 9-7. The flow of data encryption within Cosmos DB.

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. Examples include static analysis tools, encryption libraries, and security practices.

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.

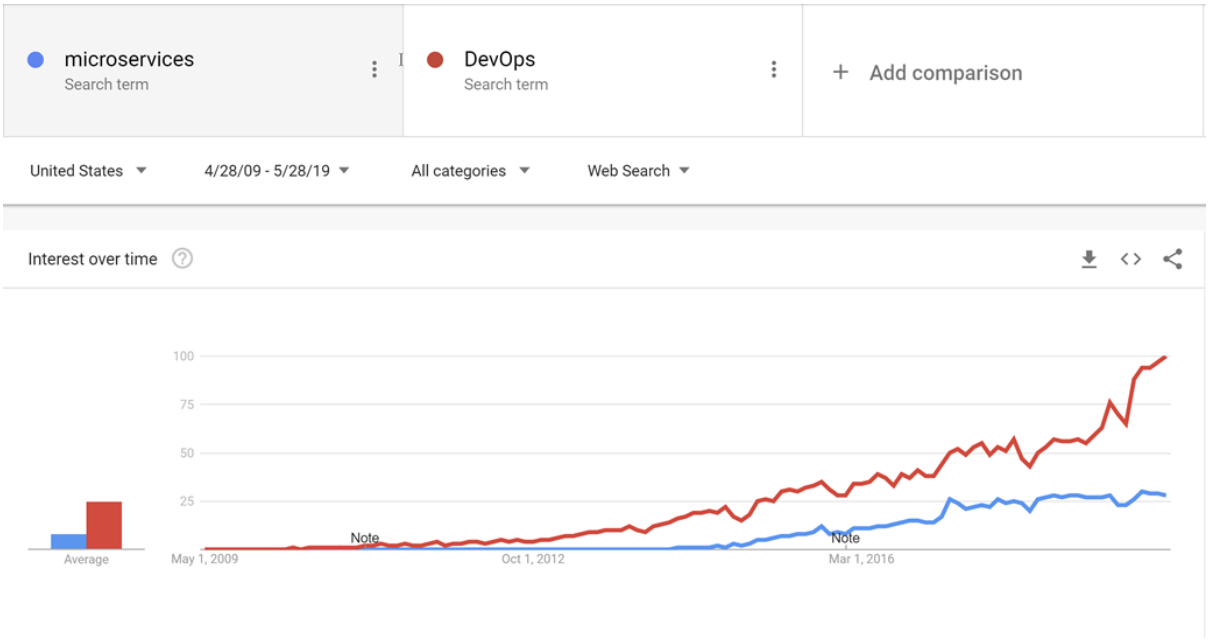


Figure 10-1 - DevOps and microservices.

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:

Figure 10-2 - Azure DevOps.

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 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 Pipelines - A build and release management system that supports tight integration with Azure. Builds can be run on various 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. Each of these components provides different advantages for cloud-native applications. The three most useful are repositories, boards, and pipelines. If users want to manage their source code in another repository stack, such as GitHub, but still take advantage of Azure Pipelines and other components, that's perfectly possible.

Fortunately, development teams have many options when selecting a repository. One of them is GitHub.

GitHub Actions

Founded in 2009, GitHub is a widely popular web-based repository for hosting projects, documentation, and code. Many large tech companies, such as Apple, Amazon, Google, and mainstream corporations use GitHub. GitHub uses the open-source, distributed version control system named Git as its foundation. On top, it then adds its own set of features, including defect tracking, feature and pull requests, tasks management, and wikis for each code base.

As GitHub evolves, it too is adding DevOps features. For example, GitHub has its own continuous integration/continuous delivery (CI/CD) pipeline, called `GitHub Actions`. GitHub Actions is a community-powered workflow automation tool. It lets DevOps teams integrate with their existing tooling, mix and match new products, and hook into their software lifecycle, including existing CI/CD partners."

GitHub has over 40 million users, making it the largest host of source code in the world. In October of 2018, Microsoft purchased GitHub. Microsoft has pledged that GitHub will remain an open platform that any developer can plug into and extend. It continues to operate as an independent company. GitHub offers plans for enterprise, team, professional, and free accounts.

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 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.

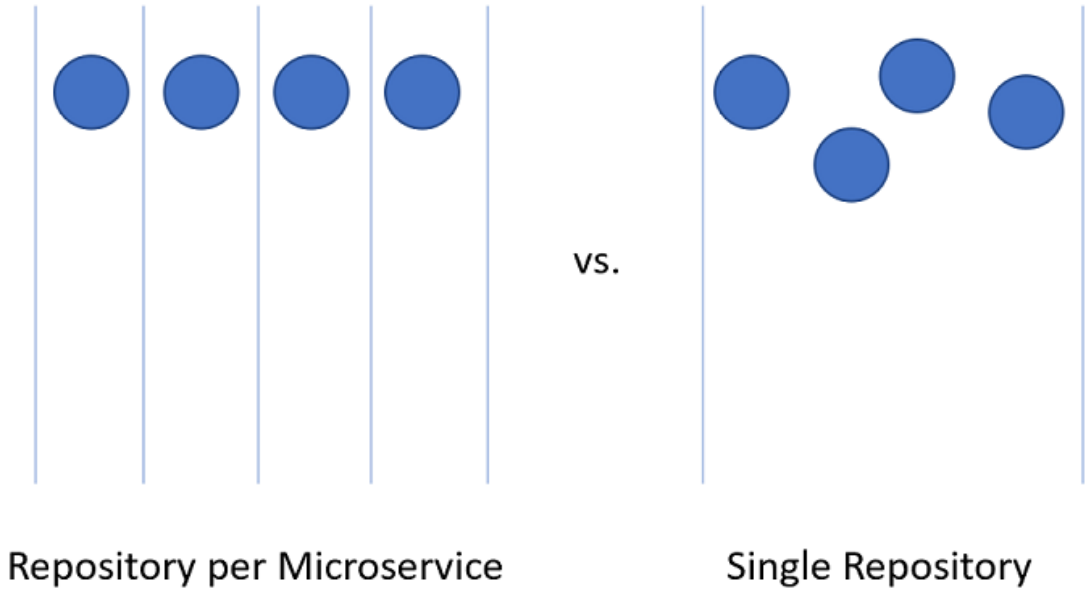


Figure 10-3 - One vs. many repositories.

Repository per microservice

At first glance, this approach 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 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 activity can be difficult to coordinate.

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 approach 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 activity 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 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 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.

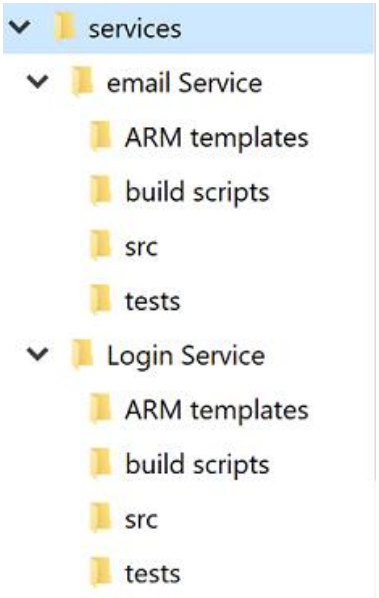


Figure 10-4 - Standard directory structure.

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, 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.

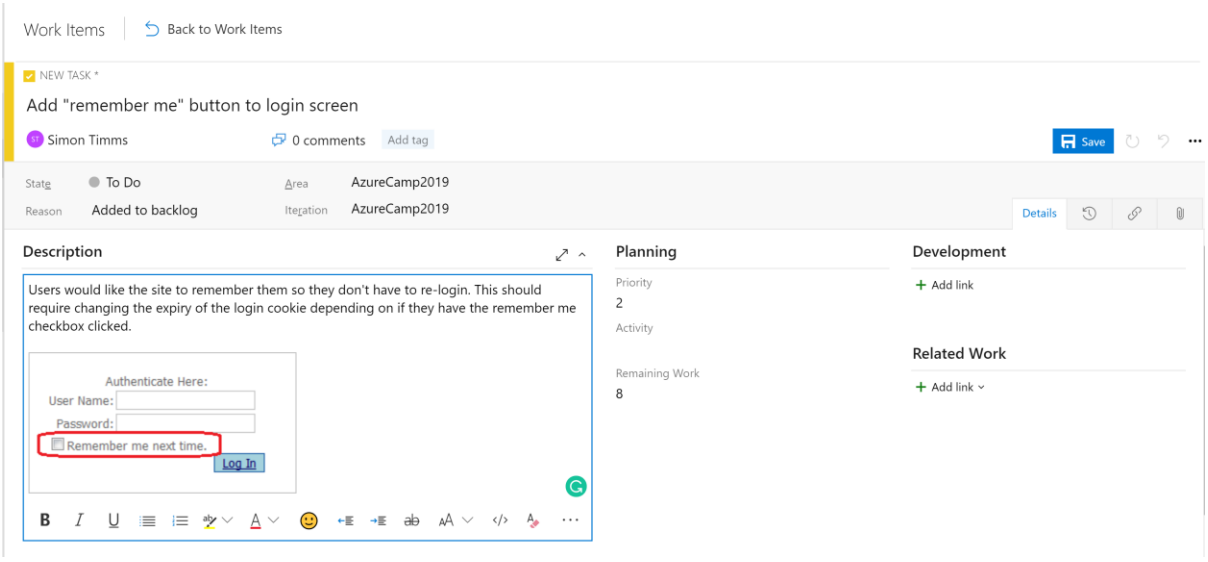


Figure 10-5 - Task in Azure DevOps.

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 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.

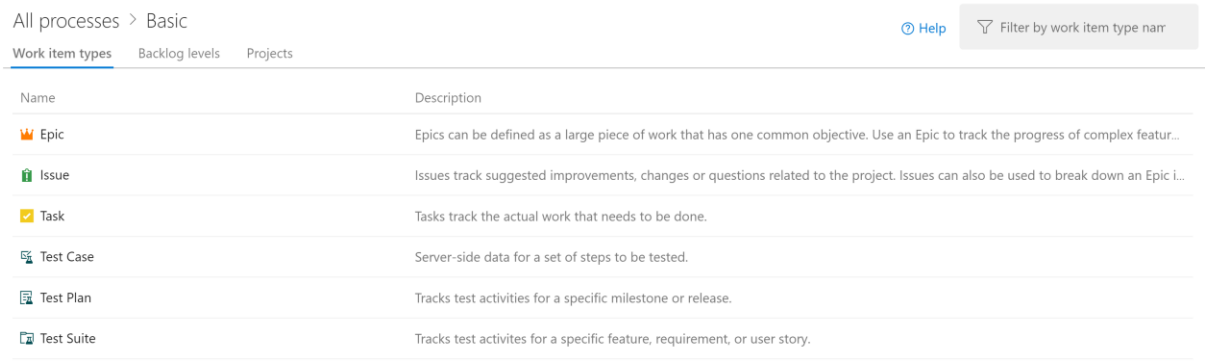


Figure 10-6 - Work item in Azure DevOps.

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.

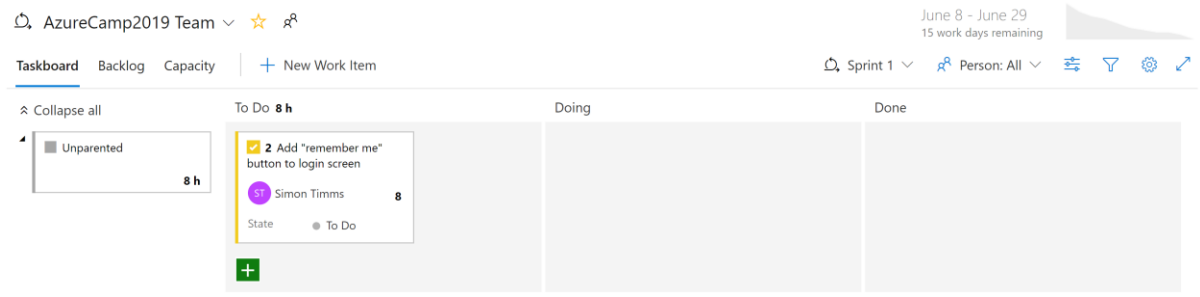


Figure 10-7 - Board in Azure DevOps.

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 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.

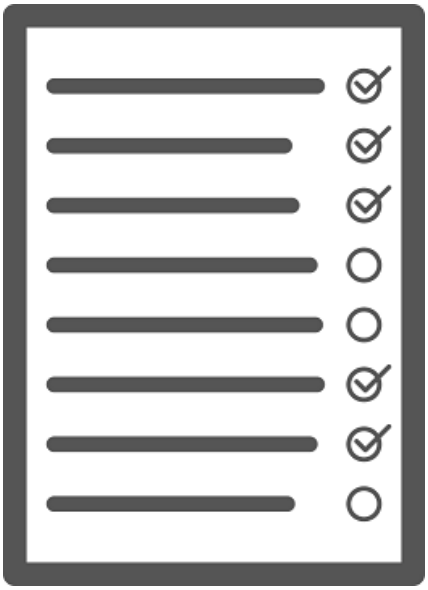


Figure 10-8 - Checklist.

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 10-9.

```
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 10-9 - 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 Artifacts 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. The Releases tool makes 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.

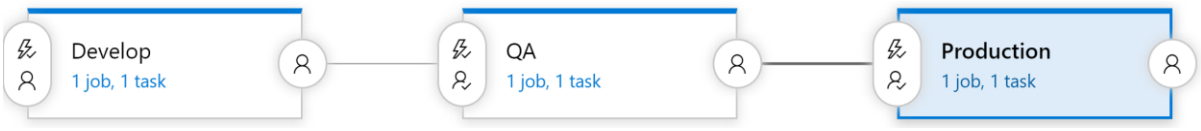


Figure 10-10 - Release pipeline

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. The Releases tool 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!

Feature flags

In chapter 1, we affirmed that cloud native is much about speed and agility. Users expect rapid responsiveness, innovative features, and zero downtime. `Feature flags` are a modern deployment technique that helps increase agility for cloud-native applications. They enable you to deploy new features into a production environment, but restrict their availability. With the flick of a switch, you can activate a new feature for specific users without restarting the app or deploying new code. They separate the release of new features from their code deployment.

Feature flags are built upon conditional logic that control visibility of functionality for users at runtime. In modern cloud-native systems, it's common to deploy new features into production early, but test them with a limited audience. As confidence increases, the feature can be incrementally rolled out to wider audiences.

Other use cases for feature flags include:

- Restrict premium functionality to specific customer groups willing to pay higher subscription fees.
- Stabilize a system by quickly deactivating a problem feature, avoiding the risks of a rollback or immediate hotfix.
- Disable an optional feature with high resource consumption during peak usage periods.
- Conduct `experimental feature releases` to small user segments to validate feasibility and popularity.

Feature flags also promote `trunk-based` development. It's a source-control branching model where developers collaborate on features in a single branch. The approach minimizes the risk and complexity of merging large numbers of long-running feature branches. Features are unavailable until activated.

Implementing feature flags

At its core, a feature flag is a reference to a simple `decision object`. It returns a Boolean state of `on` or `off`. The flag typically wraps a block of code that encapsulates a feature capability. The state of the flag determines whether that code block executes for a given user. Figure 10-11 shows the implementation.

```
if (featureFlag) {
    // Run this code block if the featureFlag value is true
} else {
    // Run this code block if the featureFlag value is false
}
```

Figure 10-11 - Simple feature flag implementation.

Note how this approach separates the decision logic from the feature code.

In chapter 1, we discussed the `Twelve-Factor App`. The guidance recommended keeping configuration settings external from application executable code. When needed, settings can be read in from the external source. Feature flag configuration values should also be independent from their codebase. By externalizing flag configuration in a separate repository, you can change flag state without modifying and redeploying the application.

Azure App Configuration provides a centralized repository for feature flags. With it, you define different kinds of feature flags and manipulate their states quickly and confidently. You add the App Configuration client libraries to your application to enable feature flag functionality. Various programming language frameworks are supported.

Feature flags can be easily implemented in an ASP.NET Core service. Installing the .NET Feature Management libraries and App Configuration provider enable you to declaratively add feature flags to your code. They enable `FeatureGate` attributes so that you don't have to manually write if statements across your codebase.

Once configured in your Startup class, you can add feature flag functionality at the controller, action, or middleware level. Figure 10-12 presents controller and action implementation:

```
[FeatureGate(MyFeatureFlags.FeatureA)]
public class ProductController : Controller
{
    ...
}
```

```
[FeatureGate(MyFeatureFlags.FeatureA)]
public IActionResult UpdateProductStatus()
{
    return ObjectResult(ProductDto);
}
```

Figure 10-12 - Feature flag implementation in a controller and action.

If a feature flag is disabled, the user will receive a 404 (Not Found) status code with no response body.

Feature flags can also be injected directly into C# classes. Figure 10-13 shows feature flag injection:

```
public class ProductController : Controller
{
    private readonly IFeatureManager _featureManager;

    public ProductController(IFeatureManager featureManager)
    {
        _featureManager = featureManager;
    }
}
```

Figure 10-13 - Feature flag injection into a class.

The Feature Management libraries manage the feature flag lifecycle behind the scenes. For example, to minimize high numbers of calls to the configuration store, the libraries cache flag states for a specified duration. They can guarantee the immutability of flag states during a request call. They also offer a `Point-in-time snapshot`. You can reconstruct the history of any key-value and provide its past value at any moment within the previous seven days.

Infrastructure as code

Cloud-native systems embrace microservices, containers, and modern system design to achieve speed and agility. They provide automated build and release stages to ensure consistent and quality code. But, that's only part of the story. How do you provision the cloud environments upon which these systems run?

Modern cloud-native applications embrace the widely accepted practice of Infrastructure as Code, or `IaC`. With IaC, you automate platform provisioning. You essentially apply software engineering practices such as testing and versioning to your DevOps practices. Your infrastructure and deployments are automated, consistent, and repeatable. Just as continuous delivery automated the traditional model of manual deployments, Infrastructure as Code (IaC) is evolving how application environments are managed.

Tools like Azure Resource Manager (ARM), Terraform, and the Azure Command Line Interface (CLI) enable you to declaratively script the cloud infrastructure you require.

Azure Resource Manager templates

ARM stands for Azure Resource Manager. It's an API provisioning engine that is built into Azure and exposed as an API service. ARM enables you to deploy, update, delete, and manage the resources contained in Azure resource group in a single, coordinated operation. You provide the engine with a JSON-based template that specifies the resources you require and their configuration. ARM automatically orchestrates the deployment in the correct order respecting dependencies. The engine ensures idempotency. If a desired resource already exists with the same configuration, provisioning will be ignored.

Azure Resource Manager templates are a JSON-based language for defining various resources in Azure. The basic schema looks something like Figure 10-14.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-
01/deploymentTemplate.json#",
  "contentVersion": "",
  "apiProfile": "",
  "parameters": {  },
  "variables": {  },
  "functions": [  ],
  "resources": [  ],
  "outputs": {  }
}
```

Figure 10-14 - 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 10-15 - An example of a storage account defined in a Resource Manager template

An ARM template can be parameterized with dynamic environment and configuration information. Doing so enables it to be reused to define different environments, such as development, QA, or production. Normally, the template creates all resources within a single Azure resource group. It's possible to define multiple resource groups in a single Resource Manager template, if needed. You can delete all resources in an environment by deleting the resource group itself. 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 examples of ARM templates available in the Azure Quickstart Templates project on GitHub. They can help accelerate creating a new template or modifying an existing one.

Resource Manager templates can be run in many of ways. Perhaps the simplest way is to simply paste them into the Azure portal. For experimental deployments, this method can be 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 reconcile differences between the current resources and those defined in the template. Resources will then be created or altered so they match what is defined in the template.

Terraform

Cloud-native applications are often constructed to be `cloud agnostic`. Being so means the application isn't tightly coupled to a particular cloud vendor and can be deployed to any public cloud.

Terraform is a commercial templating tool that can provision cloud-native applications across all the major cloud players: Azure, Google Cloud Platform, AWS, and AliCloud. Instead of using JSON as the template definition language, it uses the slightly more terse HCL (Hashicorp Configuration Language).

An example Terraform file that does the same as the previous Resource Manager template (Figure 10-15) is shown in Figure 10-16:

```
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 10-16 - An example of a Resource Manager template

Terraform also provides intuitive error messages for problem templates. There's also a handy validate task that can be used in the build phase to catch template errors early.

As with Resource Manager templates, command-line tools are available to deploy Terraform templates. There are also community-created tasks in Azure Pipelines that can validate and apply Terraform templates.

Sometimes Terraform and ARM templates output meaningful values, such as a connection string to a newly created database. This information can be captured in the build pipeline and used in subsequent tasks.

Azure CLI Scripts and Tasks

Finally, you can leverage Azure CLI to declaratively script your cloud infrastructure. Azure CLI scripts can be created, found, and shared to provision and configure almost any Azure resource. The CLI is simple to use with a gentle learning curve. Scripts are executed within either PowerShell or Bash. They're also straightforward to debug, especially when compared with ARM templates.

Azure CLI scripts work well when you need to tear down and redeploy your infrastructure. Updating an existing environment can be tricky. Many CLI commands aren't idempotent. That means they'll recreate the resource each time they're run, even if the resource already exists. It's always possible to

add code that checks for the existence of each resource before creating it. But, doing so, your script can become bloated and difficult to manage.

These scripts can also be embedded in Azure DevOps pipelines as `Azure CLI tasks`. Executing the pipeline invokes the script.

Figure 10-17 shows a YAML snippet that lists the version of Azure CLI and the details of the subscription. Note how Azure CLI commands are included in an inline script.

```
- task: AzureCLI@2
  displayName: Azure CLI
  inputs:
    azureSubscription: <Name of the Azure Resource Manager service connection>
    scriptType: ps
    scriptLocation: inlineScript
    inlineScript: |
      az --version
      az account show
```

Figure 10-17 - Azure CLI script

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."

Cloud Native Application Bundles

A key property of cloud-native applications is that they leverage the capabilities of the cloud to speed up development. This design 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 large variety 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 by many 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 aspect 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 10-18 - 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 various 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.

References

- Azure DevOps
- Azure Resource Manager
- Terraform
- Azure CLI

Summary

In summary, here are important conclusions from this guide:

- **Cloud-native** is about designing modern applications that embrace rapid change, large scale, and resilience, in modern, dynamic environments such as public, private, and hybrid clouds.
- The **Cloud Native Computing Foundation** **(CNCF)** is an influential open-source consortium of over 300 major corporations. It's responsible for driving the adoption of cloud-native computing across technology and cloud stacks.
- **CNCF guidelines** recommend that cloud-native applications embrace six important pillars as shown in Figure 11-1:

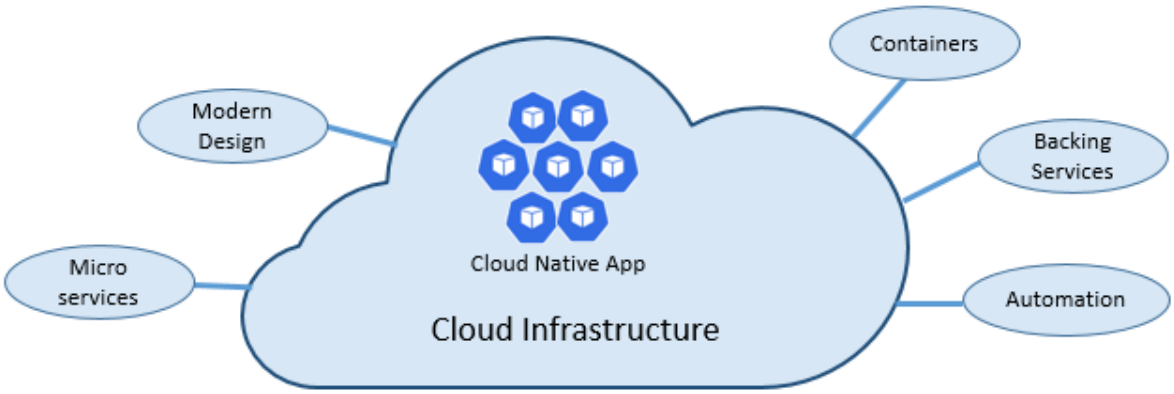


Figure 11-1. Cloud-native foundational pillars

- These cloud-native pillars include:
- The cloud and its underlying service model
- Modern design principles
- Microservices
- Containerization and container orchestration
- Cloud-based backing services, such as databases and message brokers
- Automation, including Infrastructure as Code and code deployment
- **Kubernetes** is the hosting environment of choice for most cloud-native applications. Smaller, simple services are sometimes hosted in serverless platforms, such as Azure Functions. Among many key automation features, both environments provide automatic scaling to handle fluctuating workload volumes.

- **Service communication** becomes a significant design decision when constructing a cloud-native application. Applications typically expose an API gateway to manage front-end client communication. Then backend microservices strive to communicate with each other implementing asynchronous communication patterns, when possible.
- **gRPC** is a modern, high-performance framework that evolves the age-old remote procedure call (RPC) protocol. Cloud-native applications often embrace gRPC to streamline messaging between back-end services. gRPC uses HTTP/2 for its transport protocol. It can be up to 8x faster than JSON serialization with message sizes 60-80% smaller. gRPC is open source and managed by the Cloud Native Computing Foundation (CNCF).
- **Distributed data** is a model often implemented by cloud-native applications. Applications segregate business functionality into small, independent microservices. Each microservice encapsulates its own dependencies, data, and state. The classic shared database model evolves into one of many smaller databases, each aligning with a microservice. When the smoke clears, we emerge with a design that exposes a `database-per-microservice` model.
- **No-SQL databases** refer to high-performance, non-relational data stores. They excel in their ease-of-use, scalability, resilience, and availability characteristics. High volume services that require sub second response time favor NoSQL datastores. The proliferation of NoSQL technologies for distributed cloud-native systems can't be overstated.
- **NewSQL** is an emerging database technology that combines the distributed scalability of NoSQL and the ACID guarantees of a relational database. NewSQL databases target business systems that must process high-volumes of data, across distributed environments, with full transactional/ACID compliance. The Cloud Native Computing Foundation (CNCF) features several NewSQL database projects.
- **Resiliency** is the ability of your system to react to failure and still remain functional. Cloud-native systems embrace distributed architecture where failure is inevitable. Applications must be constructed to respond elegantly to failure and quickly return to a fully functioning state.
- **Service meshes** are a configurable infrastructure layer with built-in capabilities to handle service communication and other cross-cutting challenges. They decouple cross-cutting responsibilities from your business code. These responsibilities move into a service proxy. Referred to as the `Sidecar pattern`, the proxy is deployed into a separate process to provide isolation from your business code.
- **Observability** is a key design consideration for cloud-native applications. As services are distributed across a cluster of nodes, centralized logging, monitoring, and alerts, become mandatory. Azure Monitor is a collection of cloud-based tools designed to provide visibility into the state of your system.
- **Infrastructure as Code** is a widely accepted practice that automates platform provisioning. Your infrastructure and deployments are automated, consistent, and repeatable. Tools like Azure Resource Manager, Terraform, and the Azure CLI, enable you to declaratively script the cloud infrastructure you require.

- **Code automation** is a requirement for cloud-native applications. Modern CI/CD systems help fulfill this principle. They provide separate build and deployment steps that help ensure consistent and quality code. The build stage transforms the code into a binary artifact. The release stage picks up the binary artifact, applies external environment configuration, and deploys it to a specified environment. Azure DevOps and GitHub are full-featured DevOps environments.